"Thea! No! What are you doing?"
"I'm going to let the titans in."
The amulet flared brightly,
The gate swung open . . .
–from Into the Labyrinth

Chapter 4

Programs and Their Properties

A brief review on the programming logic UNITY will be presented. Special attention will be given to the issue of compositionality, that is, the ability to decompose a specification of a program into specifications of its components. A new progress operator which is more compositional will be introduced, and a set of calculational laws for the new operator will be provided.

4.1 Introduction

THE previous chapter introduced the basic building blocks which constitute a program, namely variables and actions. In this chapter we will talk about programs and their behavior. Basically, a program is only a collection of actions. During its execution, the actions are executed in a certain order. It is however possible to encode the ordering in the actions themselves by adding program counters. In this sense, a program is really a collection of actions, without any ordering. This way of viewing programs is especially attractive if we consider a parallel execution of actions where strict orderings begin to break down. In fact, a number of distributed programming logics are based on this idea. Examples thereof are Action Systems [Bac90], Temporal Logic of Action [Lam90], and UNITY [CM88].

Recall that we aim to verify the work of Lentfert [Len93] on a general, self-stabilizing, and distributed algorithm to compute minimal distances in a hierarchical network. Lentfert's work is based on the programming logic UNITY, which was chosen mainly for its simplicity and its high level of abstraction. We will simply follow his choice.

Examples of programs derivation and verification using UNITY are many. The introductory book to UNITY [CM88] itself contains numerous examples, ranging from a simple producer-consumer program, to a parallel garbage collection program. Realistic problems have also been addressed. In [Sta93] Staskauskas derives an I/O sub-system of an existing operating system, which is responsible for allocating I/O resources. In [Piz92] Pizzarello used UNITY to correct an error found in a large operating system. The fault had been corrected before, and verified using the traditional approach of testing and tracing [KB87]. It is interesting to note that the amount of work using UNITY is small, compared to that of the traditional approach. A review of Pizzarello's industrial experience on the use of UNITY can be found in [Piz91]. In [CKW+91] Chakravarty and his colleagues developed a simulation of the diffusion and aggregation of particles in a cement like porous media.

In practice, many useful programs do not, in principle, terminate; some examples are file servers, routing programs in computer networks, and control systems in an air plane. For such a program, its responses during its execution are far more important than the state it ends up with when it eventually terminates. To specify such a program we cannot therefore use Hoare triples as we did in Chapter 3. Two aspects are especially important: *progress* and *safety*. A progress property of a program expresses what the program is expected to eventually realize. For example, if a message is sent through a routing system, a progress property may state that eventually the message will be delivered to its destination. A safety property, on the other hand, tells us what the program should not do: for example, that the message is only to be delivered to its destination, and not to any other computer. The two kinds of properties are not mutually exclusive. For example, a safety property, stating that a computer in a network should *not* either ignore an incoming message or discard it, implies that the computer should either consume the message or re-route it to some neighbors. This states progress. In UNITY there is an operator called *unless* to express safety properties, and two more operators, *ensures* and *leads-to*, to express progress properties.

In [CM88] two kinds of program composition are discussed. In the first, programs are composed by simply 'merging' them. This can be thought of as modelling parallel composition. In the second, called *super-position*, actions may be extended with concurrent assignments to fresh variables. Both will be discussed here, but in a quite different light than in [CM88]. In addition we will discuss program compositions in which guards may be added. In the applications verified in this thesis however, only parallel composition is used.

Ideally, a program is developed hand in hand with its proof. One starts with a specification, which is refined, step by step using a set of rules, until an implementable program is obtained. In sequential programming, one begins with an imaginary program. In each development step, the program is refined by adding more details to it. The original specification is then reduced to specifications for the components of the programs. Subsequently, each component can be developed in isolation. This kind of hierarchical program decomposition is not an issue which is very well explored in UNITY. Safety properties decompose nicely. Unfortunately, the same cannot be said for progress properties, especially with respect to the parallel composition. A general law to decompose progress is provided by Singh [Sin89]. Similar laws were also used by Lentfert to decompose self-stabilization. An unpleasant discovery that was made during our research was that these laws were all flawed[1]. Fortunately, the flaw was not so serious that it bears no consequence to Lentfert's results. To facilitate the mechanical verification of Lentfert's work, we have also re-written Lentfert's proofs as to make them simpler and more intuitive. This is made possible by, among other things, the Transparency Law (2.2.4). If the reader recalls the discussion in Section 2.2, the law is used to assign the task of establishing a progress property to a write-disjoint

[1] Partly, the discovery is due to the absolute rigor imposed by HOL. When a supposedly obvious fact seems to be impossible to be proven in HOL, it is a good indication that we do not formulate the fact correctly and completely.

```
prog  Fizban
read  {a, x, y}
write {x, y}
init  true
assign
      if a = 0 then x := 1 else skip
[]    if a ≠ 0 then x := 1 else skip
[]    if x ≠ 0 then y, x := y + 1, 0 else skip
```

Figure 4.1: *The program* Fizban

component. For the purpose of this law, a new progress operator will be introduced.

Section 4.2 briefly reviews the ideas behind UNITY. Section 4.3 discusses the standard UNITY operators to express behavior of a program. Various basic laws used to reason about them are presented in Section 4.4. Section 4.5 introduces the new progress operator mentioned above. Various laws about parallel composition will be discussed in Section 4.7. Section 4.8 discusses the parallel composition of write-disjoint programs. Section 4.9 briefly discusses some other kinds of program composition. And finally, Section 4.10 briefly discusses the soundness of UNITY with respect to its operational semantics.

4.2 UNITY Programs

UNITY is a programming logic invented by Chandy and Misra in 1988 [CM88] for reasoning about safety and progress behavior of distributed programs. Figure 4.1 displays an example. The precise syntax will be given later.

The read and write sections declare, respectively, the read and write variables of the program[2]. The init section describes the assumed initial states of the program. In the program Fizban in Figure 4.1, the initial condition is true, which means that the program may start in any state. The assign section lists the actions of the programs, separated by the [] symbol.

The actions in a UNITY program are assumed to be *atomic*. An execution of a UNITY program is an infinite and interleaved execution of its actions. In a fully parallel system, each action may be thought of as being executed by a separate processor. To make our reasoning independent from the relative speed of the processors, nothing is said about when a particular action should be executed. Consequently, there is no ordering imposed on the execution of the actions. There is a *fairness* condition though:

[2] When declaring a variable one may also want to declare its type (instead of assuming all variables to be of type, say, $\mathbb{N}$). UNITY does not disallow such a declaration, but its logic as in [CM88] does not include laws to deal with subtleties which may arise from typing the variables. For example, nothing is said about the effect of assigning the number 10 to a $\mathbb{B}$ valued variable. Of course it is possible to extend UNITY with a type theory, but this issue is beyond the scope of this thesis.

in a UNITY execution, which is infinite, each action must be executed infinitely often (and hence cannot be ignored forever).

For example, by now the reader should be able to guess that in the program Fizban, eventually $x = 0$ holds and that if $M = y$, then eventually $M < y$ holds.

Notice that the program Fizban resembles the program Fizb in Figure 3.1_{22}, which has the following body:

```
do forever
   begin
   if a = 0 then x := 1 else skip ;
   if a ≠ 0 then x := 1 else skip ;
   if x ≠ 0 then y, x := y + 1, 0 else skip
   end
```

In fact, Fizb is a sequential implementation of Fizban. Indeed, as far as UNITY concerns, the actions can be implemented sequentially, fully parallel, or anything in between, as long as the atomicity and the fairness conditions of UNITY are being met. Perhaps, the best way to formulate the UNITY's philosophy is as worded by Chandy and Misra in [CM88]:

> *A UNITY program describes* ***what*** *should be done in the sense that it specifies the initial state and the state transformations (i.e., the assignments). A UNITY program does not specify precisely* ***when*** *an assignment should be executed ... Neither does a UNITY program specify* ***where*** *(i.e., on which processor in a multiprocessor system) an assignment is to be executed, nor to which process an assignment belongs.*

That is, in UNITY one is encouraged to concentrate on the 'real' problem, and not to worry about the actions ordering and allocation, as such are considered to be implementation issues.

Despite its simple view, UNITY has a relatively powerful logic. The wide range of applications considered in [CM88] illustrates this fact quite well. Still, to facilitate programming, more structuring methods would be appreciated. An example thereof is sequential composition of actions. Structuring is an issue which deserves more investigation in UNITY.

By now the reader should have guessed that a UNITY program P can be represented by a quadruple (A, J, V_r, V_w) where $A \subseteq$ Action is a set consisting of the actions of P, $J \in$ Pred is a predicate describing the possible initial states of P, and $V_r, V_w \in$ Var are sets consisting of respectively read and write variables of P. The set of all such structures will be denoted by Uprog. So, all UNITY programs will be a member of this set, although, as will be made clear later, the converse is not necessarily true.

To access each component of an Uprog object, the destructors **a**, **ini**, **r**, and **w** are introduced. They satisfy the following property:

Theorem 4.2.1 Uprog DESTRUCTORS *ID_UPROG*

$$P \in \mathsf{Uprog} \;=\; (P \;=\; (\mathbf{a}P, \mathbf{ini}P, \mathbf{r}P, \mathbf{w}P))$$

◀

In addition, the input variables of P, that is, the variables read by P but not written by it, is denoted by $\mathbf{i}P$:

$$\mathbf{i}P = \mathbf{r}P \backslash \mathbf{w}P \tag{4.2.1}$$

4.2.1 The Programming Language

Below is the syntax of UNITY programs that is used in this thesis. The syntax deviates slightly from the one in [CM88][3].

$$\begin{array}{lll} \langle \textit{Unity Program} \rangle & ::= & \mathsf{prog}\ \langle \textit{name of program} \rangle \\ & & \mathsf{read}\ \langle \textit{set of variables} \rangle \\ & & \mathsf{write}\ \langle \textit{set of variables} \rangle \\ & & \mathsf{init}\ \langle \textit{predicate} \rangle \\ & & \mathsf{assign} \langle \textit{actions} \rangle \end{array}$$

actions is a list of *action*s separated by $[]$. An *action* is either a single action or a set of indexed actions.

$$\begin{array}{lll} \langle \textit{actions} \rangle & ::= & \langle \textit{action} \rangle \mid \langle \textit{action} \rangle \;[]\; \langle \textit{actions} \rangle \\ \langle \textit{action} \rangle & ::= & \langle \textit{single action} \rangle \mid ([]i : i \in V : \langle \textit{actions} \rangle_i) \end{array}$$

A single action is either a simple assignment such as $x := x + 1$ or a guarded action. A simple assignment can simultaneously assign to several variables. An example is $x, y := y, x$ which swaps the values of x and y. The precise meaning of assignments has been given in Chapter 3. A guarded action has the form:

$$\begin{array}{l} \mathsf{if}\ g_1\ \mathsf{then}\ a_1 \\ \quad g_2\ \mathsf{then}\ a_2 \\ \quad g_3\ \mathsf{then}\ a_3 \\ \quad \ldots \end{array}$$

An else part can be added with the usual meaning. If more than one guard is true then one is selected non-deterministically. If none of the guards is true, a guarded action behaves like skip. So, for example, the action "if $a \neq 0$ then $x := 1$ else skip" from the program Fizban can also be written as "if $a \neq 0$ then $x := 1$".

In addition we have the following requirements regarding the well-formedness of a UNITY program:

i. A program has at least one action.

[3] We omit the always section and we find it necessary to split the declare section into read and write parts

ii. The actions of a program should only write to the declared write variables.

iii. The actions of a program should only depend on the declared read variables.

iv. A write variable is also readable.

These are perfectly natural requirements for a program. Most programs that a programmer writes will satisfy them[4].

In Chapter 3 the notions of ignored and invisible variables have been explained. If a set of variables V^{c} is ignored by an action a, that is, $V^{\mathsf{c}} \nleftarrow a$, then a can only write to the variables in V. So, *ii* can be encoded as:

$$(\forall a : a \in \mathbf{a}P : (\mathbf{w}P)^{\mathsf{c}} \nleftarrow a)$$

Quite similarly, If V^{c} is invisible to a, that is, $V^{\mathsf{c}} \nrightarrow a$, then a will only depend on the variables in V. So, *iii* can be encoded as:

$$(\forall a : a \in \mathbf{a}P : (\mathbf{r}P)^{\mathsf{c}} \nrightarrow a)$$

In [CM88] it is required that all actions in a UNITY program are deterministic. We find that this restriction is unnecessary. If a program contains a non-deterministic action, the only consequence is that the program will probably show less predictable behavior. In [CM88] it is also required that all actions in a UNITY program are terminating. This is a perfectly logical requirement because if a statement does not terminate, then no further progress will be made, which violates the infinite execution model of UNITY. Here, we will refrain from imposing this requirement. We consider non-termination to be as bad as chaos, which is totally non-deterministic. A program which contains chaos can always be refined by removing some non-determinism at the action level. We leave it to the designers to come up with actions which are terminating.

