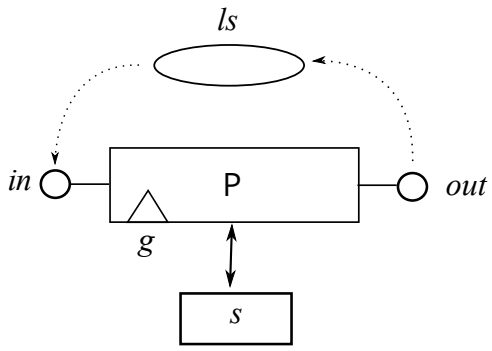

Figure 1: High-level view of a concurrent program

1 Denotations

1.1 Observation Variables

Our program language is built upon basic atomic state-change actions that modify global shared state s. The concurrent flow of control is managed using a global label-set ls and the association of two distinguished labels in and out, and a label generator g with every language construct (See Fig. 1).

A key feature of this UTP semantics is that we have a key distinction between two groups of observations that we wish to make of a running program.

Dynamic State

Both the global shared state and control-flow label-set are observations that change as the program executes. So our UTP theory will need to record both before- and after-values for both of these.

$$\begin{array}{rcll} s, s' & : & State & \text{«obs-s»} \\ ls, ls' & : & \mathcal{P}Lbl & \text{«obs-ls»} \end{array}$$

Static Parameters

The labels and generators associated with any given program construct are fixed during the lifetime of the program. They are determined statically by the context of each construct. For this reason we do not distinguish between before- and after-values

$$\begin{array}{rcll} in, out & : & Lbl & \text{«obs-in-out»} \\ g & : & Gen & \text{«obs-g»} \end{array}$$

So our UTP theory has alphabet

$$\alpha P = \{s, s', ls, ls', in, out, g\} \qquad \text{«alpha-P»}$$

and hence is based on a non-homogeneous relation. This will not turn out to be problematical because the formulation of the theory as described below will largely isolate the two classes of observations.

1.1.1 Ground Expressions

We assume a general notion of expressions (e) and say that an expression e is "ground" if its free variables are limited to only g, in and out. We say that a predicate is ground if all constituent expressions are ground.

$$\begin{array}{rcll} isGnd(e) & \widehat{=} & FV(e) \subseteq \{g, in, out\} & \text{«·gnd-expr-def·»} \\ isGnd(P) & \widehat{=} & FV(P) \subseteq \{g, in, out\} & \text{«·gnd-pred-def·»} \end{array}$$

This means that ground expressions in this setting can only define either generators, label-sets, or atomic predicates over these. Indeed, generator expressions are ground by construction, because the only variable they contain is precisely g. It also means that a ground expression can always be simplified to either a generator expression, enumerated label-set, and any ground predicate to either true (valid) or false (a contradiction).

1.2 "Standard" UTP Predicates

By "standard" UTP predicates, we mean sequential composition ($\_ \,;\, \_$), skip ($I\!I$), conditionals $\_ \lhd \_ \rhd \_$, and iteration ($\_ * \_$). We put "standard" in quotes because we are defining these in the context of our non-homogeneous alphabet, which is itself non-standard.

Those familiar with UTP may wonder why we distinguish between sequential composition in the language (;;) and UTP sequential composition (;). In UTP theories of sequential programming, and reactive systems like CSP, sequential composition in the language has relational predicate composition as its semantics, and so the same symbol is used for both. In concurrent shared-variable programs, there is no such coincidence between the language operator, and its semantics. The precise interpretation of UTP sequential composition (;) in this theory will be explained when we introduce the actual denotational semantics in Section 1.4.

Given a non-homogeneous alphabet, some care has to be taken to define standard notions from UTP theories, such as sequential composition (;), and its unit predicate Skip $I\!I$. The important thing here turns out to be that these concepts completely ignore the static parameters. So the definition of standard sequential composition defines a mid-point state based solely on s and ls:

$$\begin{array}{rcll} P \,;\, Q & \widehat{=} & \exists s_m, ls_m \bullet & \\ & & \quad P[s_m, ls_m / s', ls'] \wedge Q[s_m, ls_m / s, ls] & \text{«·UTP-seq-def·»} \end{array}$$

Similarly, $I\!I$, the behaviour that immediately terminates without changing any state, only refers to the dynamic observations:

$$I\!I \quad \widehat{=} \quad s' = s \wedge ls' = ls \quad \text{«·UTP-skip-def·»}$$

It is easy to show that $I\!I$ is both a left- and right-unit for ;, that ; is associative, and that disjunction distributes through it:

$$\begin{array}{rcccll} I\!I\ ;\ P & = & P & = & P\ ;\ I\!I & \text{«·seq-unit·»} \\ P\ ;\ (Q\ ;\ R) & & = & & (P\ ;\ Q)\ ;\ R & \text{«·seq-assoc·»} \\ P\ ;\ (Q \vee R) & & = & & P\ ;\ Q \vee P\ ;\ R & \text{«·seq-or-distr·»} \\ (P \vee Q)\ ;\ R & & = & & P\ ;\ R \vee Q\ ;\ R & \text{«·or-seq-distr·»} \end{array}$$

Note that we assume that ; binds more tightly than $\vee$. We also note the following law, an easy consequence of the above laws:

$$(I\!I \vee P)\ ;\ (I\!I \vee Q) \quad = \quad I\!I \vee P \vee Q \vee P\ ;\ Q \qquad \text{«·skip-lor-twice·»}$$

It, and its generalisation will play an important role in what is to come. Also important is that ground predicates in conjunction with a sequential composition can be pushed inside:

$$\begin{array}{rcll} K \wedge (P\ ;\ Q) & = & K \wedge P\ ;\ Q & \text{«·GND-and-seq-L·»} \\ & = & P\ ;\ K \wedge Q & \text{«·GND-and-seq-R·»} \\ & = & K \wedge P\ ;\ K \wedge Q & \text{«·GND-and-seq-both·»} \\ K\ ;\ K & = & K & \text{«·GND-seq-GND-is-GND·»} \end{array}$$

where $isGnd(K)$, and $\wedge$ binds tighter than ;.

