SOFSEM'92 — Ždiar, Magura, Vysoké Tatry, 22.11. — 4.12.1992

TOP-DOWN DESIGN OF TOTALLY CORRECT SHARED-STATE PARALLEL PROGRAMS

Ketil Stølen

Institute für Informatik, der Technischen Universität,
Postfach 20 24 20, Arcisstrasse 21, W-8000 München 2
email:stoelen@informatik.tu-muenchen.de

Abstract: VDM, the Vienna Development Method, has been used successfully to aid software design in a vide variety of areas. However, VDM is basically a technique for the design of sequential programs. The object of this paper is to explain how shared-state parallel programs can be designed in a similar style. Rules for program decomposition and data refinement are given. The importance of compositionality is stressed. The presentation is informal and based on simple examples. Formal definitions can be found in a separate appendix at the end of the paper.
Key words: program-design, top-down, compositionality, decomposition, data refinement, total correctness, fairness.

1. INTRODUCTION

Appeared in: Proc. SOFSEM'92, pages 291-310, Czechoslovak Society for Computer Science 1992.

As in Hoare logic [5], a VDM [7] specification is split into two predicates: A *pre-condition* which characterises the initial state, and a *post-condition* constraining the terminal state. The pre-condition can be thought of as an *assumption* about the *environment* in which the specified *component* is supposed to run. The post-condition, on the other hand, is a *commitment* the component must fulfil when executed in an environment which satisfies the pre-condition. Thus, a pre/post specification can be said to be of an assumption/commitment form.
For pre/post specifications it is usual to distinguish between *partial* and *total* correctness. Partial correctness means that whenever the initial state satisfies the pre-condition and the component terminates, then the terminal state satisfies the post-condition. Total correctness is partial correctness plus the additional requirement that the component terminates whenever the initial state satisfies the pre-condition. In VDM, and also in this paper, the component is required to be totally correct with respect to its specification.
The specification

$$x\colon [x = 0, x = 2], \tag{1}$$

requires that whenever the pre-condition $x = 0$ holds in the initial state, then the component terminates in a state such that $x = 2$. On the other hand, the specification

$$x\colon [\mathsf{true}, x = \overleftarrow{x} + 3] \tag{2}$$

requires that for any initial state, the component terminates in a state such that the initial value of x is increased with 3. Thus, $\overleftarrow{x}$ in the post-condition refers to the initial value of x.

Now, assume that z_1 and z_2 are programs which satisfy 1 and 2, respectively. Then their sequential composition $z_1; z_2$ satisfies

$$x\colon [x = 0, x = 5]. \tag{3}$$

To see this, it is enough to observe that it follows from 1 that if the initial state satisfies $x = 0$ then z_1 terminates in a state where $x = 2$, in which case 2 implies that z_2 terminates in a state such that $x = 5$. Thus, that $z_1; z_2$ satisfies 3 can be deduced from 1 and 2 without knowing the internal structure of z_1 and z_2. This is often referred to as *compositionality* [17]. Basically, a design method is compositional if the specification of a component can be deduced from the specifications of its immediate subcomponents, without knowledge of the internal structure of those subcomponents. This gives a number of advantages:

- Design decisions can more often be verified at the point were they are taken. This reduces the amount of backtracking needed during program design.
- Specifications can be split into subspecifications which can be implemented separately. This is of importance when large software modules are implemented by different groups.
- The specification of a component contains all information needed to use and combine it with other components.

An interesting question at this point: Can the specifications 1 and 2 be used to make a similar prediction about the parallel composition $z_1 \parallel z_2$? Unfortunately, the answer is "no". Before explaining this in detail, it is necessary to fix the level of granularity: Throughout this paper assignments and boolean tests are required to be atomic, in other words, they can be thought of as being executed in isolation. This means that if

z_1::
    if $x = 0$ then $x := 2$ else $x := m$ fi,

z_2::
    $x := x + 1; x := x + 1; x := x + 1,$

then z_1 and z_2 perform respectively two and three atomic steps.
It is clear that z_1 and z_2 satisfy 1 and 2, respectively. Moreover, given $x = 0$ as pre-condition, then

$$x\colon [x = 0, 2 \leq x \leq 5 \vee m \leq x \leq m + 2]$$

is the strongest possible specification of $z_1 \parallel z_2$. Since z_1 satisfies 1 for any constant m, it follows that without knowing the internal structure of z_1 and z_2 it is not possible to make any prediction about the terminal value of x. But this is not the worst — we cannot even be sure of termination. For example, if

z_1::
    while $x \neq 2$ do $x := x + 1$ od,

z_2::
    $x := x + 3,$

then z_1 and z_2 satisfy 1 and 2, respectively, but their parallel composition is not guaranteed to terminate — not even if the initial state satisfies $x = 0$.
It follows from the above discussion that the traditional pre/post specifications are insufficient for design of parallel programs in a compositional style. The problem with parallel composition is that when z_1 and z_2 are executed in parallel, then they have both write access to the

same global variable x. This is known as shared-state *interference*. To achieve compositionality in the case of parallel composition it is necessary to find a way to specify interference.

2. INTERFERENCE

The question now is of course: How can we get around this problem? Before giving an answer, let us have a look at a well-known proof method for shared-state parallel programs — the so-called Owicki/Gries method [8]. Let

$$\begin{array}{ll} z_1:: & z_2:: \\ \quad x:=x+1; x:=x+2, & \quad x:=x+4, \end{array}$$

and assume we want to prove that their parallel composition $z_1 \parallel z_2$ satisfies

$$x:[x=0, x=7]. \tag{4}$$

In the Owicki/Gries method the following recipe is employed:

- Firstly, annotate z_1 and z_2 with assertions a_1, a_2, a_3, a_4 and a_5

  $$\{a_1\}\ x:=x+1;\ \{a_2\}\ x:=x+2\ \{a_3\}, \tag{5}$$
  $$\{a_4\}\ x:=x+4\ \{a_5\}, \tag{6}$$

  such that whenever z_1 is executed in isolation, and the initial state satisfies a_1, then a_2 holds in between the two assignments and a_3 holds after termination. Similarly, whenever z_2 is executed in isolation and the initial state satisfies a_4, then the terminal state satisfies a_5. That the programs are correctly annotated can be proved using standard Hoare-logic.