The requirement that a UNITY guarded action behaves as skip if none of its guards is true means that all UNITY actions are required to be always-enabled (= not potentially miraculous). It is also not necessary to require this explicitly, but we will do it anyway. As we will see later, this requirement is crucial for a law called Impossibility Law.

Recall that any UNITY program is an object of type Uprog. Now we can define a predicate Unity to define the well-formedness of an Uprog object. From now on, with a "UNITY program", we mean an object satisfying Unity.

Definition 4.2.2 Unity *UNITY*

$$\begin{aligned} \mathsf{Unity}.P \;=\; & (\mathbf{a}P \neq \emptyset) \;\wedge\; (\mathbf{w}P \subseteq \mathbf{r}P) \;\wedge\; (\forall a : a \in \mathbf{a}P : \Box_{\mathsf{En}} a) \;\wedge \\ & (\forall a : a \in \mathbf{a}P : (\mathbf{w}P)^{\mathsf{c}} \nleftarrow a) \;\wedge\; (\forall a : a \in \mathbf{a}P : (\mathbf{r}P)^{\mathsf{c}} \nrightarrow a) \end{aligned}$$

◀

---

[4] One may however want to drop requirements *i* and *iv*. *i* is required by some laws. So, if omitted it will re-appear somewhere else. *iv* was added because it seems convenient.

4.2.2 Parallel Composition

A consequence of the absence of ordering in the execution of a UNITY program is that the parallel composition of two programs can be modelled by simply merging the variables and actions of both programs. In UNITY parallel composition is denoted by []. In [CM88] the operator is also called *program union*.

Definition 4.2.3 PARALLEL COMPOSITION *PAR*

$$P[]Q \;=\; (\mathbf{a}P \cup \mathbf{a}Q, \mathbf{ini}P \wedge \mathbf{ini}Q, \mathbf{r}P \cup \mathbf{r}Q, \mathbf{w}P \cup \mathbf{w}Q)$$

Parallel composition is reflexive, commutative, and associative. It has a unit element, namely $(\emptyset, \mathsf{true}, \emptyset, \emptyset)$ (although this is not a well-formed UNITY program).

As an example, we can compose the program Fizban in Figure 4.1 in parallel with the program below:

prog TikTak
read $\{a\}$
write $\{x\}$
init true
assign if $a = 0$ then $a := 1$ [] if $a \neq 0$ then $a := 0$

The resulting program consists of the following actions (the else skip part of the actions in Fizban will be dropped, which is, as remarked in Section 4.2.1, allowed):

a_0 : if $a = 0$ then $a := 1$
a_1 : if $a \neq 0$ then $a := 0$
a_2 : if $a = 0$ then $x := 1$
a_3 : if $a \neq 0$ then $x := 1$
a_4 : if $x \neq 0$ then $y, x := y + 1, 0$

Whereas in Fizban $x \neq 0$ will always hold somewhere in the future, the same cannot be said for Fizban [] TikTak. Consider the execution sequence $(a_0; a_2; a_1; a_3; a_4)*$, which is a fair execution and therefore a UNITY execution. In this execution, the assignment $x := 1$ will never be executed. If initially $x \neq 1$ this will remain so for the rest of this execution sequence.

4.3 Programs' Behavior

To facilitate reasoning about program behavior UNITY provides several primitive operators. The discussion in Section 4.2 revealed that an execution of a UNITY program never, in principle, terminates. Therefore we are going to focus on the behavior of a program *during* its execution. Two aspects will be considered: safety and progress. Safety behavior can be described by an operator called unless. By the fairness condition of UNITY, an action cannot be continually ignored. Once executed, it may

Figure 4.2: unless *and* ensures. *The predicates $p \wedge \neg q$ and q define sets of states. The arrows depict possible transitions between the two sets of states. The arrow marked with* ! *is a guaranteed transition.*

induce some progress. For example, the execution of the action a_4 in Fizban [] TikTak will establish $x = 0$ regardless when it is executed. Actually, any action which is not skip or chaos induces some meaningful progress. This kind of single-action progress is described by an operator called ensures.

In the sequel, P, Q, and R will range over UNITY programs; a, b, and c over Action; and p, q, r, s, J and K over Pred.

Definition 4.3.1 UNLESS *UNLESS*

$$_P\vdash p \text{ unless } q \;=\; (\forall a : a \in \mathbf{a}P : \{p \wedge \neg q\}\ a\ \{p \vee q\})$$

Definition 4.3.2 ENSURES *ENSURES*

$$_P\vdash p \text{ ensures } q \;=\; (\,_P\vdash p \text{ unless } q) \;\wedge\; (\exists a : a \in \mathbf{a}P : \{p \wedge \neg q\}\ a\ \{q\})$$

Intuitively, $_P\vdash p$ unless q implies that once p holds during an execution of P, it remains to hold at least until q holds. Figure 4.2 may be helpful. Note that this given interpretation says nothing about what p unless q means if p never holds during an execution. $_P\vdash p$ ensures q encompasses p unless q, and adds that there should also exist an action that can, and because of the fairness assumption of UNITY, will establish q.

If progress can be made from p to q, and from q to r, we would expect that progress can also be made from p to r. Similarly, if progress can be made from p_1 to q_1, and from p_2 to q_2, then starting from either p_1 or p_2, progress will be made to either q_1 or q_2. These two are natural properties of progress. However, ensures does not have these properties. It is because it only describes single-action progress. To describe the combined effect of several actions, the smallest transitive and disjunctive closure of ensures has to be used. This relation is denoted by $\mapsto$ ("leads-to"). Leads-to describes progress in general.

The notion of transitivity is well known; we will write Trans.R to denote that a relation R is transitive. The notion of left-disjunctivity is defined below:

Definition 4.3.3 LEFT DISJUNCTIVE RELATION *LDISJ_DEF*

A relation R over $A \rightarrow \mathbb{B}$ is called *left-disjunctive*, denoted $\mathsf{Ldisj}.R$ iff for all $q \in A \rightarrow \mathbb{B}$ and all non-empty sets W (of predicates over A):

$$(\forall p : p \in W : R.p.q) \;\Rightarrow\; R.(\exists p : p \in W : p).q$$

◀

The formula above may be confusing. Notice that the $\exists$ on the right hand side of $\Rightarrow$ denotes $\exists$ on the predicate level, not Boolean level. If we write the formula without notational overloading (as warned in Section 3.3), it looks like:

$$(\forall p : p \in W : R.p.q) \;\Rightarrow\; R.(\lambda s.\ (\exists p : p \in W : p.s)).q$$

In a simple case, $\mathsf{Ldisj}.R$ implies $R.p_1.q \wedge R.p_2.q \Rightarrow R.(p_1 \vee p_2).q$. For example, unless is left-disjunctive. Now we can define TDC, the smallest transitive and disjunctive closure of a given relation, as follows:

Definition 4.3.4 SMALLEST TRANSITIVE AND DISJUNCTIVE CLOSURE *TDC*

$$\mathsf{TDC}.R.p.q \;=\; (\forall S : R \subseteq S \wedge \mathsf{Trans}.S \wedge \mathsf{Ldisj}.S : S.p.q)$$

◀

Note that we can also define TDC as follows:

$$\mathsf{TDC}.R \;=\; \cap\{S \mid R \subseteq S \wedge \mathsf{Trans}.S \wedge \mathsf{Ldisj}.S\} \tag{4.3.1}$$

which shows more clearly that TDC is some smallest closure of R. Note that the total relation is transitive and left-disjunctive. Hence the set $\{S \mid R \subseteq S \wedge \mathsf{Trans}.S \wedge \mathsf{Ldisj}.S\}$ is non-empty, and hence $\mathsf{TDC}.R$ is non-trivial.

Now we can define $\mapsto$ as follows:

Definition 4.3.5 LEADS-TO *LEADSTO*

$$(\lambda p, q.\ {}_P\vdash p \mapsto q) \;=\; \mathsf{TDC}.(\lambda p, q.\ {}_P\vdash p \text{ ensures } q)$$

◀

Introducing TDC may seem only to serve as adding flavor to the notation, but it is not. Many useful properties of $\mapsto$ are actually pure properties of TDC. Later, we introduce a variant of $\mapsto$. The new progress operator is also a TDC-relation, but based on a different relation. This new operator will then automatically inherit all properties of TDC. This has saved a lot of time and effort in mechanically verifying the properties of the new operator.

As an example, the program Fizban in Figure 4.1, which has the following assign section:

$$\begin{array}{l} \quad \textsf{if } a = 0 \textsf{ then } x := 1 \\ [\!] \ \textsf{if } a \neq 0 \textsf{ then } x := 1 \\ [\!] \ \textsf{if } x \neq 0 \textsf{ then } y, x := y + 1, 0 \end{array}$$

satisfies the following properties:

$$_{\mathsf{Fizban}}\vdash \quad (a = X) \text{ unless false} \tag{4.3.2}$$

$$_{\mathsf{Fizban}}\vdash \quad \text{true unless } (x = 1) \tag{4.3.3}$$

$$_{\mathsf{Fizban}}\vdash \quad (a = 0) \text{ ensures } (x = 1) \tag{4.3.4}$$

$$_{\mathsf{Fizban}}\vdash \quad (a \neq 0) \text{ ensures } (x = 1) \tag{4.3.5}$$

$$_{\mathsf{Fizban}}\vdash \quad \text{true} \mapsto (x = 1) \tag{4.3.6}$$

If (4.3.2) holds for any X then it states that Fizban cannot change the value of a. (4.3.3) is an example of a property that trivially holds in any program. The reader can check it by unfolding the definition of unless.

(4.3.4) and (4.3.5) describe single-action progress from, respectively $a = 0$ and $a \neq 0$ to $x = 1$. Because $\mapsto$ is a closure of ensures, and hence includes ensures, we conclude that $(a = 0) \mapsto (x = 1)$ and $(a \neq 0) \mapsto (x = 1)$ also hold. Using the disjunctivity property of progress, we can conclude (4.3.6), which states that eventually $x = 1$. Note that ensures is *not* disjunctive. So, despite (4.3.4) and (4.3.5), we cannot conclude:

$$_{\mathsf{Fizban}}\vdash \text{true ensures } (x = 1) \tag{4.3.7}$$

The above cannot be true because there is no single action in Fizban which can establish $x = 1$ regardless in which state it is executed. The point is that single-action progress is a bit special and cannot be expressed using $\mapsto$. The composition Fizban [] TikTak, despite property (4.3.6), does not satisfy $\text{true} \mapsto (x = 1)$. On the other hand, composing TikTak with a program P that does satisfy (4.3.7) yields a program that does satisfy $\text{true} \mapsto (x = 1)$.

Properties of the form $_P\vdash p \text{ unless false}$ are called *stable* properties, which are very useful properties because they express that once p holds during any execution of P, it will remain to hold. Because of their importance we will define a separate abbreviation:

Definition 4.3.6 STABLE PREDICATE *STABLE*

$$_P\vdash \circlearrowleft p \;=\; _P\vdash p \text{ unless false}$$

$_P\vdash \circlearrowleft p$ is pronounced "*p is stable in P*" and p is called a *stable predicate*. Notice that $\circlearrowleft$ can also be defined as follows:

$$_P\vdash \circlearrowleft p \;=\; (\forall a : a \in \mathbf{a}P : \{p\}\ a\ \{p\}) \tag{4.3.8}$$

Consequently, if p holds initially and is stable in P, it will hold throughout any execution of P, and hence it is an *invariant*. There seem to be at least two notions of invariant. Here, we define an invariant of a program P as a predicate that holds throughout any execution of P. Note that with this definition, an invariant is not necessarily stable. For example, consider a program P consisting of a single action a:

if $x = 1$ then $x := 2$

If $\mathbf{ini}P = (x = 0)$, then $(x = 0) \vee (x = 1)$ holds initially and throughout the execution of P. Hence it is an invariant. However, $(x = 0) \vee (x = 1)$ is *not* stable (because if $x = 1$ then a will assign 2 to x).