Given the definitions above, then the definitions of UTP conditionals and iteration are quite standard.

$$\begin{array}{rcll} P \lhd C \rhd Q & \mathrel{\widehat{=}} & C \wedge P \vee \neg C \wedge Q & \text{«·UTP-cond-def·»} \\ C * P & = & P; C * P \lhd C \rhd I\!I & \text{«·UTP-loop-unroll·»} \end{array}$$

For iteration we just present the loop unrolling law, as the most useful here.

We also define a specialised form of sequential composition to be used when neither component refers to ls or ls', and its unit ii:

$$\begin{array}{rcccll} P\ ;_s\ Q & \mathrel{\widehat{=}} & \exists s_m \bullet P[s_m/s'] \wedge Q[s_m/s] & & & \text{«·UTP-s-seq-def·»} \\ ii & \mathrel{\widehat{=}} & s' = s & & & \text{«·ii-def·»} \\ ii\ ;_s\ a & = & a & = & a\ ;_s\ ii & \text{«·seq-s-unit·»} \end{array}$$

Note that if neither predicate mentions ls or ls' then the effect of ; and $;_s$ is the same. We often omit the s subscript when its use is clear from context.

1.2.1 Ground Substitutions

Substitution of expressions for free variables is an important mechanism used to construct and reason about UTP theories. We define a substitution as being ground if the expressions are all ground, and the target variables belong to g, in and out.

$$isGnd[G, I, O/g, in, out] \quad \mathrel{\widehat{=}} \quad isGnd(G) \wedge isGnd(I) \wedge isGnd(O) \qquad \text{«·gnd-subs-def·»}$$

where G is a generator expression, and I and O are label-set expressions. In the sequel we shall always assume that G, I and O stand for ground expressions, as here.

We shall use γ to denote a ground substitution.

$$\gamma \quad = \quad [G, I, O/g, in, out] \quad \text{«·gamma-def·»}$$

The identity substitution can be written as a ground substitution:

$$\gamma_{id} \quad = \quad [g, in, out/g, in, out] \quad \text{«·gamma-id-def·»}$$

The significance of ground substitutions is that they ignore UTP sequential composition, in that they distribute through them, and have no effect on Skip

$$\begin{array}{rcll} (P \,;\, Q)\gamma & = & P\gamma \,;\, Q\gamma & \text{«·seq-gnd-distr·»} \\ I\!I\gamma & = & I\!I & \text{«·skip-gamma·»} \end{array}$$

They also distribute nicely through our invariant shorthands

$$\begin{array}{rcll} \{L_1|\ldots|L_n\}\gamma & = & \{L_1\gamma|\ldots|L_n\gamma\} & \text{«·DL-gamma-subst·»} \\ [L_1|\ldots|L_n]\gamma & = & [L_1\gamma|\ldots|L_n\gamma] & \text{«·LE-gamma-subst·»} \end{array}$$

In addition, the set of all ground substitutions is closed under substitution composition.

$$\exists G, I, O \quad \bullet \quad [G_1, I_1, O_1/g, in, out][G_2, I_2, O_2/g, in, out] = [G, I, O/g, in, out] \quad \text{«·gnd-sub-closure·»}$$

We can even give a direct definition of the result:

$$[G_1, I_1, O_1/g, in, out]\gamma_2 \quad = \quad [G_1\gamma_2, I_1\gamma_2, O_1\gamma_2/g, in, out] \quad \text{«·gnd-sub-comp·»}$$

1.2.2 Sound Substitutions

Unfortunately, there are ground substitutions that can break the unique label properties we have worked so hard to achieve with our label generators. A good example of this is $[g, \ell_g, \ell_g/g, in, out]$, which can transform an invariant respecting predicate P into one that violates that invariant in a strong way.

A ground substitution $[G, I, O/g, in, out]$ is sound if the disjoint label invariant $\{I|G|O\}$ is true.

$$isSound[G, I, O/g, in, out] \quad \widehat{=} \quad \{I|G|O\} \quad \text{«·sound-sub-def·»}$$

We shall use ς to denote a sound substitution

$$\varsigma \quad = \quad [G, I, O/g, in, out] \textbf{ where } \{I|G|O\} \quad \text{«·vsigma-def·»}$$

Note that γ_{id} is sound.

All the substitutions explicitly given in the semantic rules below are sound. So it is particularly important that soundness is preserved by substitution composition

$$isSound(\varsigma_1) \land isSound(\varsigma_2) \quad \Longrightarrow \quad isSound(\varsigma_1\varsigma_2) \quad \text{«·sound-sub-closure·»}$$

1.3 Healthiness Conditions

As is normal in a UTP theory, we define a number of healthiness conditions that characterise the predicates we consider to be sound assertions with respect to our intended interpretation. At present we have two, called Disjoint Labels and Wheels-within-Wheels respectively.

1.3.1 Disjoint Labels

We have a general healthiness condition (Disjoint Labels) which asserts that the labels associated with *in*, *out* and g are different:

$$\mathbf{DL}(P) \quad = \quad P \land \{in|g|out\} \quad \text{«·DL-def·»}$$

This is just the Disjoint Labels invariant of §**??**, reformulated as a healthiness condition. Also, it is a ground predicate, and so distributes through sequential composition:

$$\mathbf{DL}(P\,;Q) \quad = \quad \mathbf{DL}(P)\,;\mathbf{DL}(Q) \quad \text{«·DL-seq-distr, p??·»}$$

1.3.2 Wheels-within-Wheels

The key intuition behind this compositional semantics was to take the *run* function of the action-system based semantic model used in UTPP, and drive it inwards to every level of the program. The original *run* can be defined in the context of this theory as

$$run(P) = ls := \{in\}; \neg(out \in ls) * P$$

However this failed to keep atomic components "live". They could never be re-executed, as would be required if they were within an iteration. Instead it was realised that every construct (atomic and composite) would have to be within an infinite loop.

$$\mathbf{true} * P \quad \text{«·W-as-loop·»}$$

This wrapping of the basic property in an infinite loop would occur at every level of the program hierarchy, hence the term "Wheels within Wheels".