- Secondly, prove that no atomic step due to z_1 interferes with the annotations of z_2, and that no atomic step due to z_2 interferes with the annotations of z_1. This means that it must be shown that each atomic step in z_1 maintains the annotations in z_2, and similarly, that each atomic step in z_2 maintains the annotations in z_1 — more explicitly, for the first assignment in z_1, prove that

  $$\{a_4 \wedge a_1\}\, x:=x+1\, \{a_4\}, \tag{7}$$
  $$\{a_5 \wedge a_1\}\, x:=x+1\, \{a_5\}. \tag{8}$$

  If 7 holds, it follows that a_4 is maintained, and if 8 holds, it follows that a_5 is maintained. That a_1 in both cases is taken as an additional assumption is ok since 5 requires that a_1 holds until the execution of $x:=x+1$ is started-up. Similarly, for the second assignment in z_1, it must be shown that

  $$\{a_4 \wedge a_2\}\, x:=x+2\, \{a_4\},$$
  $$\{a_5 \wedge a_2\}\, x:=x+2\, \{a_5\},$$

and finally the assignment in z_2 is required to satisfy

$$\{a_1 \wedge a_4\}\, x := x + 4\, \{a_1\},$$
$$\{a_2 \wedge a_4\}\, x := x + 4\, \{a_2\},$$
$$\{a_3 \wedge a_4\}\, x := x + 4\, \{a_3\}.$$

Again, these *freedom of interference proofs* can be carried out using standard Hoare-logic.

- Thirdly, it then follows that $z_1 \parallel z_2$ satisfies

$$x : [a_1 \wedge a_4, a_3 \wedge a_5], \tag{9}$$

Thus, if

$$a_1 \stackrel{\mathsf{def}}{=} x = 0 \vee x = 4,$$
$$a_2 \stackrel{\mathsf{def}}{=} x = 1 \vee x = 5,$$
$$a_3 \stackrel{\mathsf{def}}{=} x = 3 \vee x = 7,$$
$$a_4 \stackrel{\mathsf{def}}{=} x = 0 \vee x = 1 \vee x = 3,$$
$$a_5 \stackrel{\mathsf{def}}{=} x = 4 \vee x = 5 \vee x = 7,$$

it follows that $z_1 \parallel z_2$ satisfies 9, in which case 4 can be deduced using a straightforward consequence rule. For this little example, the freedom of interference proof is trivial, but in the general case such proofs are both time consuming and tedious, because a properly annotated program has one annotation between each atomic program statement.
As indicated in [9], a straightforward way to reformulate the Owicki/Gries method in a compositional style is to extend the pre/post specifications used above with sets of predicates called respectively the rely- and guarantee-conditions. The pre- and post-conditions are given the same interpretation as before, the rely-condition is the set of annotations assumed to be maintained by the environment, and the guarantee-condition is the set of annotations which the component is required to maintain. Thus, these extended specifications are also of an assumption/commitment form — the pre- and rely-conditions make assumptions about the environment, while the guarantee- and post-conditions state commitments to the component. Observe, that the environment is assumed to behave according to the rely-condition both before the first and after the last atomic step due to the component — in other words, not only between the atomic steps.
z_1 and z_2 can now be specified as follows

$$(x = 0, \{a_1, a_2, a_3\}, \{a_4, a_5\}, x = 3 \vee x = 7), \tag{10}$$
$$(x = 0, \{a_4, a_5\}, \{a_1, a_2, a_3\}, x = 4 \vee x = 5 \vee x = 7), \tag{11}$$

while their parallel composition is characterised by

$$(x = 0, \{a_1, a_2, a_3, a_4, a_5\}, \{\,\}, x = 7). \tag{12}$$

Thus, if the overall environment maintains $\{a_1, a_2, a_3, a_4, a_5\}$ then $z_1 \parallel z_2$ will terminate in a state such that $x = 7$. Since the guarantee-condition is the empty set $\{\,\}$, it follows that 12

does not make any claims about the components behaviour during execution, only that $x = 7$ holds after termination whenever the initial state satisfies $x = 0$.
From a practical point of view, it may be argued that a set of annotations can better be represented as a binary predicate. For example instead of assuming that any atomic step by the environment maintains

$$\{x = 0 \vee x = 4, x = 1 \vee x = 5, x = 3 \vee x = 7\},$$

it is also possible to assume that any atomic step due to the environment, which changes the global state, satisfies the binary predicate

$$(\overleftarrow{x} = 0 \vee \overleftarrow{x} = 1 \vee \overleftarrow{x} = 3) \wedge x = \overleftarrow{x} + 4, \tag{13}$$

and similarly for the guarantee-condition. This is basically the position taken in [6] ([6] was published several years before [9]). The following rule

Rule 1

$$\frac{\begin{array}{l} G_1 \vee G_2 \Rightarrow G \\ Q_1 \wedge Q_2 \Rightarrow Q \end{array}}{\vartheta\colon [P, R, G, Q] \rightsquigarrow \vartheta\colon [P, R \vee G_2, G_1, Q_1] \parallel \vartheta\colon [P, R \vee G_1, G_2, Q_2]}$$

then holds. The conclusion states that the parallel composition of the two process specifications is a refinement of the overall specification. The parallel composition of the two process specifications is called a *mixed* specification since it is a mixture of specification and program notation. The set of programs and the set of "pure" specifications can be thought of as subsets of the set of mixed specifications. $\rightsquigarrow$ is a binary relation on mixed specifications such that $S_1 \rightsquigarrow S_2$ iff S_2 is a refinement of S_1 — more particularly, if the set of programs satisfying the mixed specification S_2 is a subset (or equal to) the set of programs which satisfies the mixed specification S_1. A program z satisfies a mixed specification S iff z can be generated from S by substituting a program which satisfies S' for every pure specification S' in S.
The above rule is sound in the following sense: If both premises hold, and z_1 and z_2 are programs such that

$$\vartheta\colon [P, R \vee G_2, G_1, Q_1] \rightsquigarrow z_1, \tag{14}$$
$$\vartheta\colon [P, R \vee G_1, G_2, Q_2] \rightsquigarrow z_2, \tag{15}$$

then

$$\vartheta\colon [P, R, G, Q] \rightsquigarrow z_1 \parallel z_2.$$

To see this, first observe that the rely-condition $R \vee G_2$ of z_1 allows any interference due to z_2 since 15 implies that this interference is characterised by G_2. Similarly, the rely-condition $R \vee G_1$ of z_2 allows any interference due to z_1 since 14 implies that this interference is characterised by G_1. Thus, if the initial state satisfies P, if any interference due to the overall environment satisfies R, it follows that any atomic step due to $z_1 \parallel z_2$ satisfies $G_1 \vee G_2$, and since a post-condition covers interference both before the first atomic step and after the last atomic step due to the component, it follows that the overall effect is characterised by $Q_1 \wedge Q_2$. Thus,

the two premises imply that $z_1 \parallel z_2$ satisfies the overall specification. Observe that to ensure termination it is assumed that a component is never infinitely overtaken by its environment.