Invariants and stable predicates are disjunctive and conjunctive. That is, if p and q are invariants (or stable), then so are $p \wedge q$ and $p \vee q$. However, whereas invariants are monotonic with respect to $\Rightarrow$, stable predicates are not. Invariants are useful, but for self-stabilizing programs, whose initial condition can be as liberal as **true**, the notion of stability is more useful.

4.4 UNITY Laws

Figures 4.3 and 4.4 display a set of basic laws for **unless** properties. Figure 4.5 displays a set of basic laws for $\circlearrowleft$, and 4.6 for **ensures**. The properties are taken from [CM88]. As a notational convention: *if it is clear from the context which program P is meant, we often omit it from a formula.* For example we may write p **unless** q to mean $\ {}_P\!\vdash p$ **unless** q. Also, for laws we write, for example:

$$P: \ \frac{\ldots (p \textbf{ unless } q) \ldots}{r \mapsto s} \quad \text{to abbreviate:} \quad \frac{\ldots ({}_P\!\vdash p \textbf{ unless } q) \ldots}{{}_P\!\vdash r \rightarrowtail s}$$

Note that the **unless** CONJUNCTION and DISJUNCTION laws in Figure 4.3 can be generalized to combine an arbitrary number of **unless** properties (the generalization has also been verified). A similar remark also holds for the conjunction and disjunction of $\circlearrowleft$. Note that **ensures** is conjunctive but *not* disjunctive. Note also that the **ensures** INTRODUCTION law depends on the fact that the program P is non-empty (otherwise there is no sense in talking about 'single-action' progress).

There is another law of **ensures** called Impossibility Law, stating that it is impossible to progress to **false** unless if one starts from **false**, which is just not possible:

Theorem 4.4.16 **ensures** IMPOSSIBILITY *ENSURES_IMPOS*

$$P: \ \frac{p \textbf{ ensures } \mathsf{false}}{[\neg p]}$$

A crucial assumption to this law is that all actions in the program P are always-enabled. Otherwise, if a miracle is possible, then it can be used to establish **false**, and then it would follow by the **ensures** POST-WEAKENING law in page 47 that any progress is possible. The proof of the IMPOSSIBILITY law is as follows:

Proof:

By definition, ${}_P\!\vdash p$ **ensures false** implies that there exists an action $a \in \mathbf{a}P$ such that $\{p\}\ a\ \{\mathsf{false}\}$ holds. Since P is a UNITY program, a is always enabled. We derive:

$\{p\}\ a\ \{\mathsf{false}\}$

$=$ { definition Hoare triple }

Theorem 4.4.1 unless INTRODUCTION *UNLESS_IMP_LIFT1, UNLESS_IMP_LIFT2*

$$P: \frac{[p \Rightarrow q] \ \vee \ [\neg p \Rightarrow q]}{p \text{ unless } q}$$

Theorem 4.4.2 unless POST-WEAKENING *UNLESS_CONSQ_WEAK*

$$P: \frac{(p \text{ unless } q) \ \wedge \ [q \Rightarrow r]}{p \text{ unless } r}$$

Theorem 4.4.3 unless CONJUNCTION *UNLESS_CONJ*

$$P: \frac{(p \text{ unless } q) \ \wedge \ (r \text{ unless } s)}{p \wedge r \text{ unless } (p \wedge s) \vee (r \wedge q) \vee (q \wedge s)}$$

Theorem 4.4.4 unless DISJUNCTION *UNLESS_DISJ*

$$P: \frac{(p \text{ unless } q) \ \wedge \ (r \text{ unless } s)}{p \vee r \text{ unless } (\neg p \wedge s) \vee (\neg r \wedge q) \vee (q \wedge s)}$$

Figure 4.3: *Basic laws for* unless.

Corollary 4.4.5 unless REFLEXIVITY *UNLESS_REFL*

p unless p

Corollary 4.4.6 ANTI-REFLEXIVITY *UNLESS_ANTI_REFL*

$\neg p$ unless p

Corollary 4.4.7 SIMPLE CONJUNCTION *UNLESS_SIMPLE_CONJ*

$$P: \frac{(p \text{ unless } q) \ \wedge \ (r \text{ unless } s)}{p \wedge r \text{ unless } q \vee s}$$

Corollary 4.4.8 SIMPLE DISJUNCTION *UNLESS_SIMPLE_DISJ*

$$P: \frac{(p \text{ unless } q) \ \wedge \ (r \text{ unless } s)}{p \vee r \text{ unless } q \vee s}$$

Figure 4.4: *Some corollaries of* unless.

Theorem 4.4.9 $\circlearrowleft$ CONJUNCTION *STABLE_GEN_CONJ*

$$P: \frac{(\circlearrowleft p) \ \wedge \ (\circlearrowleft q)}{\circlearrowleft (p \wedge q)}$$

Theorem 4.4.10 $\circlearrowleft$ DISJUNCTION *STABLE_GEN_DISJ*

$$P: \frac{(\circlearrowleft p) \ \wedge \ (\circlearrowleft q)}{\circlearrowleft (p \vee q)}$$

Figure 4.5: *Basic laws of* $\circlearrowleft$.

Theorem 4.4.11 ensures INTRODUCTION *ENSURES_IMP_LIFT*

$$P: \frac{[p \Rightarrow q]}{p \text{ ensures } q}$$

Theorem 4.4.12 ensures POST-WEAKENING *ENSURES_CONSQ_WEAK*

$$P: \frac{(p \text{ ensures } q) \ \wedge \ [q \Rightarrow r]}{p \text{ ensures } r}$$

Theorem 4.4.13 ensures PROGRESS SAFETY PROGRESS (PSP) *ENSURES_PSP*

$$P: \frac{(p \text{ unless } q) \ \wedge \ (r \text{ unless } s)}{p \wedge r \text{ ensures } (p \wedge s) \vee (r \wedge q) \vee (q \wedge s)}$$

Figure 4.6: *Basic laws for* ensures

Corollary 4.4.14 ensures REFLEXIVITY *ENSURES_REFL*

p ensures p

Corollary 4.4.15 ensures CONJUNCTION *ENSURES_CONJ*

$$P: \frac{(p \text{ ensures } q) \ \wedge \ (r \text{ ensures } s)}{p \wedge r \text{ ensures } (p \wedge s) \vee (r \wedge q) \vee (q \wedge s)}$$

Figure 4.7: *Some corollaries of* ensures

$(\forall s, t :: p.s \wedge a.s.t \Rightarrow \mathsf{false})$
$=$ { predicate calculus }
$(\forall s, t :: p.s \Rightarrow \neg a.s.t)$
$=$ { a is always enabled, that is, $(\forall s :: (\exists t :: a.s.t))$ }
$(\forall s :: \neg p.s)$
$=$ { definition of [.] }
$[\neg p]$

▲

4.4.1 Transitive and Disjunctive Relations

Recall that the general progress operator $\mapsto$ is defined as the TDC of ensures. That is, it is the smallest transitive and left-disjunctive closure of ensures. Many laws of leads-to can be derived from general properties of TDC, which are presented in this subsection.

Being the smallest closure of some sort, TDC induces an induction principle. Actually it is trivial: since TDC.R is the smallest transitive and left-disjunctive closure of R, any other relation S which includes R, is transitive, and left-disjunctive will also include TDC.R.

Theorem 4.4.17 TDC INDUCTION *TDC_INDUCT1*

$$\frac{R \subseteq S \ \wedge \ \mathsf{Trans}.S \ \wedge \ \mathsf{Ldisj}.S}{\mathsf{TDC}.R \subseteq S}$$

◀

The principle gives a sufficient condition for a relation S to include TDC.R. In [CM88] the principle is invoked many times to prove other laws for $\mapsto$.

It can be shown that TDC.R itself includes R, and is transitive and left-disjunctive. These last two are the most basic properties progress. In addition, it is also monotonic with respect to $\subseteq$.

Theorem 4.4.18 *TDC_LIFT, TDC_TRANS, TDC_LDISJ*

$$(R \subseteq \mathsf{TDC}.R) \ \wedge \ \mathsf{Trans}.(\mathsf{TDC}.R) \ \wedge \ \mathsf{Ldisj}.(\mathsf{TDC}.R)$$

Theorem 4.4.19 TDC MONOTONICITY *TDC_MONO*

$$(R \subseteq S) \ \Rightarrow \ (\mathsf{TDC}.R \subseteq \mathsf{TDC}.S)$$

◀

In [CM88] some laws of the form $\mapsto \subseteq S$ are proven through $\mapsto \subseteq (\mapsto \cap S)$. In general, to show TDC.$R \subseteq (\mathsf{TDC}.R \cap S)$, using TDC INDUCTION it suffices to show:

$$(R \subseteq \mathsf{TDC}.(\mathsf{TDC}.R \cap S)) \ \wedge \ \mathsf{Trans}.(\mathsf{TDC}.R \cap S) \ \wedge \ \mathsf{Ldisj}.(\mathsf{TDC}.R \cap S) \qquad (4.4.1)$$

The above is often easier to prove. Note that part of it is by Theorem 4.4.18 trivial[5]. We have defined $\mapsto$ as the smallest transitive and left-disjunctive closure of ensures. One may wonder if replacing "transitive" with "right-transitive" would still define the same relation. That is, we would like to define $\mapsto$ as:

$$\mapsto \;=\; \cap\{S \mid (\,\mathsf{ensures}\; \subseteq S) \wedge (\,\mathsf{ensures}\,; S \subseteq S) \wedge \mathsf{Ldisj}.S\} \tag{4.4.2}$$

Note that $R;S \subseteq S$ means that S is right-transitive with respect to the base relation R. The question whether or not the above definition of $\mapsto$ is equal to the old one can be generalized to the question whether or not the following equation holds:

$$\mathsf{TDC}.R \;=\; \cap\{S \mid R \subseteq S \wedge (R;S \subseteq S) \wedge \mathsf{Ldisj}.S\} \tag{4.4.3}$$

Let $\Sigma^{\blacktriangleright}.R$ abbreviate the set $\{S \mid R \subseteq S \wedge (R;S \subseteq S) \wedge \mathsf{Ldisj}.S\}$ and let $\mathsf{TDC}^{\blacktriangleright}.R$ abbreviate $\cap(\Sigma^{\blacktriangleright}.R)$. Analogous to the case with TDC, one can show that $\mathsf{TDC}^{\blacktriangleright}.R$ itself is a closure of R, is right-transitive, and left-disjunctive. That is, $\mathsf{TDC}^{\blacktriangleright}.R$ is a member of $\Sigma^{\blacktriangleright}.R$. Being the smallest closure, $\mathsf{TDC}^{\blacktriangleright}$ also induces an induction principle:

$$\frac{R \subseteq S \;\wedge\; (R;S \subseteq S) \;\wedge\; \mathsf{Ldisj}.S}{\mathsf{TDC}^{\blacktriangleright}.R \subseteq S} \tag{4.4.4}$$

This, and TDC Induction state sufficient conditions for TDC to be equal to $\mathsf{TDC}^{\blacktriangleright}$. Among these conditions, the only non-trivial one is that $\mathsf{TDC}^{\blacktriangleright}.R$ is transitive, but this can be proven using the induction principle (4.4.4). The conclusion is that TDC and $\mathsf{TDC}^{\blacktriangleright}$ are equal:

Theorem 4.4.20 *TDC_EQU_Ri_TDC*

$$\mathsf{TDC} \;=\; \mathsf{TDC}^{\blacktriangleright}$$

As a corollary, the definition (4.4.2) of $\mapsto$ is equal to the old one. It also means the following induction principle is applicable to $\mapsto$:

$$P:\; \frac{(\mathsf{ensures} \subseteq S) \;\wedge\; (\mathsf{ensures};S \subseteq S) \;\wedge\; \mathsf{Ldisj}.S}{\mapsto \subseteq S} \tag{4.4.5}$$

There are cases where the induction principle above is more useful than the one in Theorem 4.4.17. For example, it has been crucial in proving a progress law called Completion law (page 52).