This bold step turns out to be remarkably effective, with some quite counter-intuitive outcomes. However it does depend on a specific tweak to the definition of an atomic action. In effect we define an atomic action as placing a basic action inside such a loop, but within a non-deterministic choice between it and a *stuttering* step, denoted by UTP skip:

$$\langle a \rangle \quad \widehat{=} \quad \mathbf{true} * (II \lor A(in|a|out))$$

A result of this is that this stuttering step gets propagated up to enclosing composites, so in effect we see $\mathbf{true} * C = \mathbf{true} * (II \lor C)$ where C is any predicate denoting the semantics of a command. In effect, we see that

$$\mathbf{true} * (II \lor P) \quad = \quad \bigvee_{i \in \mathbb{N}} P^i \quad \text{«·loop-as-NDC·»}$$

This leads us to define the healthiness condition as

$$\begin{array}{rcll} P^0 & \widehat{=} & II & \text{«·seq-0·»} \\ P^{i+1} & \widehat{=} & P \,;\, P^i & \text{«·seq-i-plus-1·»} \\ \mathbf{W}(P) & \widehat{=} & \bigvee_{i \in \mathbb{N}} P^i & \text{«·W-as-NDC·»} \end{array}$$

We note explicitly here that, in effect, our semantic model is based on unbounded non-determinism.

For $\mathbf{W}$ to be a proper healthiness condition is has to be monotonic and idempotent.

$$\begin{array}{rcll} P \sqsubseteq Q & \Longrightarrow & \mathbf{W}(P) \sqsubseteq \mathbf{W}(Q) & \text{«·W-monotonic·»} \\ \mathbf{W}(\mathbf{W}(P)) & = & \mathbf{W}(P) & \text{«·W-idempotent·»} \end{array}$$

The latter is easy to show once we realise that composing the result of $\mathbf{W}(P)$ with itself results in no change:

$$\mathbf{W}(P) \,;\, \mathbf{W}(P) \quad = \quad \mathbf{W}(P) \quad \text{«·W-seq-W-is-W·»}$$

Given the definition of $\mathbf{W}$, it is easy to see that ground substitution distributes in:

$$\mathbf{W}(P)\gamma \quad = \quad \mathbf{W}(P\gamma) \quad \text{«·W-gnd-subst·»}$$

as does any top-level conjunction with a ground predicate K:

$$K \land \mathbf{W}(Q) \quad = \quad \mathbf{W}(K \land Q) \quad \text{«·W-gnd-and-distr·»}$$

For iteration-free programs we find that there is a finite k such that $P^k = \mathbf{false}$, in which case the non-determinism is bounded. We get these finite results that seem very similar to the results obtained by using *run* above. Using *run* results in predicates that cannot be composed to get composite behaviour. However, using $\mathbf{W}$ results in a slight variation, which is composable! What turns out to be crucial to this outcome is the explicit stuttering option.

1.4 UTP Denotational Semantics

The denotational semantic definitions below will have $\mathbf{W}(P)$ as a component, where P will turn out to be a disjunction of lifted atomic actions, one of which will be $I\!I$.

$$P = I\!I \vee A_1 \vee A_2 \vee A_3 \vee \ldots$$

The effect of this is that $\mathbf{W}(P)$ itself will be a disjunction of elements that are either $I\!I$, one of those actions making up P or the sequential composition, using ;, of a sequence of two or more of those actions.

$$\begin{array}{lcl} \mathbf{W}(P) & = & \bigvee_{i\in\mathbb{N}} P^i \\ & = & I\!I \vee A_1 \vee A_2 \vee A_3 \vee \ldots \\ & \vee & A_1^2 \vee A_2^2 \vee A_3^2 \vee \ldots \\ & \vee & A_1 \,;\, A_2 \vee A_2 \,;\, A_1 \vee \ldots \\ & \vee & \cdots \vee A_3 \,;\, A_3 \,;\, A_1 \,;\, A_2 \vee \ldots \end{array}$$

In practise, many of these sequential compositions will reduce to **false**, because they are not viable—this is discussed in more detail in the section on calculating with the theory (§**??**).

Given a composition $A_3 \,;\, A_3 \,;\, A_1 \,;\, A_2$ (say), it is to be interpreted as an observation of the execution of its components in order, without any interference from outside. In effect these terms correspond to mumblings in the denotational semantics of Brookes[**?**]. So, in this theory, ; can be thought of as *mumbling composition*. In a similar vein, the $I\!I$ predicate represents all the possible stuttering steps in Brookes' semantics.

To proceed we shall first introduce the notation of a Basic Action, which is the lifting of an atomic shared state transformer a into something that waits for itself to be enabled by appropriate labels. Then we give the semantics of each language construct in turn.

1.4.1 Basic Action

Basic Action $A(E|a|N)$ is enabled when all the labels in E are present in the global label-set and atomic action a does not evaluate to **false** in the current program state. If so enabled, it performs action a, removes the labels in E from the label-set, and adds in those in N.

$$A(E|a|N) \quad \widehat{=} \quad E \subseteq ls \wedge a \wedge ls' = (ls \setminus E) \cup N \quad \text{«·A-def·»}$$

This is a basic building block that is used in subsequent definitions, both for lifting atomic state change actions, and for defining control-flow label management.

A key property of basic actions is that ground substitutions can be driven inside

$$A(E|a|N)\gamma \quad = \quad A(E\gamma|a|N\gamma) \quad \text{«·A-gamma-subs·»}$$

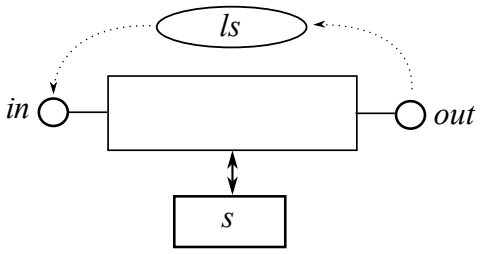

Figure 2: Atomic Action

1.4.2 Atomic Action

An atomic action (Fig.2) is a basic action where the enabling label-set is simply $\{in\}$, and the new label introduced once complete is $\{out\}$. It has no subcomponents, so it does not use the label generator g.