3. SYNCHRONISATION

In a parallel program it is often necessary to synchronise the behaviour of the processes. If for example one process is updating a record in a data-base, then we would normally like to make sure that no other process accesses this record until the update has been completed. A well-known synchronisation construct is $\mathsf{await}\ b\ \mathsf{do}\ z\ \mathsf{od}$, which waits until b evaluates to true and then immediately (without allowing the environment to interfere) executes the body z as one atomic step. An await-statement is *disabled* in a state s iff its boolean test evaluates to false in s, and more generally, a program which has not terminated is disabled in a state s iff the program counter of each of its active processes is situated immediately before an await-statement which is disabled in s. A program which remains disabled forever is said to *deadlock*.

In the Owicki/Gries method there is a rule which allows freedom from deadlock to be proved. This rule is global and can be used only after the code is complete. The rule is too complicated to be explained here. However, the relatively simple approach described below can be thought of as a compositional reformulation of the technique proposed by Owicki/Gries.

The idea is to extend specifications with a fifth predicate called the *wait-condition*. The wait-condition is required to characterise the states in which the component is allowed to be disabled. The pre-, rely-, guarantee- and post-conditions have exactly the same interpretation as before, with the exception that the component is no longer required to terminate, but only to terminate whenever it does not deadlock. Specifications are still of an assumption/commitment form: The pre- and rely-conditions make assumptions about the environment, while the wait-, guarantee- and post-conditions should be understood as commitments to the component.

Given a set of variables ϑ, a unary predicate B, and two binary predicates C and D, then I_ϑ denotes the predicate $\bigwedge_{v \in \vartheta} v = \overleftarrow{v}$; $\overleftarrow{B}$ denotes the result of hooking all free variables in B; $C \mid D$ denotes the relational composition of C and D, in other words, (s, s') satisfies $C \mid D$ iff there is a state s'' such that (s, s'') satisfies C and (s'', s') satisfies D; while C^* denotes the reflexive and transitive closure of C. This allows for the following await-rule:

Rule 2

$$\frac{\begin{array}{l}\overleftarrow{P} \wedge R \Rightarrow P \\ P \wedge \neg b \Rightarrow W \\ R^* \mid Q_2 \mid R^* \Rightarrow Q_1\end{array}}{\vartheta\colon [P, R, W, G, Q_1] \rightsquigarrow \mathsf{await}\ b\ \mathsf{do}\ \vartheta\colon [P \wedge b, \mathsf{false}, \mathsf{false}, \mathsf{true}, (G \vee I_\vartheta) \wedge Q_2]\ \mathsf{od}}$$

It is here required that all variables occurring in b are contained in ϑ. The first premise implies that the environment will maintain the pre-condition until the boolean test evaluates to true. Thus, the second premise implies that the await-statement is disabled only in states which satisfy the wait-condition. Q_2 characterises the effect of the await-statement's body. The third premise is needed since interference both before and after the execution of the await-statement's body is included in the overall effect. The environment is constrained from interfering with the await-statement's body, which explains the choice of rely- and wait-conditions in the right-hand side specification. Moreover, the first premise implies that it is

enough to insist that the await-statement's body terminates for any state which satisfies $P \wedge b$. The first conjunct of the post-condition in the right-hand side specification implies that if the component step changes the global state, then this change satisfies the guarantee-condition G. The parallel-rule introduced above must be altered:

Rule 3

$$\frac{\neg(W_1 \wedge Q_2) \wedge \neg(W_2 \wedge Q_1) \wedge \neg(W_1 \wedge W_2) \quad G_1 \vee G_2 \Rightarrow G \quad Q_1 \wedge Q_2 \Rightarrow Q}{\vartheta\colon [P, R, W, G, Q] \rightsquigarrow \vartheta\colon [P, R \vee G_2, W \vee W_1, G_1, Q_1] \parallel \vartheta\colon [P, R \vee G_1, W \vee W_2, G_2, Q_2]}$$

If z_1 and z_2 satisfy respectively the first and the second process specification, it is clear that z_1 can deadlock only in $W \vee W\_1$, when executed in an environment characterised by P and $R \vee G_2$. Moreover, z_2 can deadlock only in $W \vee W_2$, when executed in an environment characterised by P and $R \vee G\_1$. But then, since the first premise implies that z_1 cannot be deadlocked in W_1 after z_2 has terminated, that z_2 cannot be deadlocked in W_2 after z_1 has terminated, and that z_1 (z_2) cannot be deadlocked in $W_1 \wedge \neg W$ ($W_2 \wedge \neg W$) if z_2 (z_1) deadlocks it follows that $z_1 \parallel z_2$ will either terminate or deadlock in W whenever the overall environment is characterised by P and R.

4. REFINEMENT

Up do this point we have mainly been concerned with decomposing specifications into mixed specifications. The object of this section is characterise what it means for a specification to refine another specification. The first rule

Rule 4

$$\frac{P_1 \Rightarrow P_2 \quad R_1 \Rightarrow R_2 \quad W_2 \Rightarrow W_1 \quad G_2 \Rightarrow G_1 \quad Q_2 \Rightarrow Q_1}{\vartheta\colon [P_1, R_1, W_1, G_1, Q_1] \rightsquigarrow \vartheta\colon [P_2, R_2, W_2, G_2, Q_2]}$$

allows the assumptions to be weakened and the commitments to be strengthened. The first two premises imply that the assumptions are weakened, while the latter three imply that the commitments are strengthened. The rule is obviously sound because if z is a program which satisfies the right-hand side specification, then z will also satisfy the left-hand side specification since this has stronger assumptions and weaker commitments.
The different predicates in a specification are of course not independent of each other. The following rule

Rule 5

$$\frac{\overleftarrow{P} \wedge (R \vee G)^* \wedge Q_2 \Rightarrow Q_1}{\vartheta\colon [P, R, W, G, Q_1] \rightsquigarrow \vartheta\colon [P, R, W, G, Q_2]}$$

for example, allows one to weaken the post-condition, while the rule

Rule 6

$$\frac{\begin{array}{l} P \Rightarrow K \\ \overleftarrow{K} \wedge (R \vee G_2) \Rightarrow K \\ K \wedge W_2 \Rightarrow W_1 \\ \overleftarrow{K} \wedge G_2 \Rightarrow G_1 \end{array}}{\vartheta\colon [P, R, W_1, G_1, Q] \rightsquigarrow \vartheta\colon [P, R, W_2, G_2, Q]}$$

allows the wait- and guarantee-conditions to be weakened. The first two premises imply that K is an invariant in the right-hand side specification, in which case it follows that the component can deadlock or perform an atomic step only in a state which satisfies K.