Another kind of induction that is often used in practice is well-founded induction. A relation $\prec \in A \to A \to \mathbb{B}$ is said to be *well-founded* if it is not possible to construct an infinite sequence of ever decreasing values in A. That is, $\ldots, x_2 \prec x_1 \prec x_0$ is

---

[5] For example, to prove $\mathsf{Trans}.(\mathsf{TDC}.R \cap S)$ we have to prove:

$$\mathsf{TDC}.R.p.q \wedge S.p.q \wedge \mathsf{TDC}.R.q.r \wedge S.q.r \;\Rightarrow\; \mathsf{TDC}.R.p.q \wedge S.p.r$$

for all p, q, and r. However, since by by Theorem 4.4.18 $\mathsf{TDC}.R$ is already transitive, we only need to show $S.p.r$.

not possible. For example the ordering $<$ on $\mathbb{N}$ and $\subset$ on sets and relations are well-founded. A well-founded relation admits the well-founded induction principle given below[6].

Definition 4.4.21 WELL-FOUNDED INDUCTION *ADMIT_WF_INDUCTION*

A relation $\prec \in A \to A \to \mathbb{B}$ is said to admit the well-founded induction if:

$$(\forall y :: (\forall x : x \prec y : p.x) \Rightarrow p.y) \;=\; (\forall y :: p.y)$$

◀

Let m be a function —so-called bound function— that maps State to A. If from p a program can progress to q, or else it maintains p while decreasing the value of m with respect to a well-founded ordering $\prec$, then, since $\prec$ is well-founded, it is not possible to keep decreasing m, and hence eventually q will be established. We call this principle *Bounded Progress*. It holds for any relation on predicates which is reflexive, transitive, and left-disjunctive.

Let in the sequel $\to$ be a relation over predicates over B (which can be program states), $\prec$ be a *well founded* relation over a *non-empty* type A, and m be a mapping from B to A.

Theorem 4.4.22 BOUNDED PROGRESS *BOUNDED_REACH_i*

$$\frac{\mathsf{Trans.} \to \;\wedge\; \mathsf{Ldisj.} \to \;\wedge\; q \to q \qquad (\forall M :: p \wedge (m = M) \to (p \wedge (m \prec M)) \vee q)}{p \to q}$$

▶

Note: The notation is overloaded. In the above, "$p \wedge (m = M)$" and "$(p \wedge (m \prec M)) \vee q$" actually mean, respectively, $(\lambda s.\; p.s \wedge (m.s = M))$ and $(\lambda s.\; (p.s \wedge (m.s \prec M)) \vee q.s)$

Proof:

$$\begin{array}{ll} & p \to q \\ \Leftarrow & \quad \{ \to \text{ is left-disjunctive and } B \text{ is non-empty } \} \\ & (\forall M :: p \wedge (m = M) \to q) \\ = & \quad \{ \text{WELL-FOUNDED INDUCTION } \} \\ & (\forall M :: (\forall M' : M' \prec M : p \wedge (m = M') \to q) \;\Rightarrow\; (p \wedge (m = M) \to q)) \end{array}$$

If M is a minimal element, thus there is no M' such that $M' \prec M$, then the assumption:

$$p \wedge (m = M) \to (p \wedge (m \prec M)) \vee q$$

is equal to $p \wedge (m = M) \to q$, which trivially implies the last formula in the derivation above. If M is not a minimal element:

[6] It has been showed that the above formulation of well-foundedness is actually equivalent with the admittance of the well-founded induction itself.

$$
\begin{array}{ll}
 & (\forall M' : M' \prec M : p \wedge (m = M') \rightarrow q) \\
\Rightarrow & \quad \{ \rightarrow \text{is left-disjunctive} \} \\
 & p \wedge (m \prec M) \rightarrow q \\
\Rightarrow & \quad \{ q \rightarrow q; \rightarrow \text{is left-disjunctive} \} \\
 & (p \wedge (m \prec M)) \vee q \rightarrow q \\
\Rightarrow & \quad \{ \text{from the assumption: } p \wedge (m = M) \rightarrow (p \wedge (m \prec M)) \vee q; \rightarrow \text{is transitive} \} \\
 & p \wedge (m = M) \rightarrow q
\end{array}
$$

▲

A corollary of the BOUNDED PROGRESS principle is the following, stating that if from $\neg p$ progress can be made in which the value of the bound function m decreases, then eventually p will be reached.

Theorem 4.4.23 INEVITABLE FULFILMENT *BOUNDED_ALWAYS_REACH_i*

$$\frac{\mathsf{Trans.} \rightarrow \ \wedge \ \mathsf{Ldisj.} \rightarrow \ \wedge \ q \rightarrow q \qquad (\forall M :: \neg p \wedge (m = M) \rightarrow (m \prec M))}{\mathsf{true} \rightarrow p}$$

◀

4.4.2 Laws of Leads-to

We are not going to use the $\mapsto$ operator very often. A variant thereof, better suited for our purpose, will be introduced in Section 4.5. For the sake of completeness, Figure 4.8 displays a set of basic laws for $\mapsto$. Some of them follow directly from the laws mentioned in the previous subsection. The reader may want to compare those laws with those of the new operator given in Section 4.5.

4.5 Introducing the Reach Operator

Consider the program P and TikToe in Figure 4.9. The program P can establish $x = 1$ if a is less than 2. So, it satisfies ${}_P\vdash (a < 2) \mapsto (x = 1)$. As with Hoare triples, we may strengthen the 'pre-condition' of $\mapsto$, and come up with the following property of P:

$$(b = 0) \wedge (a < 2) \mapsto (x = 1) \tag{4.5.1}$$

We note that P does not write to either a or b, and hence should maintain $(b = 0) \wedge (a < 2)$. We expect then, that if we compose P in parallel with another program which maintains the stability of $(b = 0) \wedge (a < 2)$, but does not write to x —the program TikToe is an example thereof— then (4.5.1) will be respected by the composition. At least, this works with $P[\!]$TikToe. The programs P and TikToe are *write-disjoint*, that is, they do not write to a common write variable. Compositions of write-disjoint

Theorem 4.4.24 $\mapsto$ INTRODUCTION *LEADSTO_IMPLICATION, LEADSTO_ENS_LIFT_thm*

$$P: \frac{[p \Rightarrow q] \ \vee \ (p \text{ ensures } q)}{p \mapsto q}$$

Theorem 4.4.25 $\mapsto$ INDUCTION *LEADSTO_INDUCT_thm1*

$$P: \frac{(\forall p, q :: (p \text{ ensures } q) \Rightarrow R.p.q) \ \wedge \ \mathsf{Trans}.R \ \wedge \ \mathsf{Ldisj}.R}{(\forall p, q :: (p \mapsto q) \Rightarrow R.p.q)}$$

Theorem 4.4.26 $\mapsto$ TRANSITIVITY *LEADSTO_TRANS_thm*

$$P: \frac{(p \mapsto q) \ \wedge \ (q \mapsto r)}{p \mapsto r}$$

Theorem 4.4.27 $\mapsto$ DISJUNCTION *LEADSTO_GEN_DISJ*

For all non-empty sets W:

$$P: \frac{(\forall i : i \in W : p.i \mapsto q.i)}{(\exists i : i \in W : p.i) \mapsto (\exists i : i \in W : q.i)}$$

Corollary 4.4.28 $\mapsto$ CANCELLATION *LEADSTO_CANCEL*

$$P: \frac{(p \mapsto q \vee b) \ \wedge \ (b \mapsto r)}{p \mapsto q \vee r}$$

Corollary 4.4.29 $\mapsto$ STRENGTHENING & WEAKENING *LEADSTO_ANTE_STRONG, LEADSTO_CONSQ_WEAK*

$$P: \frac{[p \Rightarrow q] \ \wedge \ (q \mapsto r) \ \wedge \ [r \Rightarrow s]}{p \mapsto s}$$

Theorem 4.4.30 $\mapsto$ PROGRESS SAFETY PROGRESS (PSP) *LEADSTO_PSP*

$$P: \frac{(p \mapsto q) \ \wedge \ (r \text{ unless } s)}{p \wedge r \mapsto (q \wedge r) \vee s}$$

Theorem 4.4.31 $\mapsto$ IMPOSSIBILITY *LEADSTO_IMPOS*

$$P: \frac{p \mapsto \mathsf{false}}{[\neg p]}$$

Theorem 4.4.32 $\mapsto$ COMPLETION *LEADSTO_COMPLETION*

For all *finite* sets W:

$$P: \frac{(\forall i : i \in W : p.i \mapsto q.i \vee r) \ \wedge \ (\forall i : i \in W : q.i \text{ unless } r)}{(\forall i : i \in W : p.i) \mapsto (\forall i : i \in W : q.i) \vee r}$$

◀

Figure 4.8: *Basic laws of* $\mapsto$

prog	P
read	$\{a, x\}$
write	$\{x\}$
init	true
assign	if $a < 2$ then $x := 1$
[]	if $a = 2$ then $x := x + 1$

prog	TikToe
read	$\{a, b\}$
write	$\{a\}$
init	true
assign	if $a = 0$ then $a := 1$
[]	if $a = 1$ then $a := 0$
[]	if $b \neq 0$ then $a := a + 1$

Figure 4.9: *The program P and* TikToe.

programs are frequently found in practice. In Chapter 2 we hypothesize that the parallel composition of write-disjoint programs satisfies a principle called Transparency Principle. It states that progress made through the writable part of a write-disjoint component P will be preserved by the composition, as long as the other write-disjoint components respect whatever assumptions P has on its non-writable part. The example with P and TikToe suggests that the principle is valid. It is valid, as we will see later, but not if we use $\mapsto$ to specify progress.

To illustrate the problem with $\mapsto$ consider the following program P', with the same read and write variables as P, and the same initial condition. However, the action if $a < 2$ then $x := 1$ in P will be broken in two. P' has the following assign section:

```
  if a = 0 then x := 1
[] if a = 1 then x := 1
[] if a = 2 then x := x + 1
```

The program P' also satisfies (4.5.1). However, if composed with TikToe the progress may fail if both programs choose a wrong order of execution. Consider the following scheduling of the actions in $P'[]$TikToe:

```
[ if a = 0 then a := 1 ; if a = 0 then x := 1 ; if b ≠ 0 then a := a + 1 ;
  if a = 1 then a := 0 ; if a = 1 then x := 1 ; if a = 2 then x := x + 1 ]*
```

which is fair, but if initially all a, b, and x are 0, then x will never become 1 in this execution sequence.

So, if we imagine P as a program which is still under development (so, we cannot look into its code), and if the specification of P states that it should satisfy $(b = 0) \wedge (a < 2) \mapsto (x = 1)$ and $\{a, b\} \not\subseteq \mathbf{w}P$, we cannot just say that composing it with TikToe will preserve $(b = 0) \wedge (a < 2) \mapsto (x = 1)$. The moral of the story is that we cannot generally conclude the $\mapsto$ properties of a composite program from the $\mapsto$ properties of its components, without further information about the interior of the components, or at least, information about how the components' properties are derived. There are however cases where it is possible. A sufficient condition was given by Singh in [Sin89]. This will be discussed in Section 4.7. But even the result by Sigh is not strong enough to derive the Transparency Principle. In this section, a new progress operator will be introduced, with which the principle is provable.

Let us define the following variant of ensures:

Definition 4.5.1 ensures *B_ENS*

$$J \ {}_P\vdash\ p \ \textbf{ensures}\ q \ = \ p,q \in \mathsf{Pred}.(\mathbf{w}P) \ \wedge \ ({}_P\vdash \circlearrowleft J) \wedge ({}_P\vdash J \wedge p \ \mathsf{ensures}\ q)$$

Note that **ensures** only describes progress through the writable part of a program. An additional parameter J is added, which is required to be stable in the program. Since the state of the non-writable part of the program cannot change, it can be specified within J.

A variant of leads-to called "reach", denoted by $\rightarrowtail$, can be defined as the smallest transitive and left-disjunctive closure of **ensures**. It follows that $\rightarrowtail$ can only specify progress made through the writable part of a program, but this should not be a hindrance as such is the only kind of progress a program can make.

Definition 4.5.2 REACH *REACH*

$$(\lambda p,q.\ J \ {}_P\vdash\ p \rightarrowtail q) \ = \ \mathsf{TDC}.(\lambda p,q.\ J \ {}_P\vdash\ p \ \textbf{ensures}\ q)$$

Alternatively, we can also define $\rightarrowtail$ as follows:

$$\begin{array}{c} J \ {}_P\vdash\ p \rightarrowtail q \\ = \\ ({}_P\vdash \circlearrowleft J) \ \wedge \ \mathsf{TDC}.(\lambda r,s.\,({}_P\vdash J \wedge r \ \mathsf{ensures}\ s) \wedge r,s \in \mathsf{Pred}.(\mathbf{w}P)).p.q \end{array} \tag{4.5.2}$$

It should now be obvious why we introduced TDC. The properties of TDC mentioned in Subsection 4.4.1 can easily be instantiated for $\rightarrowtail$. This operator is *not* equal to $\mapsto$, but let us postpone the details until Section 4.10. It is however easy to see that:

$$(\mathsf{true}\ {}_P\vdash\ p \rightarrowtail q) \Rightarrow ({}_P\vdash\ p \mapsto q) \tag{4.5.3}$$

By its definition, $\mathsf{true}\ {}_P\vdash\ p \ \textbf{ensures}\ q$ implies ${}_P\vdash\ p \ \mathsf{ensures}\ q$. Hence, by TDC MONOTONICITY$_{48}$, (4.5.3) follows.