The key invariant here is that in and out are never simultaneously in ls. So we simply wrap the corresponding action with the healthiness conditions and conjoin our invariant:

$$\langle a \rangle \quad \widehat{=} \quad [in|out] \wedge \mathbf{DL}(\mathbf{W}(A(in|a|out))) \quad \text{«·sem:atomic·»}$$

Note here that we write x rather than $\{x\}$, as a shorthand. This is why the notation for A uses vertical bars to separate arguments.

We need to show that $A(in|a|out)$ preserves the specified invariant, under any arbitrary sound substitution ς:

$$\{in|out\}\varsigma \wedge [in|out]\varsigma \wedge A(in|a|out)\varsigma \quad \Longrightarrow \quad [in|out]'\varsigma \quad \text{«·atom-inv-ok, p??·»}$$

Here $[in|out]'$ is simply $[in|out]$ with ls replaced by ls'.

1.4.3 Guarded Atomic Action

In effect there is no real difference between $c\&a$ and $\mathbf{true}\&(c \wedge a)$, so in fact we don't need guarded actions as basic. This is an advantage of treating the atomic action as a relational predicate on state.

$$c\&a \quad \widehat{=} \quad \langle c \wedge a \rangle \quad \text{«·sem:pgrd·»}$$

1.4.4 Skip

Skip in the programming language (`skip`) is simply an atomic action that leaves the program state s unchanged:

$$\begin{array}{rcll} \texttt{skip} & \widehat{=} & \langle s' = s \rangle & \text{«·sem:skip·»} \\ & = & \langle ii \rangle & \end{array}$$

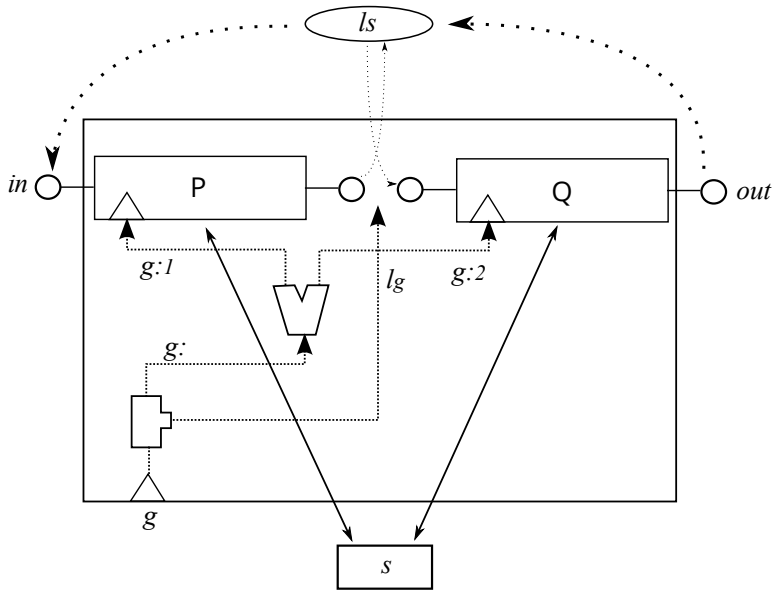

Figure 3: Sequential Composition

1.4.5 Sequential Composition

In order to sequentially compose P and Q (Fig.3), we want to arrange for the *out* label of P to coincide with the *in* label of Q. We do this by generating a new unique label with our generator g, giving us label ℓ_g and the "leftover" generator $g_{:}$. That ℓ_g is not the same as either *in* or *out* is guaranteed by the **DL** invariant. When then replace all references in P to *out*, and in Q to *in*, by ℓ_g. We also need to ensure that the generators used by P and Q do not produce labels that clash with *in*, *out*, ℓ_g, or each others' generator. We do this by splitting our leftover generator into two parts, $g_{:1}$ and $g_{:2}$, and replacing all references to g in both P and Q with these split generators.

So P becomes $P[g_{:1}, \ell_g/g, out]$, and Q is changed to $Q[g_{:2}, \ell_g/g, in]$, both substitutions being sound. Next, we connect our modified P and Q with *disjunction*, and healthify. For all of this to work we require the LE invariant to ensure that only one of *in*, ℓ_g or *out* are ever present in ls.

$$P \;;; Q \quad \widehat{=} \quad [in|\ell_g|out] \wedge \mathbf{DL}(\mathbf{W}(P[g_{:1}, \ell_g/g, out] \vee Q[g_{:2}, \ell_g/g, in])) \quad \text{«·sem:seq·»}$$

If A actions inside P and Q preserve the invariant, then so does their composition. This is immediate by the sound substitution independence of invariant preservation (c.f., «·atom-inv-ok, p??·»).

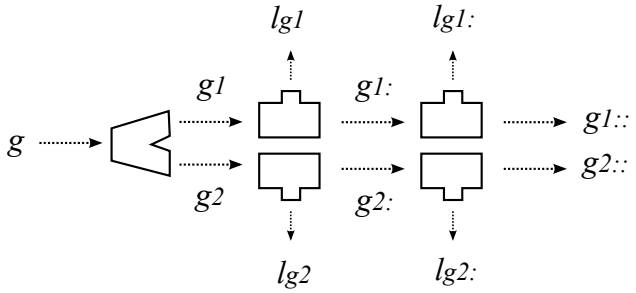

Figure 4: Generating labels of Parallel Composition

1.4.6 Parallel Composition

Parallel composition $P \parallel Q$ may seem straightforward: simply leave *in* and *out* alone, split g and feed the two generators into P and Q, put into disjunction with each other. However this won't work. Both P and Q will have an atomic action enabled by *in*, and one of these will be non-deterministically chosen to run first. When it does it will remove *in* from ls, effectively disabling the other action. Instead we need to replace the *in*s and *out*s of P and Q with four different labels, using a scheme such as that described in Fig.4.