5. SATISFIABILITY

It follows from Rule 4 that

$$\vartheta\colon [\mathsf{true}, \mathsf{false}, \mathsf{false}, \mathsf{true}, \mathsf{false}] \tag{16}$$

is a valid refinement of

$$\vartheta\colon [\mathsf{true}, \mathsf{false}, \mathsf{false}, \mathsf{true}, \mathsf{true}]. \tag{17}$$

However, this is not a sensible refinment step, because while 17 is certainly fulfilled by a large number of programs, there is no program which satisfies 16.
In VDM, to avoid refinements of this type, it is required that whenever the pre-condition holds in a state s, then there is a state s', such that (s, s') satisfies the post-condition. More formally, for any specification

$$\vartheta\colon [P, Q] \tag{18}$$

it is required that

$$\overleftarrow{P} \Rightarrow \exists \vartheta . Q. \tag{19}$$

19 does not guarantee that there is a program which satisfies 18, only that 18 is satisfied by at least one mathematical function. For example, it is straightforward to specify a component which, when given a Turing machine m and tape t as input, terminates in a state such that the boolean variable *term* is true if m halts for input t, and false otherwise. Such a specification certainly satisfies 19. However, there is no implementation since it is well-known that the halting-problem for Turing machines is not decidable. In most cases it is straightforward to prove that 19 holds. To show that there is an implementation would often require the construction of an algorithm, and it is generally accepted that such a requirement is too strong.
19 is of course also a sensible requirement for our more general specifications. For example, 16 does not satisfy 19. Thus, the refinement from 17 to 16 is no longer possible if specifications are required to satisfy 19. However, 19 is not sufficient to exclude that

$$x\colon [x = 0, \mathsf{false}, \mathsf{false}, \mathsf{true}, x > 0] \tag{20}$$

is refined to

$$x\colon [x = 0, \mathsf{false}, \mathsf{false}, \mathsf{false}, x > 0] \tag{21}$$

Clearly, $x := 1$ is a correct implementation of 20, but there is no program which satisfies 21. One way to exclude 21 is to insist, as in [15], that for any specification

$$\vartheta\colon [P, R, W, G, Q]$$

it holds that

$$\overleftarrow{P} \Rightarrow \exists \vartheta.(R \vee G)^* \wedge Q. \tag{22}$$

In other words, whenever the pre-condition holds in a state s, then there is a state s', reachable from s by a finite number of rely/guarantee steps, such that (s, s') satisfies the post-condition. However, even this requirement is quite weak. For example, 22 does not exclude

$$x\colon [x = 0, \mathsf{true}, \mathsf{false}, \mathsf{false}, x = 0]$$

which is obviously not satisfiable. As pointed out in [2], what is needed is a requirement which guarantees that the component always has a winning strategy — in other words, a strategy such that the component can always win no matter what the environment does as long as it satisfies the assumptions. Such a constraint is formulated for the transition axiom method in [2], and we believe a related requirement can be formulated for our specifications. However, such a requirement would be quite complicated and often hard to prove, and it is debatable whether it would be of much practical value.

6. DATA REFINEMENT

Normally, the data representation needed to find an efficient implementation solving a certain problem is quite different from the data representation needed to write an abstract specification characterising the problem. For example, while a database at the abstract level may be represented by a set of records, the concrete representation may be based on some sort of tree structure to allow for efficient searching. This means that it is desirable to have rules which allow the data representation to be changed during the design process. Such rules are often referred to as *data refinement* rules and play an important role in many program design methods. The central idea is quite simple: If $r\colon C \to A$ is a surjective function from the concrete to the abstract state, then the following rule

Rule 7

$$\frac{\begin{array}{l} P_1(r(c)) \Rightarrow P_2(c) \\ R_1(r(\overleftarrow{c}), r(c)) \Rightarrow R_2(\overleftarrow{c}, c) \\ W_2(c) \Rightarrow W_1(r(c)) \\ G_2(\overleftarrow{c}, c) \Rightarrow G_1(r(\overleftarrow{c}), r(c)) \\ Q_2(\overleftarrow{c}, c) \Rightarrow Q_1(r(\overleftarrow{c}), r(c)) \end{array}}{a\colon [P_1(a), R_1(\overleftarrow{a}, a), W_1(a), G_1(\overleftarrow{a}, a), Q_1(\overleftarrow{a}, a)] \rightsquigarrow c\colon [P_2(c), R_2(\overleftarrow{c}, c), W_2(c), G_2(\overleftarrow{c}, c), Q_2(\overleftarrow{c}, c)]}$$

holds. Observe the close relationship to Rule 4. Actually, Rule 4 can be seen as the special case of Rule 7 when r is the identity function on C. The soundness of Rule 7 should be obvious. There are many ways to strengthen this rule. For example, Rule 5 and 6 can be built-in, however, this is no real improvement since these rules can just as well be applied immediately before and/or after a data refinement step. More importantly, however, there may be occasions when we do not want all abstract states in A to be represented by a concrete state in C, or that we would like several states in A to be represented by the same state in C. In that case the above rule is too weak. In [7] it is argued that such situations normally occur only when the abstract specification is not sufficiently abstract. Nevertheless, the above rule can easily be generalised to deal with such situations. For example, a refinement relation can be used instead of a refinement function. Interesting discussions of completeness in connection with data refinement can be found in [4], [1].
Rule 7 is global in the meaning that the global data representation used in the component must be the same as the global data representation used in its environment. Thus, when one decides to split a specification into two subspecification to have them implemented independently at different locations, it is also necessary to make an agreement with respect to the concrete data representation — in other words, decide upon which refinment function to use.

7. TERMINATION

The object of this section is to give a rule for proving conditional termination of while-loops, namely that the component terminates whenever it does not deadlock. A binary predicate B is *well-founded* iff there is no infinite sequence of states $s_1 s_2 \ \ldots \ s_k \ \ldots$, such that for all $j \geq 1$, (s_j, s_{j+1}) satisfies B.

Rule 8

$$\frac{\begin{array}{l} Q_2 \textit{ is well-founded} \\ Q_2^* \wedge \neg b \Rightarrow Q_1 \end{array}}{\vartheta\colon [P, Q_1] \rightsquigarrow \mathsf{while}\ b\ \mathsf{do}\ \vartheta\colon [P \wedge b, Q_2 \wedge P]\ \mathsf{od}}$$

holds in the sequential case. Assume that z is a program which satisfies $\vartheta\colon [P \wedge b, Q_2 \wedge P]$. If P holds initially, it follows that P will hold each time the boolean test is evaluated. Thus, P can thought of as an invariant. Q_2 characterises the effect of the loop's body. This means that the overall effect is characterised by $Q_2^* \wedge \neg b$ if the loop terminates, which explains the second premise. Since the first premise requires Q_2 to be well-founded, it follows that the loop terminates. The rule for the general case is now straightforward:

Rule 9

$$\frac{\overleftarrow{P} \wedge R \Rightarrow P \quad Q_2 \textit{ is well-founded} \quad R^* \mid (Q_2^* \wedge \neg b) \mid R^* \Rightarrow Q_1}{\vartheta\colon [P, R, W, G, Q_1] \rightsquigarrow \mathsf{while}\ b\ \mathsf{do}\ \vartheta\colon [P \wedge b, R, W, G, Q_2 \wedge P]\ \mathsf{od}}$$

The first premise is needed to make sure that the invariant holds when the boolean test is evaluated for the first time. The second premise in Rule 8 corresponds to the third premise in Rule 9. The antecedent of this premise has been changed to cover interference before the first and after the last evaluation of the boolean test. It follows from the specification of the loop's body that any other environment interference is already captured in Q_2.