As a notational convention: *if it is clear from the context which program P or which stable predicate J are meant, we often omit them from an expression.* For example we may write ${}_P\vdash\ p \rightarrowtail q$ or even simply $p \rightarrowtail q$ to mean $J \ {}_P\vdash\ p \rightarrowtail q$. Also, for laws we write, for example:

$$P,J: \ \frac{\ldots(p \ \mathsf{unless}\ q)\ldots}{r \rightarrowtail s} \ \text{ to abbreviate: } \ \frac{\ldots({}_P\vdash\ p \ \mathsf{unless}\ q)\ldots}{J \ {}_P\vdash\ r \rightarrowtail s}$$

Figure 4.11 displays a set of basic laws of $\rightarrowtail$ which have corresponding laws for $\mapsto$. The proofs of these properties follow the pattern of the related proofs for $\mapsto$ properties as found in [CM88]. Figure 4.12 displays some laws which have no $\mapsto$ counterpart. Some comment as to how the laws can be proven is also included. In particular, the $\rightarrowtail$ CONFINEMENT law states that an expression of the form $J \ {}_P\vdash\ p \rightarrowtail q$ is only valid if both

```
prog    Buffer
read    {in, inRdy, ack, buf, out}
write   {ack, buf, out}
init    ¬ack ∧ (buf = [])
assign
        if inRdy ∧ ¬ack ∧ (ℓ.buf < N) then buf, ack := buf ++[in], true
[]      if ¬inRdy ∧ ack then ack := false
[]      if buf ≠ [] then out, buf := hd.buf, tl.buf
```

Figure 4.10: *An N-placed buffer.*

p and q are confined by Pred.($\mathbf{w}P$). This confirms what is said before, namely that $\rightarrowtail$ describes only progress through the writable-part of a program. The $\rightarrowtail$ DISJUNCTION states the disjunctivity property of $\rightarrowtail$. It should be noted however, that $\rightarrowtail$ is *not* disjunctive in its J-argument. That is, $J_1 \vdash p \rightarrowtail q$ and $J_2 \vdash p \rightarrowtail q$ do not necessarily imply $J_1 \wedge J_2 \vdash p \rightarrowtail q$. There are good reasons for this, but let us postpone the discussion for a while.

As an example, consider the program Buffer displayed in Figure 4.10. It uses a buffer buf of size N. What it does is passing on the values in buf to out, first in, first out. Data are entered to buf via the input register in. The boolean variable inRdy is an input variable, which is expected to become true if a new datum becomes available. Buffer is ready to receive a new datum if there is a place in buf and if ack is false. When a new datum is entered to buf, it is acknowledged by setting ack true. A property of the program Buffer is that a new and un-acknowledged datum will eventually appear in out. Using $\mapsto$ this can be expressed as follows:

$$(\forall X :: {}_{\text{Buffer}}\vdash (\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \mapsto (\mathsf{out} = X)) \tag{4.5.4}$$

Using $\rightarrowtail$ the property can be expressed as follows:

$$(\forall X :: (\mathsf{in} = X) \wedge \mathsf{inRdy}\ {}_{\text{Buffer}}\vdash \neg\mathsf{ack} \rightarrowtail (\mathsf{out} = X)) \tag{4.5.5}$$

However, the following, which would be quite tempting to write due to its resemblance to (4.5.4):

$$(\forall X :: \mathsf{true}\ {}_{\text{Buffer}}\vdash (\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \rightarrowtail (\mathsf{out} = X)) \tag{4.5.6}$$

is *not* a valid expression because the argument "(in $= X$) $\wedge$ inRdy $\wedge$ $\neg$ack" is not a predicate confined by $\mathbf{w}$(Buffer).

The previously mentioned Transparency Law will be presented and proven in Section 4.7.

4.6 On the Substitution Law

Recall that we call a predicate p invariant in a program P if it holds throughout any computation of P. Consequently, the truth of p can be assumed in manipulating specifications of P, if by a 'specification' we mean a predicate over all possible executions of

Theorem 4.5.3 $\rightarrowtail$ INTRODUCTION *REACH_ENS_LIFT,REACH_IMP_LIFT*

$$P, J : \frac{p, q \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ (\circlearrowleft J) \quad [J \wedge p \Rightarrow q] \ \vee \ (J \wedge p \ \mathsf{ensures} \ q)}{p \rightarrowtail q}$$

Theorem 4.5.4 $\rightarrowtail$ INDUCTION *REACH_INDUCT1*

$$P, J : \frac{(\forall p, q :: (p \ \mathsf{ensures} \ q) \Rightarrow R.p.q) \quad \mathsf{Trans}.R \ \wedge \ \mathsf{Ldisj}.R}{(p \rightarrowtail q) \Rightarrow R.p.q}$$

Theorem 4.5.5 $\rightarrowtail$ TRANSITIVITY *REACH_TRANS*

$$P, J : \frac{(p \rightarrowtail q) \ \wedge \ (q \rightarrowtail r)}{p \rightarrowtail r}$$

Theorem 4.5.6 $\rightarrowtail$ DISJUNCTION *REACH_DISJ*

For all *non-empty* sets W:

$$P, J : \frac{(\forall i : i \in W : p.i \rightarrowtail q.i)}{(\exists i : i \in W : p.i) \rightarrowtail (\exists i : i \in W : q.i)}$$

Corollary 4.5.7 $\rightarrowtail$ REFLEXIVITY *REACH_REFL*

$$P, J : \frac{p \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ (\circlearrowleft J)}{p \rightarrowtail p}$$

Corollary 4.5.8 $\rightarrowtail$ CANCELLATION *REACH_CANCEL*

$$P, J : \frac{q \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ (p \rightarrowtail q \vee r) \ \wedge \ (r \rightarrowtail s)}{p \rightarrowtail q \vee s}$$

Theorem 4.5.9 $\rightarrowtail$ PSP *REACH_PSP*

$$P, J : \frac{r, s \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ (r \wedge J \ \mathsf{unless} \ s) \wedge \ (p \rightarrowtail q)}{p \wedge r \rightarrowtail (q \wedge r) \vee s}$$

Theorem 4.5.10 $\rightarrowtail$ COMPLETION *REACH_COMPLETION*

For all *finite* and *non-empty* sets W:

$$P, J : \frac{r \in \mathsf{Pred.}(\mathbf{w}P) \quad (\forall i : i \in W : q.i \wedge J \ \mathsf{unless} \ r) \ \wedge \ (\forall i : i \in W : p.i \rightarrowtail q.i \vee r)}{(\forall i : i \in W : p.i) \rightarrowtail (\forall i : i \in W : q.i) \vee r}$$

◀

Figure 4.11: *Basic properties of* $\rightarrowtail$ *which are analogous to those of* $\mapsto$

Theorem 4.5.11 $\rightarrowtail$ STABLE SHIFT *REACH_STABLE_SHIFT*

$$P: \frac{p_2 \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ (\circlearrowleft J) \ \wedge \ (J \wedge p_2 \vdash p_1 \rightarrowtail q)}{J \vdash p_1 \wedge p_2 \rightarrowtail q}$$

Can be proven using $\rightarrowtail$ INDUCTION.

Theorem 4.5.12 $\rightarrowtail$ STABLE STRENGTHENING *REACH_STAB_MONO*

$$P: \frac{(\circlearrowleft J_2) \ \wedge \ (J_1 \vdash p \rightarrowtail q)}{J_1 \wedge J_2 \vdash p \rightarrowtail q}$$

Can be proven using $\rightarrowtail$ INDUCTION.

Corollary 4.5.13 $\rightarrowtail$ STABLE BACKGROUND *REACH_IMP_STABLE*

$$P: \frac{J \vdash p \rightarrowtail q}{\circlearrowleft J}$$

Follows straightforwardly from the alternative definition of $\rightarrowtail$ (4.5.2).

Theorem 4.5.14 $\rightarrowtail$ CONFINEMENT *REACH_IMP_CONF*

$$P, J: \frac{p \rightarrowtail q}{p, q \in \mathsf{Pred.}(\mathbf{w}P)}$$

Can be proven using $\rightarrowtail$ INDUCTION.

Theorem 4.5.15 $\rightarrowtail$ SUBSTITUTION *REACH_SUBST*

$$P, J: \frac{\begin{array}{c} p, s \in \mathsf{Pred.}(\mathbf{w}P) \\ [J \wedge p \Rightarrow q] \ \wedge \ (q \rightarrowtail r) \ \wedge \ [J \wedge r \Rightarrow s] \end{array}}{p \rightarrowtail s}$$

Follows from $\rightarrowtail$ INTRODUCTION, STABLE BACKGROUND, TRANSITIVITY, and CONFINEMENT.

Figure 4.12: *More properties of* $\rightarrowtail$

P. This very natural principle is imposed in [CM88] as an axiom called the *Substitution Law*, and has the following form. Let $\mathfrak{R}$ be either unless, ensures, or $\mapsto$:

$$P: \frac{\begin{array}{c} \text{"}J \text{ is invariant"} \\ [J \Rightarrow (p = p')] \ \wedge \ \mathfrak{R}.p.q \ \wedge \ [J \Rightarrow (q = q')] \end{array}}{\mathfrak{R}.p'.q'} \qquad (4.6.1)$$

The law was a source of anxiety because it was found that the law makes the logic inconsistent [San91]. On the other hand, without the Substitution Law UNITY is incomplete relative to a certain operational semantics. In [San91] Sanders proposed an extension, from which the law can be derived instead of imposed as an axiom. The consistency of the logic was thus guaranteed.

In this section we will briefly discuss the Substitution Law and Sander's extension, and their relation to the UNITY logic plus the $\rightarrowtail$ operator as we have described so far.

As we have no Substitution Law, at least, not for unless, ensures, and $\mapsto$, one may ask what kind of incompleteness one may expect. Let us consider just the case of unless. The other two operators can be argued about in much the same way. Recall that in Section 4.3 we have carefully given the following operational interpretation for unless:

> *Intuitively,* ${}_P\vdash p$ unless q **implies** *that once p holds during an execution of P, it remains to hold at least until q holds.*

This deviates slightly from the traditional interpretation, for example as in [CM88], in which the "implies" above is replaced by "if and only if". Indeed, we use "implies" because we wish to avoid questions about incompleteness, until now. Now let us see what kind of problem we run into if we replace "implies" with "if and only if".

Let ${}_P\vdash p \,\mathfrak{U}\, q$ means "once p holds during an execution of P, it remains to hold at least until q holds". Now consider a program Lazy as follows:

prog	Lazy
read	$\{a, x\}$
write	$\{x\}$
init	$(a \neq 0) \wedge (x = 0)$
assign	if $a = 0$ then $x := x + 1$

In Lazy we have $(x = 0)$ unless $(a = 0)$. Because initially $a \neq 0$, the value of x will remain constant. So, $(x = 0)\ \mathfrak{U}$ false holds. However, $(x = 0)$ unless false cannot be derived without the Substitution Law:

$$
\begin{array}{ll}
 & (x = 0) \text{ unless } (a = 0) \\
= & \quad \{\ a \neq 0 \text{ is invariant, Substitution Law }\ \} \\
 & (x = 0) \text{ unless false}
\end{array}
$$

But ${}_{\text{Lazy}}\vdash (x = 0)$ unless false by the definition of unless cannot hold since it is equal to false. So obviously, unless is *not* equal to the interpretation $\mathfrak{U}$[7].