We then introduce two new control-flow actions: the first (Split) is enabled by *in*, and adds into ls the labels replacing the *in*s; the second (Merge) is enabled by both of the labels replacing the *out*s, and adds *out* into ls. In effect, Split enforces that both P and Q start together, once $P \parallel Q$ has started, while Merge ensures that $P \parallel Q$ only terminates when both P and Q have (See Fig.5).

The LE invariant for parallel has not only to ensure that *in* and *out* are never in ls together or when any of the four generated labels are, but also that the labels that replace *in* are never in ls at the same time as the corresponding replacements for *out* (see also §**??**).

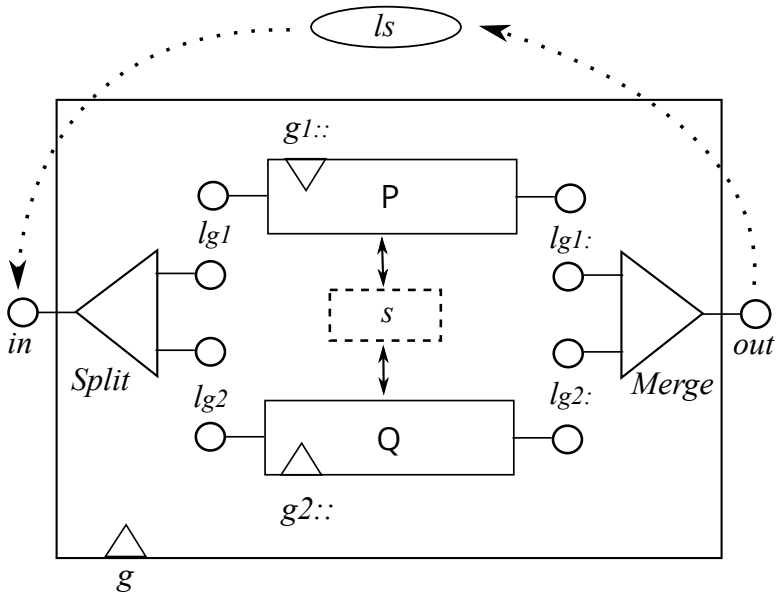

Figure 5: Parallel Composition

Putting all of this together gives the following definition

$$
\begin{array}{lcll}
P \parallel Q & \widehat{=} & [in|(\ell_{g1}|\ell_{g1:}),(\ell_{g2}|\ell_{g2:})|out] \wedge & \text{«·sem:par·»} \\
 & & \mathbf{DL}(\mathbf{W}(\quad A(in|ii|\ell_{g1},\ell_{g2}) & \\
 & & \qquad \vee P[g_{1::},\ell_{g1},\ell_{g1:}/g,in,out] & \\
 & & \qquad \vee Q[g_{2::},\ell_{g2},\ell_{g2:}/g,in,out] & \\
 & & \qquad \vee A(\ell_{g1:},\ell_{g2:}|ii|out)\ )) &
\end{array}
$$

We need to demonstrate that the Split and Merge actions preserve the LE invariant:

$$
\begin{array}{rcll}
\{in|g|out\}\varsigma \wedge [in|(\ell_{g1}|\ell_{g1:}),(\ell_{g2}|\ell_{g2:})|out]\varsigma & & & \text{«·split-inv-ok, p??·»} \\
\wedge\, A(in|ii|\ell_{g1},\ell_{g2})\varsigma & \Longrightarrow & [in|(\ell_{g1}|\ell_{g1:}),(\ell_{g2}|\ell_{g2:})|out]'\varsigma & \\
\{in|g|out\}\varsigma \wedge [in|(\ell_{g1}|\ell_{g1:}),(\ell_{g2}|\ell_{g2:})|out]\varsigma & & & \text{«·merge-inv-ok, p??·»} \\
\wedge\, A(\ell_{g1:},\ell_{g2:}|ii|out)\varsigma & \Longrightarrow & [in|(\ell_{g1}|\ell_{g1:}),(\ell_{g2}|\ell_{g2:})|out]'\varsigma &
\end{array}
$$

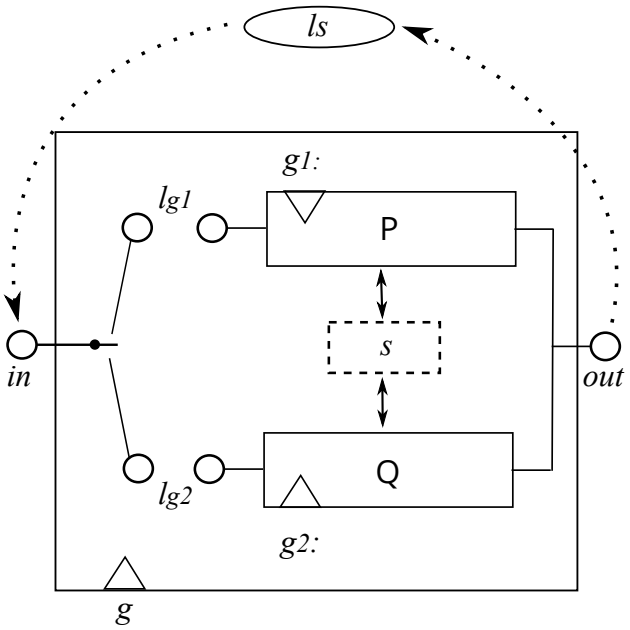

Figure 6: Nondeterministic Choice

1.4.7 Nondeterministic Choice

Nondeterministic choice $P + Q$ is very simple: we simply split the generator and pass one part into each component. We do not need to relabel *in* or *out*. The first atomic action enabled by *in* to run in either component will disable all other such actions in both components, because it will remove *in* from the label-set.