8. EXPRESSIVENESS

The logic introduced so far is rather incomplete in the meaning that many valid developments are excluded because sufficiently strong predicates cannot be expressed. To explain what this means the Owicki/Gries method will be used once more. Let

$$\begin{array}{ll} z_1\colon: & z_2\colon: \\ \quad x\colon = x+1; x\colon = x+2, & \quad x\colon = x+3, \end{array}$$

and assume we want to prove that $z_1 \parallel z_2$ satisfies

$$x\colon [x = 0, x = 6]. \tag{23}$$

Thus, the only difference with respect to the example in Section 2 is that z_2 increases the value of x with 3 instead of 4. One would therefore expect that it is enough to augment z_1 and z_2 with a_1, a_2, a_3, a_4 and a_5 as in 5 and 6, given that

$$\begin{aligned} a_1 &\stackrel{\mathsf{def}}{=} x = 0 \vee x = 3, \\ a_2 &\stackrel{\mathsf{def}}{=} x = 1 \vee x = 4, \\ a_3 &\stackrel{\mathsf{def}}{=} x = 3 \vee x = 6, \\ a_4 &\stackrel{\mathsf{def}}{=} x = 0 \vee x = 1 \vee x = 3, \\ a_5 &\stackrel{\mathsf{def}}{=} x = 3 \vee x = 4 \vee x = 6. \end{aligned}$$

Unfortunately, this is not the case. The problem is that

$$\begin{aligned} &\{a_4 \wedge a_1\}\, x\colon = x+1\, \{a_4\}, \\ &\{a_1 \wedge a_4\}\, x\colon = x+3\, \{a_1\} \end{aligned}$$

do not hold. For example, if $x = 3$ holds immediately before the execution of $x\colon = x+1$, then the terminal state satisfies $x = 4$. Thus, a_4 is not maintained. In the Owicki/Gries method such problems are dealt with by adding auxiliary program structure. For example, if

$$\begin{array}{l} z_2'\colon: \\ \quad x, d\colon = x+3, \mathsf{true}, \end{array}$$

and a_1, a_2, a_3, a_4 and a_5 are redefined as below

$$a_1 \stackrel{\mathsf{def}}{=} (x = 0 \wedge \neg d) \vee (x = 3 \wedge d),$$
$$a_2 \stackrel{\mathsf{def}}{=} (x = 1 \wedge \neg d) \vee (x = 4 \wedge d),$$
$$a_3 \stackrel{\mathsf{def}}{=} (x = 3 \wedge \neg d) \vee (x = 6 \wedge d),$$
$$a_4 \stackrel{\mathsf{def}}{=} (x = 0 \vee x = 1 \vee x = 3) \wedge \neg d,$$
$$a_5 \stackrel{\mathsf{def}}{=} (x = 3 \vee x = 4 \vee x = 6) \wedge d,$$

using the Owicki/Gries method it can be proved that $z_1 \parallel z_2'$ satisfies

$$x, d\colon [x = 0 \wedge \neg d, x = 6]. \tag{24}$$

Then to deduce that $z_1 \parallel z_2$ satisfies 23, one can use what Owicki/Gries call the *auxiliary variable axiom*[1], which basically states that since the occurrence of d in z_2' does not influence the computation of x in any essential sense, it follows from 24 that $z_1 \parallel z_2$ satisfies

$$x\colon [\exists d.x = 0 \wedge \neg d, x = 6], \tag{25}$$

in which case 23 can be deduced using a straightforward consequence-rule.
In our approach auxiliary assignments are simulated in the deduction rules. To explain this, let us first formulate an assignment-rule for specifications without auxiliary variables:

Rule 10

$$\frac{\begin{array}{l}\overleftarrow{P} \wedge R \Rightarrow P \\ R^* \mid Q_2 \mid R^* \Rightarrow Q_1 \\ \overleftarrow{P} \wedge v = \overleftarrow{r} \wedge I_{\vartheta \setminus \{v\}} \Rightarrow (G \vee I_\vartheta) \wedge Q_2\end{array}}{\vartheta\colon [P, R, \mathsf{false}, G, Q_1] \rightsquigarrow v := r}$$

$\overleftarrow{r}$ denotes the result of hooking all variables in r. An assignment-statement will always terminate, assuming it is not infinitely overtaken by the environment. There is only one atomic step due to the component. However, the environment may interfere both before and after. Since the initial state is assumed to satisfy P, and any atomic step due to the environment, which changes the global state, is assumed to satisfy R, it follows from the first premise that P holds until the atomic assignment is carried out. Thus, it is clear that the assignment step satisfies $\overleftarrow{P} \wedge v = \overleftarrow{r} \wedge I_{\vartheta \setminus \{v\}}$, which according to premise three implies $G \vee I_\vartheta$. This means that the component step either leaves the state unchanged or satisfies G. The second and third premise imply that the overall effect is characterised by Q_1. In the general case, the following rule is needed:

Rule 11

$$\frac{\begin{array}{l}\overleftarrow{P} \wedge R \Rightarrow P \\ R^* \mid Q_2 \mid R^* \Rightarrow Q_1 \\ \overleftarrow{P} \wedge v = \overleftarrow{r} \wedge I_{\vartheta \setminus \{v\}} \wedge a = \overleftarrow{u} \wedge I_{\alpha \setminus \{a\}} \Rightarrow (G \vee I_{\vartheta \cup \alpha}) \wedge Q_2\end{array}}{(\vartheta, \alpha)\colon [P, R, \mathsf{false}, G, Q_1] \rightsquigarrow v := r}$$

[1] To avoid complicating the presentation a slightly modified version is used here.