In [San91] Sanders introduced a variant unless which is equal to $\mathfrak{U}$ (and similarly for ensures and $\mapsto$). Let ${}_P\vdash \Box J$ means that J is invariant. Sanders defines unless_S as follows:

$$
{}_P\vdash p \text{ unless}_S\ q \ = \ (\exists J :\ {}_P\vdash \Box J : J \wedge p \text{ unless } q) \tag{4.6.2}
$$

For J, one can always choose the strongest invariant of P. This strongest invariant characterizes all states reachable by P, which is why the Substitution Law is derivable

[7] This is caused by the fact that unless is defined in terms of *all* states instead of those states which are actually reachable from the starting state(s).

from the definition above. The unless as defined above coincides with the interpretation $\mathfrak{U}$

To explicitly record on which invariant a property is based, Sanders generalized (4.6.2) by parameterizing unless_S with an invariant:

$$J \;{}_P\!\vdash\; p \;\mathsf{unless}_S\; q \;=\; ({}_P\!\vdash \Box J) \wedge ({}_P\!\vdash (J \wedge p) \;\mathsf{unless}\; q) \tag{4.6.3}$$

Curiously, Sanders fell into the same trap as Chandy and Misra did in [CM88] by claiming that the Substitution Law also applies to the parameterized unless_S. Consider again the program Lazy. Note that $x < 2$ is invariant. We derive now:

$$\begin{array}{ll}
 & \mathsf{true} \\
= & \quad \{\ (4.6.3),\ x<2 \text{ is invariant, unless } \textsc{Anti-reflexivity}_{46}\ \} \\
 & (x<2) \;{}_{\mathsf{Lazy}}\!\vdash\; (x<2) \;\mathsf{unless}_S\; (2 \le x) \\
= & \quad \{\ x<2 \text{ is invariant, Substitution Law}\ \} \\
 & (x<2) \;{}_{\mathsf{Lazy}}\!\vdash\; (x<2) \;\mathsf{unless}_S\; \mathsf{false} \\
\Rightarrow & \quad \{\ (4.6.3)\ \} \\
 & \;{}_{\mathsf{Lazy}}\!\vdash\; (x<2) \;\mathsf{unless}\; \mathsf{false} \\
= & \quad \{\ \text{definition of Lazy}\ \} \\
 & \mathsf{false}
\end{array}$$

The flaw is corrected by Prasetya in [Pra94] by requiring that J is not only an invariant, but also a 'strong' invariant. A strong invariant is an invariant which is also stable.

Definition 4.6.1 Strong Invariant *Inv*

$${}_P\!\vdash \boxed{\mathsf{s}} J \;=\; [\mathbf{ini}P \Rightarrow J] \wedge ({}_P\!\vdash \circlearrowright J)$$

Definition 4.6.2 Parameterized unless_S *UNL*

$$J \;{}_P\!\vdash\; p \;\mathsf{unless}_S\; q \;=\; ({}_P\!\vdash \boxed{\mathsf{s}} J) \wedge ({}_P\!\vdash\; J \wedge p \;\mathsf{unless}\; q \wedge J)$$

One can show that the following Substitution Law holds for the above definition of unless_S. The result extends to ensures and $\mapsto$. It has been mechanically verified and available as part of our UNITY package for the theorem prover HOL.

Theorem 4.6.3 unless Substitution *UNL_SUBST*

$$P, J :\ \frac{[J \Rightarrow (p = p')] \;\wedge\; (p \;\mathsf{unless}_S\; q) \;\wedge\; [J \Rightarrow (q = q')]}{p' \;\mathsf{unless}_S\; q'}$$

The reader may note that the $\rightarrowtail$ operator already fulfils a $\textsc{Substitution}_{57}$ law. Indeed, the operator has some flavor of Sanders' parameterized $\mapsto$, but there are some important differences. Sanders' invariant-parameterized $\mapsto$ is defined as follows:

$$J \;{}_P\!\vdash\; p \overset{S}{\mapsto} q \;=\; ({}_P\!\vdash \boxed{\mathsf{s}} J) \;\wedge\; ({}_P\!\vdash\; J \wedge p \mapsto q) \tag{4.6.4}$$

Theorem 4.7.1 unless COMPOSITIONALITY *UNLESS_PAR_i*

$$({}_P\vdash p \text{ unless } q) \ \wedge \ ({}_Q\vdash p \text{ unless } q) \quad = \quad ({}_{P\|Q}\vdash p \text{ unless } q)$$

Corollary 4.7.2 $\circlearrowright$ COMPOSITIONALITY *STABLE_PAR_i*

$$({}_P\vdash \circlearrowright J) \ \wedge \ ({}_Q\vdash \circlearrowright J) \quad = \quad ({}_{P\|Q}\vdash \circlearrowright J)$$

Follows from unless COMPOSITIONALITY and the definition of $\circlearrowright$.

Corollary 4.7.3 $\boxed{\text{S}}$ COMPOSITIONALITY *Inv_PAR*

$$\frac{({}_P\vdash \boxed{\text{S}}\, J) \ \wedge \ ({}_Q\vdash \boxed{\text{S}}\, J)}{{}_{P\|Q}\vdash \boxed{\text{S}}\, J}$$

Follows from $\circlearrowright$ COMPOSITIONALITY and the definition of $\boxed{\text{S}}$.

Theorem 4.7.4 ensures COMPOSITIONALITY *ENSURES_PAR*

$$\frac{({}_P\vdash p \text{ ensures } q) \ \wedge \ ({}_Q\vdash p \text{ unless } q)}{{}_{P\|Q}\vdash p \text{ ensures } q}$$

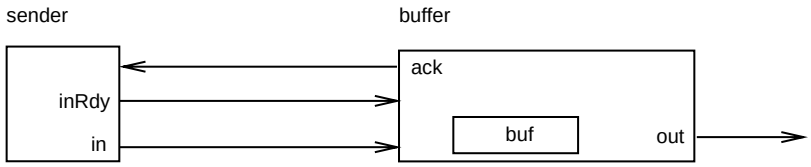
Figure 4.13: *Compositionality of safety properties and of* ensures.

In $\rightarrowtail$, the J is only required to be stable, which apparently is enough to have the SUBSTITUTION law. This is a useful generalization because when combining parallel programs, an invariant is easier to be destroyed than a stable predicate. Also, since $\overset{\text{S}}{\mapsto}$ is based on $\mapsto$, which is not confined by $\mathbf{w}P$, it will have the same problem as $\mapsto$ when it comes to parallel composition, especially the parallel composition of write-disjoint programs. If one wishes to combine the strength of Sanders' definition and that of $\rightarrowtail$, one will have to parameterize with both stable predicates and invariants.

4.7 Parallel Composition

As has been motivated in Chapter 2, composition laws are useful as they enable us to decompose a global specification of a program into local specifications of the program's components. Not only that the original problem, which may contain complicated inter-component dependencies, is thereby broken into more manageable pieces, but also each component can subsequently be developed in isolation. In this section an overview of various parallel composition laws that we have verified will be given.

The compositionality of safety properties follows a very simple principle: the safety of a composite follows from the safety of its components; the laws are given in Figure 4.13. As for the compositionality of progress properties, only the compositionality of ensures was known in the first place. A parallel component may write to a common variable, and thereby affecting, or even, destroying a progress property of another com-



prog Sender
read {ack, in, inRdy}
write {in, inRdy}
init ¬inRdy
assign if ¬inRdy ∧ ¬ack then in, inRdy := "generate a new datum", true
[] if inRdy ∧ ack then inRdy := false

prog Buffer
read {in, inRdy, ack, buf, out}
write {ack, buf, out}
init ¬ack ∧ (buf = [])
assign if inRdy ∧ ¬ack ∧ (ℓ.buf $< N$) then buf, ack := buf ++[in], true
[] if ¬inRdy ∧ ack then ack := false
[] if buf ≠ [] then out, buf := hd.buf, tl.buf

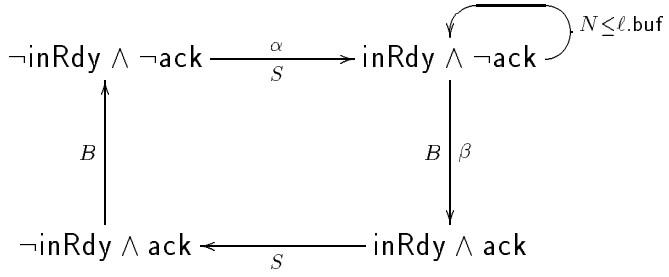
Figure 4.14: *A sender and an N-placed buffer.*

ponent. The phenomenon was not well understood and it was thought that except in the most trivial cases, no useful result can be obtained for progress properties expressed by $\mapsto$. A step forward is made by Singh [Sin89]. Although no claim is made on the strength of Singh's results, we believe that it is fairly strong. The results will be discussed in the sequel. However, instead of $\mapsto$, $\rightarrowtail$ will be used to express progress. The laws will look slightly different but the idea remains the same.

Stronger results can be obtained for parallel composition of programs which share no common write variables, but $\rightarrowtail$ is really required here. Perhaps, it should also be noted that as we attempted to verify Singh's results it was discovered that his main theorem is flawed. The flaw may look trivial and can be easily removed, but a considerably more sophisticated proof is required. This should illustrate a bit as how poor the issue was —and probably still is— understood.

4.7.1 General Progress Composition

Consider again the program Buffer in Figure 4.10 and a new program Sender, both displayed in Figure 4.14. Both programs are intended to be put together in parallel. The picture in Figure 4.14 may be helpful. The program Sender generates data and sends it through in to the program Buffer. The latter puts the data in an N-placed buffer buf and meanwhile, it also passes the data to out, first in, first out. The reader may notice that the synchronization between Sender and Buffer is a hand-shake protocol



Note: S abbreviates Sender and B Buffer. α produces a new datum and β put the datum in buf, if there is a place for it.

◀

Figure 4.15: *A 4-phase hand-shake protocol between* Sender *and* Buffer.

with the following phases:

$$\begin{array}{lcccc} \text{phase } 0 & : & \neg\mathsf{inRdy} & \wedge & \neg\mathsf{ack} \\ \text{phase } 1 & : & \mathsf{inRdy} & \wedge & \neg\mathsf{ack} \\ \text{phase } 2 & : & \mathsf{inRdy} & \wedge & \mathsf{ack} \\ \text{phase } 3 & : & \neg\mathsf{inRdy} & \wedge & \mathsf{ack} \end{array}$$

A new datum can only be generated in phase 0, which subsequently brings the system to phase 1. A new datum can only enter buf in phase 1 (and if buf has a place free), which subsequently brings the system to phase 2. Then, Sender will flip its inRdy, bringing the system into phase 3, and Buffer its ack, reverting the system to phase 0. The transition graph in Figure 4.15 may be helpful.

Let SB abbreviate Sender ‖ Buffer. A fundamental property which we claim that SB has is the following:

$$(\forall X :: \mathsf{true} \;{}_{\mathsf{SB}}\!\vdash (\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \rightarrowtail (\mathsf{out} = X)) \tag{4.7.1}$$

stating that a new, un-acknowledged datum will eventually reach out. This progress property can be proved directly from the code of SB. However, let us now see how it can be derived from the properties of Sender and Buffer. Let us first do some calculation, starting from (4.7.1):

$$\begin{array}{ll} & (\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \rightarrowtail (\mathsf{out} = X) \\ \Leftarrow & \quad \{ \rightarrowtail \textsc{Transitivity}_{56} \} \\ & ((\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \rightarrowtail X \in \mathsf{buf}) \;\wedge\; (X \in \mathsf{buf} \rightarrowtail (\mathsf{out} = X)) \end{array}$$

Now, by applying $\rightarrowtail$ PSP$_{56}$ to the first conjunct using the following instantiation of the law:

$$\begin{array}{lcl} p & \leftarrow & (\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \\ q & \leftarrow & X \in \mathsf{buf} \vee (\mathsf{in} \neq X) \vee \neg\mathsf{inRdy} \vee \mathsf{ack} \\ r & = & p \\ s & \leftarrow & X \in \mathsf{buf} \\ J & \leftarrow & \mathsf{true} \end{array}$$

we can refine the last specification to the following:

$$\quad {}_{SB}\vdash \quad (\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \text{ unless } X \in \mathsf{buf} \tag{4.7.2}$$

$$\mathsf{true} \quad {}_{SB}\vdash \quad (\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \rightarrowtail X \in \mathsf{buf} \vee (\mathsf{in} \neq X) \vee \neg\mathsf{inRdy} \vee \mathsf{ack} \tag{4.7.3}$$

$$\mathsf{true} \quad {}_{SB}\vdash \quad X \in \mathsf{buf} \rightarrowtail (\mathsf{out} = X) \tag{4.7.4}$$

(4.7.2) states that while in phase 1 the system (SB) can only either remain there, or put the value of in into buf. We leave it to the reader to figure it out why SB satisfies this. (4.7.4) should be provable from the fact that this is a 'local' progress of the Buffer. We will return to this later. Let us for now concentrate on (4.7.3). Something that might help to prove this is the following property of Buffer:

$$(\mathsf{in} = X) \wedge \mathsf{inRdy} \ {}_{\mathsf{Buffer}}\vdash \mathsf{true} \rightarrowtail X \in \mathsf{buf} \vee \mathsf{ack} \tag{4.7.5}$$

But how can this be exploited to prove (4.7.3)? Note that the only way Sender can influence Buffer is through the variables in and inRdy. However, Sender *cannot* modify any of those if inRdy $\wedge$ ¬ack holds. Consequently, if in addition in $= X$ then either by (4.7.5) Buffer will make its progress to $X \in$ buf $\vee$ ack, or either program does something to inRdy, ack, or in which it invalidates either inRdy $\wedge$ ¬ack or in $= X$. In other words, (4.7.3) is implied!