WE NEED TO CHECK - WHAT ABOUT in **to** in **mumblings, as per iteration???? Maybe the old def is the right one!**

$$\begin{array}{lcll} P+Q & \widehat{=} & [in|\ell_{g1}|\ell_{g2}|out] \wedge & \text{«·sem:NDC·»} \\ & & \mathbf{DL}(\mathbf{W}(\quad A(in|ii|\ell_{g1}) & \\ & & \quad \vee A(in|ii|\ell_{g2}) & \\ & & \quad \vee P[g_{1:}, \ell_{g1}/g, in] & \\ & & \quad \vee Q[g_{2:}, \ell_{g2}/g, in]\ )) & \end{array}$$

LE invariant preservation - not required

HMMMMM! PERHAPS NDC is just $P[g_1/g] \vee P[g_2/g]$???

$$P + Q \quad \widehat{=} \quad [in|out] \wedge \mathbf{DL}(\mathbf{W}(P[g_1/g] \vee Q[g_2/g]))$$

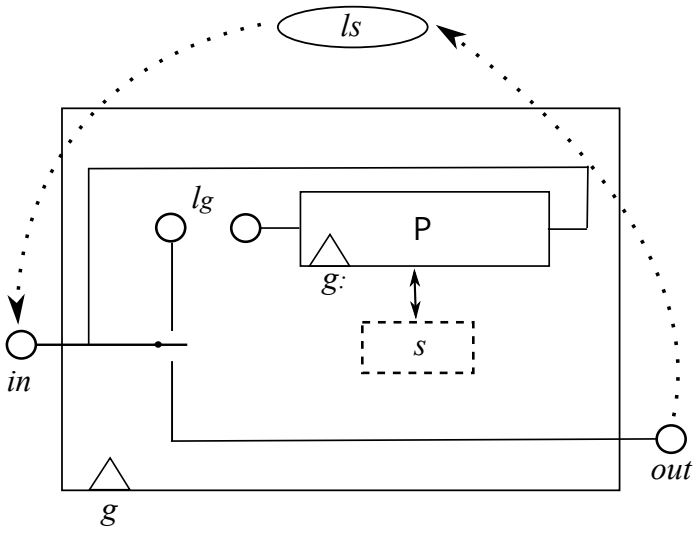

Figure 7: Nondeterministic Loop

1.4.8 Nondeterministic Loop

A nondeterministic loop P^* starts with a non-deterministic choice regarding if it will immediately terminate, or perform an iteration. In the former case we go directly from *in* to *out*, while in the latter we need a new label (ℓ_g) which replaces the *in* of P. When P terminates, we immediately go back to the top-level *in* to choose again.

$$\begin{array}{lcll} P^* & \widehat{=} & [in|\ell_g|out] \wedge & \text{«·sem:star·»} \\ & & \mathbf{DL}(\mathbf{W}(\quad A(in|ii|out) & \\ & & \qquad \vee A(in|ii|\ell_g) & \\ & & \qquad \vee P[g_:, \ell_g, in/g, in, out] \ )) & \end{array}$$

Note that the slightly strange substitution $[g_:, \ell_g, in/g, in, out]$ is in fact sound, as $\{g_:|\ell_g|in\}$ is true, given **DL**.

LE invariant preservation

$$\begin{array}{rcll} \{in|g|out\}\varsigma \wedge [in|\ell_{g1}|out]\varsigma & & & \text{«·ndc-exit-inv-ok, p??·»} \\ \wedge A(in|ii|out)\varsigma & \Longrightarrow & [in|\ell_{g1}|out]'\varsigma & \\ \{in|g|out\}\varsigma \wedge [in|\ell_{g1}|out]\varsigma & & & \text{«·ndc-loop-inv-ok, p??·»} \\ \wedge A(in|ii|\ell_{g1})\varsigma & \Longrightarrow & [in|\ell_{g1}|out]'\varsigma & \end{array}$$

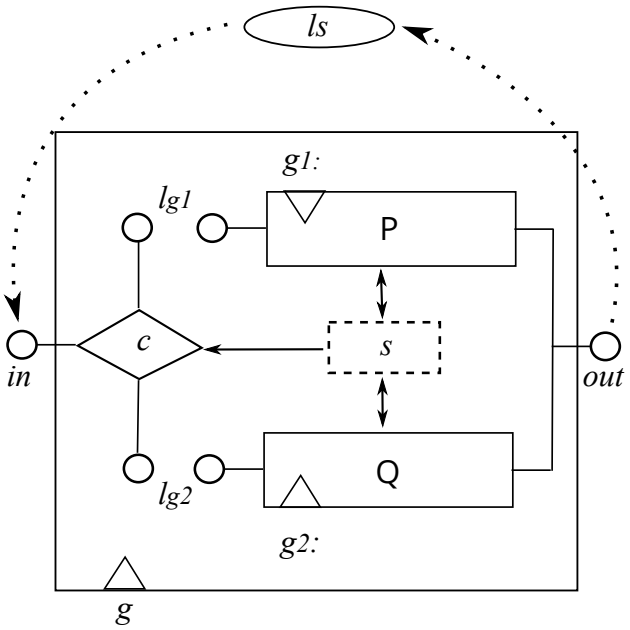

Figure 8: Conditional Choice

1.4.9 Conditional Choice

Brought here *Nondeterministic choice $P+Q$ is fairly simple. Provide two new labels to replace the ins of P and Q and introduce two independent control-flow actions, both enabled by in, each of which adds one of the new labels to ls (while removing in, of course). Then the first control-flow action to run will enable its component, while simultaneously disabling the other control-flow action. Split generators are passed into P and Q as usual. The LE invariant has to assert that only one of two new labels can be present in ls at any point in time.*

The semantics for conditional choice $P \blacktriangleleft c \blacktriangleright Q$ is very similar to that for nondeterministic choice. The only difference is that the two control-flow actions have the condition c, or its negation as appropriate, conjoined with the underlying program state skip action ii. This ensures that the control-flow action is only able to proceed if in is in ls and the condition or its negation is true in the current state s.