α is the set of auxiliary variables. The rule simulates that u is assigned to a in the same atomic step as r is assigned to x. To make it possible to remove some auxiliary variables from a specification without having to remove all of them, it is required that a is the only auxiliary variable occurring in u. Moreover, α and ϑ are constrained to be disjoint, and no auxiliary variable is allowed to occur in r.
The following rule can be used to introduce auxiliary variables:

Rule 12

$$\frac{\begin{array}{l} a \notin \vartheta \cup \alpha \\ P_1 \Rightarrow \exists a.P_2 \\ R_1 \Rightarrow \forall \overleftarrow{a}.\exists a.R_2 \end{array}}{(\vartheta, \alpha)\colon [P_1, R_1, W, G, Q] \rightsquigarrow (\vartheta, \alpha \cup \{a\})\colon [P_2, R_2, W, G, Q]}$$

[10] gives a proof of semantic (relative) completeness for a system closely related to the one presented above (without data refinement).

9. EXAMPLE

As an example, assume we want to design a component which, for a given finite, nonempty input sequence i, terminates in a state such that

$$\forall j \in dom(i).o_j = f(g(i_j)).$$

The component is supposed to apply $f \circ g$ to each element of the sequence i and store the result in the sequence o. The component is not allowed to change the value of i. The environment is restricted from updating both i and o.
Now, assume that g and f are functions which can be implemented only using complicated, time-consuming algorithms. One way to speed-up the computation is to split the component into two subcomponents, s_g and s_f, representing g and f, respectively, and execute them in parallel. Three additional variables are needed: An index k, a boolean switch d, and a variable z used by s_g to communicate its results to s_f. Let $\vartheta = \{i, o, k, d, z\}$, then the component can be specified as follows

$$\vartheta\colon [i \neq [], \mathsf{false}, \mathsf{false}, i = \overleftarrow{i}, \forall j \in dom(i).o_j = f(g(i_j))]. \tag{26}$$

k is supposed to point at the last element of i which has been processed by s_g. d is switched on by s_g when a new result has been stored in z. d is switched off by s_f when the result has been read. Let *inv* represent the invariant:

$$k \leq \#i \wedge (d \Rightarrow z = g(i_k)) \wedge (k = \#i \Rightarrow d).$$

$\#i$ denotes the length of i. Thus, the first conjunct states that k is always less than or equal to the length of i. The third conjunct is used to make sure that d remains true after s_g has completed its task. Let *binv* denote the binary invariant:

$$((\neg \overleftarrow{d} \wedge d) \Leftrightarrow k = \overleftarrow{k} + 1) \wedge (k = \overleftarrow{k} \vee k = \overleftarrow{k} + 1) \wedge i = \overleftarrow{i}.$$

The first two conjuncts imply that for any atomic step, k is incremented iff d is switched on. 26 can then be decomposed as $k, d := 0, \mathsf{false}; (s_g \parallel s_f)$, where

$s_g::\ \vartheta:[k = 0 \wedge \neg d \wedge inv,$
$(\neg \overleftarrow{d} \Rightarrow \neg d) \wedge z = \overleftarrow{z} \wedge binv \wedge inv,$
$d \wedge k < \#i,$
$(\overleftarrow{d} \Rightarrow d) \wedge o = \overleftarrow{o} \wedge binv \wedge inv,$
$d]$

$s_f::\ \vartheta:[inv,$
$(\overleftarrow{d} \Rightarrow d) \wedge o = \overleftarrow{o} \wedge binv \wedge inv,$
$\neg d,$
$(\neg \overleftarrow{d} \Rightarrow \neg d) \wedge z = \overleftarrow{z} \wedge binv \wedge inv,$
$\forall j \in dom(i).o_j = f(g(i_j)) \wedge k = \#i]$

s_g and s_f can now be implemented in isolation.

10. FAIR TERMINATION

Let

$z_1::$ $b := \mathsf{true},$ $\quad$ $z_2::$ while $\neg b$ do skip od.

Using the rules introduced above, can it then be shown that $z_1 \parallel z_2$ implements

$$b:[\mathsf{true}, \mathsf{true}, \mathsf{false}, \mathsf{true}, \mathsf{true}], \tag{27}$$

in other words, that $z_1 \parallel z_2$ is guaranteed to terminate in any environment? The answer is of course "no". Such a deduction would be unsound. For example, if b is false initially, the environment consists of the process $b := \mathsf{false}$ only, and this assignment is executed immediately after z_1 terminates, then z_2 will never terminate. Thus, 27 must at least be strengthened as below

$$b:[\mathsf{true}, \overleftarrow{b} \Rightarrow b, \mathsf{false}, \mathsf{true}, \mathsf{true}]. \tag{28}$$

The new rely-condition excludes $b := \mathsf{false}$ as a possible environment. However, $z_1 \parallel z_2$ can still not be deduced as a valid implementation. The reason is that z_1 may be infinitely overtaken by z_2 — in other words, the processors may concentrate on executing z_2 and completely ignore z_1. This means that b may be false initially and remain false forever, in which case z_2 will never terminate.
Problems of this type have led computer scientists to propose that the behaviour of a programming language should satisfy certain fairness constraints. It is usual to distinguish between weak and strong process fairness. Informally, a programming language is

- *weakly fair* iff each process either terminates, performs infinitely many atomic steps or is disabled infinitely often,
- *strongly fair* iff each process either terminates, performs infinitely many atomic steps, or is continuously disabled from a certain point onwards.

In the language considered here only await-statements may become disabled. Thus, weak and strong fairness coincides for programs without await-statements, and it is this restricted case, often referred to as unconditional fairness, that will be discussed here. The technique is described in full detail in [13]. In [14] it is explained how weak and strong fairness can be reasoned about in a similar style.
A programming language, without constructs which may become disabled, is

- *unconditionally fair* iff each process either terminates or performs infinitely many atomic steps.

So-far the wait-condition has been used to characterise the states in which the component is allowed to be disabled — or equivalently, the states in which the component is allowed to wait. We will now use the wait-condition to characterise the states in which the component may end-up *busy-waiting* (forever). For example, z_2 may end-up busy-waiting in $\neg b$. Specifications are written using the same five predicates as before. The only modification is that instead of insisting that the component either remains disabled forever in states which satisfy the wait-condition or terminates, it is now required that the component either ends-up busy-waiting in states which satisfy the wait-condition or terminates. Only Rule 9 has to be changed:

Rule 13

$$\frac{\overleftarrow{P} \wedge R \Rightarrow P \quad Q_2^+ \wedge (R \vee G)^* \mid (I_\vartheta \wedge \neg W) \mid (R \vee G)^* \textit{ is well-founded} \quad R^* \mid (Q_2^* \wedge \neg b) \mid R^* \Rightarrow Q_1}{\vartheta\colon [P, R, W, G, Q_1] \rightsquigarrow \mathsf{while}\ b\ \mathsf{do}\ \vartheta\colon [P \wedge b, R, W, G, Q_2 \wedge P]\ \mathsf{od}}$$