The principle used to conclude (4.7.3) from(4.7.5) is an instantiation of a composition law known as Singh's Law [Sin89]. Before it can be presented, first we need some definitions.

Definition 4.7.5 !? *DVa*

$$Q!?P \ = \ \mathbf{w}Q \cap \mathbf{r}P$$

So, $Q!?P$ denotes the variables through which Q can influence P. For example, Sender!?Buffer is {in, inRdy}. In practice people often use the term 'shared variables'. This term is ambiguous because it is not clear whether it is meant variables which are read, or written in common, or some other combination.

Let V be a set of variables. Let ${}_{Q}\vdash p \text{ unless}_V\ q$ roughly mean that under condition p, Q cannot alter the variables in V without establishing q:

Definition 4.7.6 unless_V

$${}_{Q}\vdash p \text{ unless}_V\ q \ = \ (\forall X :: \ {}_{Q}\vdash p \wedge (\forall v : v \in V : v = X.v) \text{ unless } q)$$

Note: the dummy X has the type Var→Val. By omitting some overloading we can also write the formula above as:

$${}_{Q}\vdash p \text{ unless}_V\ q \ = \ (\forall X :: \ {}_{Q}\vdash p \wedge (\lambda s.\ s \restriction V = X \restriction V) \text{ unless } q)$$

In particular, ${}_Q\vdash p$ unless$_{Q!?P}$ q means that under condition p, Q *cannot* influence P without establishing q. For example, ${}_Q\vdash p$ unless$_{Q!?P}$ false means that Q cannot influence P as long as p holds; ${}_Q\vdash$ true unless$_{Q!?P}$ q means that Q always marks its interference to P by establishing q. Recall that the program Sender cannot influence Buffer as long as inRdy $\wedge$ $\neg$ack holds. So, Sender satisfies:

$$_{\text{Sender}}\vdash \text{inRdy} \wedge \neg\text{ack unless}_{\text{Sender!?Buffer}} \text{ false} \tag{4.7.6}$$

There are two lemmas that we are going to use later. The first states that if p is a predicate confined by $\mathbf{r}P$, then Q cannot destroy p without changing a variable in $Q!?P$:

Lemma 4.7.7 *CONF_SAFE*

$$\frac{p \in \text{Pred.}(\mathbf{r}P)}{{}_Q\vdash p \wedge (\lambda s.\ s\restriction U = X\restriction U) \text{ unless } (\lambda s.\ s\restriction U \neq X\restriction U)} \quad \text{where } U = Q!?P$$

To prove the above lemma, the theory developed in Chapter 3 will now be brought into play.

Proof:

$$\begin{array}{ll} & p \wedge (\lambda s.\ s\restriction U = X\restriction U) \text{ unless } (\lambda s.\ s\restriction U \neq X\restriction U) \\ = & \quad \{ \text{ definition unless, predicate calculus } \} \\ & (\forall a : a \in \mathbf{a}Q : \{p \wedge (\lambda s.\ s\restriction U = X\restriction U)\}\ a\ \{(\lambda s.\ s\restriction U = X\restriction U) \Rightarrow p\}) \\ \Leftarrow & \quad \{\ U = \mathbf{w}Q \cap \mathbf{r}P, \text{ Theorem } 3.4.5_{33}\ \} \\ & p \in \text{Pred.}(\mathbf{r}P)\ \wedge\ (\forall a : a \in \mathbf{a}Q : (\mathbf{w}Q)^{\mathsf{c}} \nleftarrow a) \\ \Leftarrow & \quad \{ \text{ definition Unity } \} \\ & p \in \text{Pred.}(\mathbf{r}P)\ \wedge\ \text{Unity.}Q \end{array}$$

▲

Lemma 4.7.8

$$\frac{p \in \text{Pred.}(\mathbf{r}P)\ \wedge\ ({}_Q\vdash r \text{ unless}_{Q!?P}\ s)}{{}_Q\vdash p \wedge r \text{ unless } s}$$

Proof:

Let $U = Q!?P$. We want to prove $p \wedge r$ unless s. Using unless Simple Disjunction$_{46}$ it suffices to show that for all X:

$$p \wedge r \wedge (\lambda s.\ s\restriction U = X\restriction U) \text{ unless } s$$

Using unless Conjunction$_{46}$ and Post Weakening$_{46}$ it suffices to show:

$$\begin{array}{lll} r \wedge (\lambda s.\ s\restriction U = X\restriction U) & \text{unless} & s \\ p \wedge (\lambda s.\ s\restriction U = X\restriction U) & \text{unless} & (\lambda s.\ s\restriction U \neq X\restriction U) \end{array}$$

The first follows from ${}_Q\vdash r$ unless$_U$ s. The second follows from Lemma 4.7.7.
▲

Now here is the Singh law. If P can make progress $p \rightarrowtail q$, and under condition r, Q cannot influence P without establishing s, then in the composition $P[\!]Q$ starting from p and r, either P makes its progress to q, or it does something that invalidates r, or Q writes something to P, in which case s will hold. The law is formulated below. A more general version is also provided.

Theorem 4.7.9 SINGH LAW *REACH_SINGH*

$$\frac{r, s \in \mathsf{Pred.w}(P[\!]Q) \ \wedge \ ({}_Q\vdash \circlearrowleft J) \ \wedge \ ({}_Q\vdash J \wedge r \ \mathsf{unless}_{Q!?P} \ s) \ \wedge \ (J \ {}_P\vdash p \rightarrowtail q)}{J \ {}_{P[\!]Q}\vdash p \wedge r \rightarrowtail q \vee \neg r \vee s}$$

Theorem 4.7.10 (GENERAL) SINGH LAW *REACH_SINGH_g*

$$\frac{\begin{array}{c} r, s \in \mathsf{Pred.w}(P[\!]Q) \ \wedge \ p_1 \in \mathsf{Pred.}(\mathbf{w}P \cup (Q!?P)) \\ ({}_{P[\!]Q}\vdash \circlearrowleft J) \ \wedge \ ({}_Q\vdash J \wedge r \ \mathsf{unless}_{Q!?P} \ s) \ \wedge \ (J \wedge p_1 \ {}_P\vdash p_2 \rightarrowtail q) \end{array}}{J \ {}_{P[\!]Q}\vdash p_1 \wedge p_2 \wedge r \rightarrowtail q \vee \neg p_1 \vee \neg r \vee s}$$

Only the proof of the GENERAL SINGH LAW will be presented.

Proof:

By applying $\rightarrowtail$ INDUCTION$_{56}$, it suffices to show that $\mathfrak{R} = (\lambda a, b.\ J \ {}_{P[\!]Q}\vdash p_1 \wedge a \wedge r \rightarrowtail b \vee \neg p_1 \vee \neg r \vee s)$ is transitive, left-disjunctive, and includes $\mathfrak{E} = (\lambda a, b.\ J \wedge p_1 \ {}_P\vdash a \ \mathsf{ensures} \ b)$. To show that $\mathfrak{R}$ is transitive is easy:

$$\begin{array}{ll} & (p_1 \wedge a \wedge r \rightarrowtail b \vee \neg p_1 \vee \neg r \vee s) \ \wedge \ (p_1 \wedge b \wedge r \rightarrowtail c \vee \neg p_1 \vee \neg r \vee s) \\ = & \quad \{ \text{ predicate calculus } \} \\ & (p_1 \wedge a \wedge r \rightarrowtail (p_1 \wedge b \wedge \neg r) \vee \neg p_1 \vee r \vee s) \ \wedge \ (p_1 \wedge b \wedge r \rightarrowtail c \vee \neg p_1 \vee \neg r \vee s) \\ \Rightarrow & \quad \{ \rightarrowtail \text{CANCELLATION}_{56} \} \\ & p_1 \wedge a \wedge r \rightarrowtail c \vee \neg p_1 \vee \neg r \vee s \end{array}$$

The left-disjunctivity of $\mathfrak{R}$ follows directly from $\rightarrowtail$ DISJUNCTION$_{56}$. It remains to show that $\mathfrak{R}$ includes $\mathfrak{E}$.

By applying $\rightarrowtail$ INTRODUCTION$_{56}$ and ensures COMPOSITIONALITY$_{60}$, $\mathfrak{R}.a.b$ is implied by:

i. $(p_1 \wedge a \wedge r) \ \in \ \mathsf{Pred.}(\mathbf{w}P[\!]Q)$

ii. $(b \vee \neg p_1 \vee \neg r \vee s) \ \in \ \mathsf{Pred.}(\mathbf{w}P[\!]Q)$

iii. ${}_{P[\!]Q}\vdash \ \circlearrowleft J$

iv. ${}_P\vdash \ J \wedge p_1 \wedge a \wedge r \ \mathsf{ensures} \ b \vee \neg p_1 \vee \neg r \vee s$

v. ${}_Q\vdash \ J \wedge p_1 \wedge a \wedge r \ \mathsf{unless} \ b \vee \neg p_1 \vee \neg r \vee s$

The first two are easy and left to the reader. ***iii*** appears in the assumption. Using ensures POST-WEAKENING$_{47}$, ***iv*** is implied by ${}_P\vdash J \wedge p_1 \wedge a \wedge r \ \mathsf{ensures} \ b$, which follows from $\mathfrak{E}.a.b$. For ***v*** we derive:

$J \wedge p_1 \wedge a \wedge r$ unless $b \vee \neg p_1 \vee \neg r \vee s$

$\Leftarrow$ { unless POST-WEAKENING$_{46}$ }

$J \wedge p_1 \wedge a \wedge r$ unless s

$\Leftarrow$ { $_{P[\!]Q}\vdash \circlearrowleft J$ implies $_{Q}\vdash \circlearrowleft J$, unless SIMPLE CONJUNCTION$_{46}$ }

$p_1 \wedge a \wedge r$ unless s

$\Leftarrow$ { Lemma 4.7.8 }

$(p_1 \wedge a) \in \mathsf{Pred.}(\mathbf{r}P) \ \wedge \ r$ unless$_{Q!?P}$ s

$\Leftarrow$ { predicate confinement distributes over $\wedge$; assumptions }

$p_1 \in \mathsf{Pred.}(\mathbf{r}P) \ \wedge \ a \in \mathsf{Pred.}(\mathbf{r}P)$

$\Leftarrow$ { confinement is monotonic with respect to $\subseteq$; definition $\mathfrak{E}$ }

$p_1 \in \mathsf{Pred.}(\mathbf{w}P \cup (Q!?P)) \wedge \mathfrak{E}.a.b$

▲

Let us now try to apply the law to our example with Sender and Buffer. Recall that we want to obtain the specification (4.7.3) from (4.7.5). The first is re-displayed below:

$$\mathsf{true}\ {}_{SB}\vdash (\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \rightarrowtail X \in \mathsf{buf} \vee (\mathsf{in} \neq X) \vee \neg\mathsf{inRdy} \vee \mathsf{ack}$$

By applying the GENERAL SINGH LAW, we can refine the above to the following:

i. $(\mathsf{inRdy} \wedge \neg\mathsf{ack}) \ \in \ \mathsf{Pred.}(\mathbf{w}(\mathsf{Sender}[\!]\mathsf{Buffer}))$

ii. $\mathsf{false} \ \in \ \mathsf{Pred.}(\mathbf{w}(\mathsf{Sender}[\!]\mathsf{Buffer}))$

iii. $(\mathsf{in} = X) \wedge \mathsf{inRdy}) \ \in \ \mathsf{Pred.}(\mathbf{w}(\mathsf{Buffer}) \cup \mathsf{Sender!?Buffer})$

iv. $(\mathsf{in} = X) \wedge \mathsf{inRdy}\ {}_{\mathsf{Buffer}}\vdash \ \mathsf{true} \rightarrowtail X \in \mathsf{buf} \vee \mathsf{ack}$

v. $\ {}_{\mathsf{Sender}}\vdash \ \mathsf{inRdy} \wedge \neg\mathsf{ack}$ unless$_{\mathsf{Sender!?Buffer}}$ false

The first three are trivial. ***iv*** is (4.7.5). ***v*** is (4.7.6), stating that Sender cannot influence Buffer as long as inRdy and $\neg$ack hold. It is not too difficult to conclude from its code that Sender has this property.