$$
\begin{array}{lcll}
P \blacktriangleleft c \blacktriangleright Q & \widehat{=} & [in|\ell_{g1}|\ell_{g2}|out] \wedge & \text{«·sem:cond·»} \\
 & & \mathbf{DL}(\mathbf{W}(\quad A(in|c \wedge ii|\ell_{g1}) & \\
 & & \qquad \vee A(in|\neg c \wedge ii|\ell_{g2}) & \\
 & & \qquad \vee P[g_{1:}, \ell_{g1}/g, in] & \\
 & & \qquad \vee Q[g_{2:}, \ell_{g2}/g, in]\ )) &
\end{array}
$$

LE invariant preservation

$$
\begin{array}{rcl}
\{in|g|out\}\varsigma \wedge [in|\ell_{g1}|\ell_{g2}|out]\varsigma & & \text{«·cond-1-inv-ok, p??·»} \\
\wedge\, A(in|c \wedge ii|\ell_{g1})\varsigma & \Longrightarrow & [in|\ell_{g1}|\ell_{g2}|out]'\varsigma \\
\{in|g|out\}\varsigma \wedge [in|\ell_{g1}|\ell_{g2}|out]\varsigma & & \text{«·cond-2-inv-ok, p??·»} \\
\wedge\, A(in|\neg c \wedge ii|\ell_{g2})\varsigma & \Longrightarrow & [in|\ell_{g1}|\ell_{g2}|out]'\varsigma
\end{array}
$$

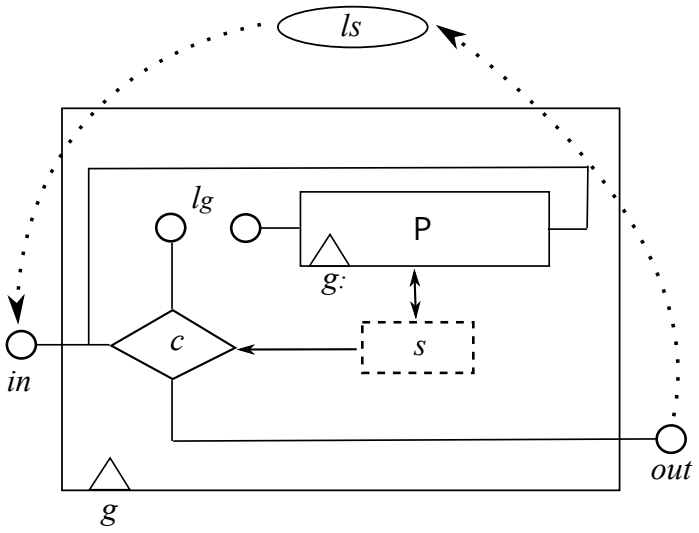

Figure 9: Conditional Loop

1.4.10 Conditional Loop

See the previous section about conditional choice regarding the role of state-condition c—the same idea applies here too.

$$
\begin{array}{lcll}
c \circledast P & \widehat{=} & [in|\ell_g|out] \wedge & \text{«·sem:iter·»} \\
 & & \mathbf{DL}(\mathbf{W}(\quad A(in|\neg c \wedge ii|out) & \\
 & & \qquad \vee A(in|c \wedge ii|\ell_g) & \\
 & & \qquad \vee P[g_:, \ell_g, in/g, in, out] \ )) &
\end{array}
$$

LE invariant preservation

$$
\begin{array}{rcll}
\{in|g|out\}\varsigma \wedge [in|\ell_{g1}|out]\varsigma & & & \text{«·cond-exit-inv-ok, p??·»} \\
\wedge A(in|\neg c \wedge ii|out)\varsigma & \Longrightarrow & [in|\ell_{g1}|out]'\varsigma & \\
\{in|g|out\}\varsigma \wedge [in|\ell_{g1}|out]\varsigma & & & \text{«·cond-loop-inv-ok, p??·»} \\
\wedge A(in|c \wedge ii|\ell_{g1})\varsigma & \Longrightarrow & [in|\ell_{g1}|out]'\varsigma &
\end{array}
$$

1.4.11 Key Semantic Properties

Label Movement We need to show that in any construct, no basic action is enabled by *out* and that it is only added into *ls*. This property holds under any sound substitution ς as well. We can't say a complementary thing about *in* because of the definition of iteration, where *in* replaces *out* in the loop body.

1.5 Semantics Summary

A Label Proofs

A.1 Proof of labs-fully-disjoint

$$\{\ell_G\} \quad labs(G_:) \quad labs(G_1) \quad labs(G_2) \qquad \text{«·labs-fully-disjoint, p??·»}$$

TBD

A.2 Proof of prefix-lab-subset

$$labs(g_\rho) \subseteq labs(g_\varrho) \quad \equiv \quad \varrho \leq \rho \qquad \text{«·prefix-lab-subset, p??·»}$$

TBD

A.3 Exclusivity Satisfaction

We start by defining a general abstract way of specifying sets with valid combinations of values drawn from a parameter type τ.