Q_2^+ represents the transitive closure of Q_2. Observe that $\wedge$ is the main symbol of the predicate which is required to be well-founded. To prove that the second premise implies that the statement terminates unless it ends up bust-waiting in W, assume there is a non-terminating computation which satisfies the environment assumptions, but does not end-up busy-waiting in W. It follows from the specification of the loop's body, that each iteration satisfies Q_2. But then, since by assumption $\neg W$ is true infinitely often, and since the overall effect of any finite sequence of environment and component steps satisfies $(R \vee G)^*$, it follows that well-foundness constraint in the second premise does not hold. Thus, the component either ends up busy-waiting in W or terminates.
Now, it is possible to deduce $z_1 \parallel z_2$ from 28. Firstly, 28 can be decomposed into

$$b\colon [\mathsf{true}, \overleftarrow{b} \Rightarrow b, \mathsf{false}, \overleftarrow{b} \Rightarrow b, b] \parallel b\colon [\mathsf{true}, \overleftarrow{b} \Rightarrow b, \neg b, \overleftarrow{b} \Rightarrow b, \mathsf{true}]$$

using Rule 3 and 4. Rule 4 and 10 can be used to deduce that z_1 satisfies the left-hand side specification. skip can be thought of as an alias for the identity assignment, in which case it follows from rule 4 and 10 that skip can be deduced from

$$b\colon [\neg b, \overleftarrow{b} \Rightarrow b, \neg b, \overleftarrow{b} \Rightarrow b, \neg \overleftarrow{b}\ ].$$

But then, since it is clear that

$$\neg\ \overleftarrow{b}\ \wedge (\overleftarrow{b} \Rightarrow b) \mid ((\overleftarrow{b} \Leftrightarrow b) \wedge b) \mid (\overleftarrow{b} \Rightarrow b)$$

is well-founded, it follows from Rule 4 and 13 that z_2 can be deduced from the right-hand side specification.

References

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

[2] M. Abadi and L. Lamport. Composing specifications. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Proc. REX Workshop on Stepwise Refinement of Distributed Systems, Lecture Notes in Computer Science 430*, pages 1–41, 1990.

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

[4] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *Proc. 1st ESOP, Lecture Notes in Computer Science 213*, pages 187–196, 1986.

[5] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[6] 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.

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

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

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

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

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

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

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

[14] K. Stølen. Shared-state design modulo weak and strong process fairness. In M. Diaz, editor, *Proc. FORTE'92*, North-Holland, 1992.

[15] 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. VDM'88, Lecture Notes in Computer Science 328*, pages 434–458, 1988.

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

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

A. FORMAL DEFINITIONS

The object of this appendix is to sketch a formal semantics for the system introduced above. More detailed descriptions can be found in [10], [12] for the unfair case. [11] contains a case-study. [16] presents a related notation. The formalism for the fair language is described in [13]. [14] presents systems for weak and strong fairness.

A.1. UNFAIR CASE

The programming language is given operational semantics in the style of [3]. To avoid complicating the semantics, while-, parallel- and await-statements are restricted from occurring in the body of an await-statement. A *state* is a mapping of all programming variables to values, while a *configuration* is a pair of the form $\langle z, s\rangle$, where z is a program or the *empty program* ϵ, and s is a state. The empty program ϵ models *termination*. s_ϑ denotes the state s restricted to the set of variables ϑ, while $s \models b$ means that the boolean expression b is true in the state s. An *external* transition is the least binary relation on configurations such that

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

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

- $\langle v := r, s\rangle \xrightarrow{i} \langle \epsilon, s\binom{v}{r}\rangle$, where $s\binom{v}{r}$ denotes the state that is obtained from s, by mapping the variables v to the values of r in the state s, and leaving all other maplets unchanged,
- $\langle \mathsf{blo}\ v{:}\,T\ \mathsf{in}\ z\ \mathsf{olb}, s_1\rangle \xrightarrow{i} \langle z, s_2\rangle$, where s_2 denotes a state that is obtained from s_1, by mapping the variables in v to randomly chosen type-correct values, and leaving all other maplets unchanged,
- $\langle z_1; z_2, s_1\rangle \xrightarrow{i} \langle z_2, s_2\rangle$ if $\langle z_1, s_1\rangle \xrightarrow{i} \langle \epsilon, s_2\rangle$,
- $\langle z_1; z_2, s_1\rangle \xrightarrow{i} \langle z_3; z_2, s_2\rangle$ if $\langle z_1, s_1\rangle \xrightarrow{i} \langle z_3, s_2\rangle$ and $z_3 \neq \epsilon$,
- $\langle \mathsf{if}\ b\ \mathsf{then}\ z_1\ \mathsf{else}\ z_2\ \mathsf{fi}, s\rangle \xrightarrow{i} \langle z_1, s\rangle$ if $s \models b$,
- $\langle \mathsf{if}\ b\ \mathsf{then}\ z_1\ \mathsf{else}\ z_2\ \mathsf{fi}, s\rangle \xrightarrow{i} \langle z_2, s\rangle$ if $s \models \neg b$,
- $\langle \mathsf{while}\ b\ \mathsf{do}\ z\ \mathsf{od}, s\rangle \xrightarrow{i} \langle z; \mathsf{while}\ b\ \mathsf{do}\ z\ \mathsf{od}, s\rangle$ if $s \models b$,
- $\langle \mathsf{while}\ b\ \mathsf{do}\ z\ \mathsf{od}, s\rangle \xrightarrow{i} \langle \epsilon, s\rangle$ if $s \models \neg b$,
- $\langle z_1 \parallel z_2, s_1\rangle \xrightarrow{i} \langle z_2, s_2\rangle$ if $\langle z_1, s_1\rangle \xrightarrow{i} \langle \epsilon, s_2\rangle$,
- $\langle z_1 \parallel z_2, s_1\rangle \xrightarrow{i} \langle z_1, s_2\rangle$ if $\langle z_2, s_1\rangle \xrightarrow{i} \langle \epsilon, s_2\rangle$,
- $\langle z_1 \parallel z_2, s_1\rangle \xrightarrow{i} \langle z_3 \parallel z_2, s_2\rangle$ if $\langle z_1, s_1\rangle \xrightarrow{i} \langle z_3, s_2\rangle$ and $z_3 \neq \epsilon$,

- $\langle z_1 \parallel z_2, s_1\rangle \xrightarrow{i} \langle z_1 \parallel z_3, s_2\rangle$ if $\langle z_2, s_1\rangle \xrightarrow{i} \langle z_3, s_2\rangle$ and $z_3 \neq \epsilon$,
- $\langle$await b do z_1 od$, s_1\rangle \xrightarrow{i} \langle \epsilon, s_n\rangle$ if $s_1 \models b$ and there is a sequence of internal transitions such that $\langle z_1, s_1\rangle \xrightarrow{i} \langle z_2, s_2\rangle \xrightarrow{i} \ldots \xrightarrow{i} \langle z_{n\text{-}1}, s_{n\text{-}1}\rangle \xrightarrow{i} \langle \epsilon, s_n\rangle$.

An internal transition represents an atomic step by the component, while an external transition represents an atomic step by the environment. A configuration $\langle z, s\rangle$ is *disabled* iff $z \neq \epsilon$ and there is no configuration $\langle z', s'\rangle$, such that $\langle z, s\rangle \xrightarrow{i} \langle z', s'\rangle$. Moreover, a program z is disabled in the state s iff the configuration $\langle z, s\rangle$ is disabled. A program is *enabled* in a state s if it is not disabled in s. A *computation* is a possibly infinite sequence of external and internal transitions

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

such that the final configuration is disabled if the sequence is finite, and no external transition updates z_1's local variables.
Given a computation σ, $Z(\sigma)$, $S(\sigma)$ and $L(\sigma)$ are the projection functions to sequences of programs, states and transition labels, respectively, and for all $j \geq 1$, $Z(\sigma_j)$, $S(\sigma_j)$, $L(\sigma_j)$ and σ_j denote respectively the j'th program, the j'th state, the j'th transition label and the j'th configuration. $\sigma(j{:}\infty)$ denotes the result of removing the j-1 first transitions, while $\sigma(1{:}j)$ denotes the prefix of σ consisting of the j-1 first transitions. $len\ \sigma$ denotes the number of configurations in σ if σ is finite, and ∞ otherwise.
Two computations (or prefixes of computations) σ of z_1 and σ' of z_2 are *compatible*, if $S(\sigma) = S(\sigma')$ and for all $j \geq 1$, $L(\sigma_j) = L(\sigma'_j)$ implies $L(\sigma_j) = e$. For any pair of compatible computations σ and σ', let $\sigma \bowtie \sigma'$ denote

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

where for all $j \geq 1$: $s_j = S(\sigma_j)$, $z_j = Z(\sigma_j) \parallel Z(\sigma'_j)$ if $Z(\sigma_j) \neq \epsilon$ and $Z(\sigma'_j) \neq \epsilon$, $z_j = Z(\sigma_j)$ if $Z(\sigma'_j) = \epsilon$, $z_j = Z(\sigma'_j)$ if $Z(\sigma_j) = \epsilon$ and $l_j = e$ iff $L(\sigma_j) = e$ and $L(\sigma'_j) = e$.
It is straightforward to show that: For any pair of compatible computations σ of z_1 and σ' of z_2, $\sigma \bowtie \sigma'$ is uniquely determined by the definition above, and $\sigma \bowtie \sigma'$ is a computation of $z_1 \parallel z_2$. Moreover, for any computation σ of $z_1 \parallel z_2$, there are two unique compatible computations σ' of z_1 and σ'' of z_2, such that $\sigma = \sigma' \bowtie \sigma''$.
For a program z, let $cp(z)$ be the set of all computations σ such that $Z(\sigma_1) = z$. For a binary predicate B, $(s_1, s_2) \models B$ iff B evaluates to true when the hooked variables are assigned values in accordance with s_1, and the unhooked variables are assigned values in accordance with s_2.
Given a set of variables ϑ, a pre-condition P and 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 $S(\sigma_j)_\vartheta \neq S(\sigma_{j+1})_\vartheta$ 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 $L(\sigma_k) = i$.

Given a set of variables ϑ, a guarantee-condition G, a wait-condition W and a post-condition Q then $int(\vartheta, G, W, Q)$ 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 $S(\sigma_j)_\vartheta \neq S(\sigma_{j+1})_\vartheta$ then $(S(\sigma_j), S(\sigma_{j+1})) \models G$,
- if $Z(\sigma_{len(\sigma)}) \neq \epsilon$ then $S(\sigma_{len(\sigma)}) \models W$,
- if $Z(\sigma_{len(\sigma)}) = \epsilon$ then $(S(\sigma_1), S(\sigma_{len(\sigma)})) \models Q$.

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

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

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

A program z_1 satisfies a specification $(\vartheta, \alpha)\colon [P, R, W, G, Q]$ iff there is a program z_2, such that $z_1 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_2$ and $ext(\vartheta \cup \alpha, P, R) \cap cp(z_2) \subseteq int(\vartheta \cup \alpha, W, G, Q)$.

A.2. FAIR CASE

In this section only programs without await-statements are considered. Moreover, computations are required to be infinite. To define what it means for a computation to be unconditionally fair, let $<$ be a binary relation on computations such that $\sigma < \sigma'$ iff $S(\sigma) = S(\sigma')$, $L(\sigma) = L(\sigma')$, and there is a (non-empty) program z such that for all $j \geq 1$, $Z(\sigma_j); z = Z(\sigma_j')$. $\sigma \leq \sigma'$ means that $\sigma < \sigma'$ or $\sigma = \sigma'$. Clearly, for any computation σ, there is a minimal computation σ', such that $\sigma' \leq \sigma$ and for all computations σ'', if $\sigma'' < \sigma$ then $\sigma' \leq \sigma''$. Unconditional fairness can then be defined as follows:

- if there is a computation σ', such that $\sigma' < \sigma$ then σ is unconditionally fair iff σ' is unconditionally fair,

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

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

For a program z, let $cp_f(z)$ be the set of all unconditionally fair computations σ such that $Z(\sigma_1) = z$.
Given a set of variables ϑ, a pre-condition P and 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 $j \geq 1$, if $L(\sigma_j) = e$ and $S(\sigma_j)_\vartheta \neq S(\sigma_{j+1})_\vartheta$ then $(S(\sigma_j), S(\sigma_{j+1})) \models R$.

Given a set of variables ϑ, a wait-condition W, a guarantee-condition G and a post-condition Q then $int(\vartheta, W, G, Q)$ denotes the set of all computations σ, such that:

- there is a $j \geq 1$, such that for all $k \geq j$, $S(\sigma_k) \models W$, or there is a $j \geq 1$, such that $Z(\sigma_j) = \epsilon$,

- for all $j \geq 1$, if $L(\sigma_j) = i$ and $S(\sigma_j)_\vartheta \neq S(\sigma_{j+1})_\vartheta$ then $(S(\sigma_j), S(\sigma_{j+1})) \models G$,

- for all $j \geq 1$, if $Z(\sigma_j) = \epsilon$ then $(S(\sigma_1), S(\sigma_j)) \models Q$.

A program z_1 satisfies a specification (ϑ, α): $[P, R, W, G, Q]$ iff there is a program z_2, such that $z_1 \overset{(\vartheta,\alpha)}{\hookrightarrow} z_2$ and $ext(\vartheta \cup \alpha, P, R) \cap cp_f(z_2) \subseteq int(\vartheta \cup \alpha, W, G, Q)$.