$i \in I_\tau$	::=	τ	can contain this value
	$\mid$	$\otimes(i, \ldots, i)$	at most one of these allowed to contain
	$\mid$	$\cup(i, \ldots, i)$	any of these allowed to contain

In effect we identify the occurrences of labels within this I-structure, and then check the multiplicity constraints:

$$\begin{array}{rcl} L \textbf{ lsat } I_{Lbl} & \widehat{=} & res = ok \\ \textbf{where} & & (res, \_) = occChk(occ_L I_{Lbl}) \end{array}$$

The *occ* function takes a set L of τ and an I_τ and returns a $I_{\mathbb{B}}$ that records if the corresponding element of τ is present in L.

$$\begin{array}{rcl} occ & : & \mathcal{P}\tau \to I_\tau \to I_{\mathbb{B}} \\ occ_L\ \ell & \widehat{=} & \ell \in L \\ occ_L\ \otimes(i_1, \ldots, i_n) & \widehat{=} & \otimes(occ_L\ i_1, \ldots, occ_L\ i_n) \\ occ_L\ \cup(i_1, \ldots, i_n) & \widehat{=} & \cup(occ_L\ i_1, \ldots, occ_L\ i_n) \end{array}$$

The *occChk* function pattern matches across the boolean values to see if constraints are satisfied. The first component of the result is an overall ok/fail indicator, while the second boolean component indicates if values are present in

any component.

$$
\begin{array}{rcl}
occChk & : & I_{\mathbb{B}} \to (\{ok, fail\} \times \mathbb{B}) \\
occChk(b) & \widehat{=} & (ok, b) \\
occChk(\cup(i_1, \ldots, i_n)) & \widehat{=} & (fail, \_), \ \textbf{if } \exists j \bullet occChk(i_j) = (fail, \_) \\
 & & (ok, b_1 \vee \cdots \vee b_n), \ \textbf{if } \forall j \bullet occChk(i_j) = (ok, b_j) \\
occChk(\otimes(i_1, \ldots, i_n)) & \widehat{=} & (fail, \_), \ \textbf{if } \exists j \bullet occChk(i_j) = (fail, \_) \\
 & & (fail, \_) \text{ if more than one } (ok, true) \\
 & & (ok, false) \text{ if all are } (ok, false) \\
 & & (ok, true) \text{ if exactly one } (ok, true)
\end{array}
$$

We note, as a consequence of the above definitions, that

$$\emptyset \quad \textbf{lsat} \quad I \quad \text{«emp-lsat-l»}$$

for any label-set invariant I.

We introduce a shorthand for invariants illustrated as follows.

$$
\begin{array}{rcl}
[a, b, c | d, e | f] & = & ls \ \textbf{lsat} \ \otimes (\cup(a, b, c), \cup(d, e), f) \\
[a | (b | c), (d | e) | f] & = & ls \ \textbf{lsat} \ \otimes (a, \cup(\otimes(b, c), \otimes(d, e)), f)
\end{array}
$$

In effect we make the involvement of ls implicit, and use bar (|) and comma (,) to replace $\otimes$ and $\cup$ respectively. We also have a shorthand that just denotes the non-intersecting nature of the arguments. E.g., $\{A|B|C\}$ asserts that A, B and C are mutually disjoint, without any reference to ls or any other set.

A.4 Proof of DL-perm

TBD

A.5 Proof of LE-perm

TBD

A.6 Proof of DL-drop

TBD

A.7 Proof of LE-drop

TBD

A.8 Proof of DL-subset

TBD

A.9 Proof of DL-subset

TBD

A.10 Proof of LE-out-of-scope

TBD

A.11 Proof of LE-trivial

TBD

A.12 Proof of ???

TBD

B Denotation BITS

$$
\begin{array}{rcll}
\langle a \rangle & \widehat{=} & [in|out] \wedge \mathbf{DL}(\mathbf{W}(A(in|a|out))) & \text{«·sem:atomic·»} \\
P \,;;\, Q & \widehat{=} & [in|\ell_g|out] \wedge & \\
 & & \mathbf{DL}(\mathbf{W}(P[g_{:1}, \ell_g/g, out] \vee Q[g_{:2}, \ell_g/g, in])) & \text{«·sem:seq·»} \\
P \parallel Q & \widehat{=} & [in|(\ell_{g1}|\ell_{g1:}), (\ell_{g2}|\ell_{g2:})|out] \wedge & \text{«·sem:par·»} \\
 & & \mathbf{DL}(\mathbf{W}(\quad A(in|ii|\ell_{g1}, \ell_{g2}) & \\
 & & \qquad \vee P[g_{1::}, \ell_{g1}, \ell_{g1:}/g, in, out] & \\
 & & \qquad \vee Q[g_{2::}, \ell_{g2}, \ell_{g2:}/g, in, out] & \\
 & & \qquad \vee A(\ell_{g1:}, \ell_{g2:}|ii|out) \;)) & \\
P + Q & \widehat{=} & [in|\ell_{g1}|\ell_{g2}|out] \wedge & \text{«·sem:NDC·»} \\
 & & \mathbf{DL}(\mathbf{W}(\quad A(in|ii|\ell_{g1}) & \\
 & & \qquad \vee A(in|ii|\ell_{g2}) & \\
 & & \qquad \vee P[g_{1:}, \ell_{g1}/g, in] & \\
 & & \qquad \vee Q[g_{2:}, \ell_{g2}/g, in] \;)) & \\
P^* & \widehat{=} & [in|\ell_g|out] \wedge & \text{«·sem:star·»} \\
 & & \mathbf{DL}(\mathbf{W}(\quad A(in|ii|out) & \\
 & & \qquad \vee A(in|ii|\ell_g) & \\
 & & \qquad \vee P[g_{:}, \ell_g, in/g, in, out] \;)) & \\
P \blacktriangleleft c \blacktriangleright Q & \widehat{=} & [in|\ell_{g1}|\ell_{g2}|out] \wedge & \text{«·sem:cond·»} \\
 & & \mathbf{DL}(\mathbf{W}(\quad A(in|c \wedge ii|\ell_{g1}) & \\
 & & \qquad \vee A(in|\neg c \wedge ii|\ell_{g2}) & \\
 & & \qquad \vee P[g_{1:}, \ell_{g1}/g, in] & \\
 & & \qquad \vee Q[g_{2:}, \ell_{g2}/g, in] \;)) & \\
c \circledast P & \widehat{=} & [in|\ell_g|out] \wedge & \text{«·sem:iter·»} \\
 & & \mathbf{DL}(\mathbf{W}(\quad A(in|\neg c \wedge ii|out) & \\
 & & \qquad \vee A(in|c \wedge ii|\ell_g) & \\
 & & \qquad \vee P[g_{:}, \ell_g, in/g, in, out] \;)) &
\end{array}
$$

B.1 Non-Det Choice (old)