The Singh Law, although it formulates a very intuitive idea, looks complicated. The law reduces a progress specification of a program to a progress specification of one of its component, two safety specifications, and some 'type' restrictions on the predicates that occur in the specifications. At first sight, applying the law seems only to generate more proof obligations and one may therefore question the merit of using the law. However, recall that a $\rightarrowtail$ (or $\mapsto$) progress property is proved by composing a number of ensures properties. Without the Singh Law, or some other parallel composition law, all these ensures properties will have to be verified with respect to the whole program. With the Singh Law they only have to be verified with respect to a component program. If the number of ensures properties and the average size of the component programs are sufficiently large, then applying the Singh Law will become more economical.

In some cases, the law can be simplified. The following corollaries show some of these cases. In the linear temporal logic there is a relation called until. A program P satisfies $\ {}_{P}\vdash p$ until q if whenever p holds during an execution of P, afterwards q will

eventually hold but in the meantime p will continue to hold until q holds. This sounds very much like **ensures**, but **until** is actually larger than **ensures**. Roughly speaking, **until** is equal to $\mapsto \cap$ **unless** . Consequently, either $p \vdash \textbf{true} \rightarrowtail q$ or $(\textbf{true} \vdash p \rightarrowtail q) \wedge (\vdash p \textbf{ unless } q)$ imply p **until** q. If either holds in P, and under condition p, Q cannot influence P without establishing q then one can conclude that $p \rightarrowtail q$ must hold in the composition $P[]Q$. This is what the following two corollaries state.

Corollary 4.7.11 **until** COMPOSITIONALITY *UNTIL_COMPO1*

$$\frac{(\ {}_Q\!\vdash \circlearrowleft J) \ \wedge\ ({}_Q\!\vdash J \wedge p \textbf{ unless}_{P?!Q}\ q) \ \wedge\ ({}_P\!\vdash J \wedge p \textbf{ unless } q) \ \wedge\ (J\ {}_P\!\vdash p \rightarrowtail q)}{J\ {}_{P[]Q}\!\vdash p \rightarrowtail q}$$

Corollary 4.7.12 **until** COMPOSITIONALITY *UNTIL_COMPO2*

$$\frac{\begin{array}{c} p \in \textbf{Pred.}(\mathbf{w}P \cup (P?!Q)) \ \wedge\ ({}_{P[]Q}\!\vdash \circlearrowleft J) \\ ({}_Q\!\vdash J \wedge p \textbf{ unless}_{P?!Q}\ q) \ \wedge\ (J \wedge p\ {}_P\!\vdash \textbf{true} \rightarrowtail q) \end{array}}{J\ {}_{P[]Q}\!\vdash p \rightarrowtail q}$$

4.7.2 Exploiting Fix Point

A state s is called a fix point of a program P, if s remains unchanged under the execution of any action in P. The notion can be lifted to the predicate level. A predicate p is called a fix point of P, denoted by $p \in \textbf{Fp}.P$, if all states $s \in p$ are fix points.

Definition 4.7.13 FIX POINT *FPp_DEF*

$$p \in \textbf{Fp}.P \ =\ (\forall a, s : a \in \mathbf{a}P : p.s \wedge a.s.t \Rightarrow (s = t))$$

Note that the definition above may not match its intended interpretation if P contains an action a which is not always-enabled (that is, for some begin state s, a may not be able to make any transition). However, in the case of UNITY programs, their actions are assumed to be always-enabled.

Fix points can be useful in parallel composition. If a program P has reached a fix point, then certainly it cannot influence any other program Q. If Q cannot throw P from its fix points space, then any progress in Q will be preserved in the composition $P[]Q$. In fact, this principle is a corollary of the Singh law. Before its formulation is presented here are some basic properties of fix points. Fix points are anti-monotonic with respect to $\Rightarrow$. In addition, if a predicate q is a fix-point of P, then for any p, $p \wedge q$ is stable in P.

Theorem 4.7.14 Fp MONOTONICITY *FPp_MONO*

$$\frac{[p \Rightarrow q] \ \wedge \ q \in \mathsf{Fp}.P}{p \in \mathsf{Fp}.P}$$

Theorem 4.7.15 Fp STABILITY *FPp_IMP_STABLE*

$$\frac{q \in \mathsf{Fp}.P}{(p \wedge q) \in \mathsf{Fp}.P}$$

◀

We now present the composition law using fix points we mentioned before[8]:

Theorem 4.7.16 *REACH_COMPO_BY_FPp*

$$\frac{(\ {}_P\vdash \circlearrowleft (J_1 \wedge J_2)) \ \wedge \ J_2 \in \mathsf{Fp}.Q \ \wedge \ (J_1 \ {}_P\vdash p \rightarrowtail q)}{J_1 \wedge J_2 \ {}_{P[\!]Q}\vdash p \rightarrowtail q}$$

▶

Proof:

$$\begin{array}{ll} & J_2 \in \mathsf{Fp}.Q \ \wedge \ (J_1 \ {}_P\vdash p \rightarrowtail q) \\ \Rightarrow & \quad \{ \text{ Theorem 4.7.15; definition of } \mathsf{unless}_V \ \} \\ & (\ {}_Q\vdash J_1 \wedge J_2 \wedge \mathsf{true} \ \mathsf{unless}_{Q!?P} \ \mathsf{false}) \ \wedge \ (J_1 \ {}_P\vdash p \rightarrowtail q) \\ \Rightarrow & \quad \{ \ {}_P\vdash \circlearrowleft (J_1 \wedge J_2), \ \circlearrowleft \text{CONJUNCTION}_{47} \ \} \\ & (\ {}_Q\vdash J_1 \wedge J_2 \wedge \mathsf{true} \ \mathsf{unless}_{Q!?P} \ \mathsf{false}) \ \wedge \ (J_1 \wedge J_2 \ {}_P\vdash p \rightarrowtail q) \\ \Rightarrow & \quad \{ \text{ SINGH LAW}_{65} \ \} \\ & J_1 \wedge J_2 \ {}_{P[\!]Q}\vdash p \rightarrowtail q \end{array}$$

▲

4.7.3 Why the Original Version of Singh Law is Flawed

Earlier it was mentioned that the original version of the Singh law is flawed. Let us now take a look at this and see what we can learn from it. However, if the reader wishes, he can skip this entire subsection.

A simple form of the original Singh Law [Sin89] looks as follow:

$$\frac{(\ {}_Q\vdash \mathsf{true} \ \mathsf{unless}_{Q!?P} \ s) \ \wedge \ (\ {}_P\vdash p \mapsto q)}{{}_{P[\!]Q}\vdash p \mapsto q \vee s} \qquad (4.7.7)$$

stating that if Q cannot influence P without establishing s, then any progress made by P will be preserved in the composition $P[\!]Q$, or Q interferes and establishes s.

Note that *there is no restriction on p, q and s*. They may be any predicates! Now consider the following programs:

[8] The law was given first by Singh in [Sin89].

prog	P	prog	Q
read	$\{x\}$	read	$\{a\}$
write	$\{x\}$	write	$\{a\}$
init	true	init	true
assign	$x := x + 1$	assign	$a := \neg a$

Since the programs share no variable then obviously Q cannot influence P ($Q!?P = \emptyset$), and hence it satisfies:

$$ {}_{Q}\vdash \mathsf{true\ unless}_{Q!?P}\ \mathsf{false} \tag{4.7.8}$$

A valid property of P is ${}_{P}\vdash a \wedge (x = 0) \mapsto a \wedge (x = 1)$. Using (4.7.8) and Singh law (4.7.7) we conclude the progress also holds in $P[\!]Q$. But this cannot of course be true.

As no restriction is put on p and q in (4.7.7), the progress $p \mapsto q$ may actually refer to some internal variable of Q, and this is what causes the problem above. Let us now put a restriction on them and see how we can prove the law:

$$\frac{p, q \in \mathsf{Pred.}(\mathbf{r}P)\ \wedge\ ({}_{Q}\vdash \mathsf{true\ unless}_{Q!?P}\ s)\ \wedge\ ({}_{P}\vdash p \mapsto q)}{{}_{P[\!]Q}\vdash p \mapsto q \vee s} \tag{4.7.9}$$

To prove the above we do not have many options but to resort to $\mapsto$ INDUCTION$_{52}$. So, assuming ${}_{Q}\vdash \mathsf{true\ unless}_{Q!?P}\ r$, prove that $\mathfrak{R} = (\lambda p, q.\ p, q \in \mathsf{Pred.}(\mathbf{r}P) \Rightarrow ({}_{P[\!]Q}\vdash p \mapsto q \vee s))$ is transitive, left-disjunctive, and includes ensures. But we have now a new problem. The transitivity (and left-disjunctivity) of $\mathfrak{R}$ cannot be proved:

$$p, q \in \mathsf{Pred.}(\mathbf{r}P) \Rightarrow ({}_{P[\!]Q}\vdash p \mapsto q \vee s))$$
$$q, r \in \mathsf{Pred.}(\mathbf{r}P) \Rightarrow ({}_{P[\!]Q}\vdash q \mapsto r \vee s))$$

are not sufficient to prove

$$p, r \in \mathsf{Pred.}(\mathbf{r}P) \Rightarrow ({}_{P[\!]Q}\vdash p \mapsto r \vee s))$$

So, let us instead prove a slightly different law:

Theorem 4.7.17 SIMPLE SINGH LAW FOR $\mapsto$

$$\frac{({}_{Q}\vdash \mathsf{true\ unless}_{Q!?P}\ s)\ \wedge\ ({}_{P}\vdash p \mapsto q)}{{}_{P[\!]Q}\vdash (p \restriction \mathbf{r}P) \mapsto (q \restriction \mathbf{r}P) \vee s}$$

Note that if $p, q \in \mathsf{Pred.}(\mathbf{r}P)$, hence in other words $(p = p \restriction \mathbf{r}P)\ \wedge\ (q = q \restriction \mathbf{r}P)$, the above implies (4.7.9).

Proof:

Assuming ${}_{Q}\vdash \mathsf{true\ unless}_{Q!?P}\ s$, the law above will be proved using $\mapsto$ INDUCTION$_{52}$. So, it will be showed that $\mathfrak{R} = (\lambda p, q.\ {}_{P[\!]Q}\vdash (p \restriction \mathbf{r}P) \mapsto (q \restriction \mathbf{r}P) \vee s)$ is transitive, left-disjunctive, and includes $(\lambda p, q.\ {}_{P}\vdash p\ \mathsf{ensures}\ q)$. For the transitivity:

$$((p \restriction \mathbf{r}P) \mapsto (q \restriction \mathbf{r}P) \vee s)\ \wedge\ ((q \restriction \mathbf{r}P) \mapsto (r \restriction \mathbf{r}P) \vee s)$$

$$
\begin{array}{ll}
\Rightarrow & \{ \mapsto \textsc{Cancellation}_{52} \} \\
& (p \restriction \mathbf{r}P) \mapsto (r \restriction \mathbf{r}P) \vee s
\end{array}
$$

The left-disjunctivity of $\mathfrak{R}$ follows directly from $\mapsto$ $\textsc{Disjunction}_{52}$ and the fact that the confinement of predicates distributes over $\vee$.

As for the inclusion of ensures, assume ${}_P\vdash p$ ensures q. We derive:

$$
\begin{array}{ll}
& {}_{P\|Q}\vdash (p \restriction \mathbf{r}P) \mapsto (q \restriction \mathbf{r}P) \vee s \\
\Leftarrow & \{ \mapsto \textsc{Introduction}_{52} \} \\
& {}_{P\|Q}\vdash (p \restriction \mathbf{r}P) \text{ ensures } (q \restriction \mathbf{r}P) \vee s \\
\Leftarrow & \{ \text{ensures } \textsc{Compositionality}_{60} \} \\
& ({}_P\vdash (p \restriction \mathbf{r}P) \text{ ensures } (q \restriction \mathbf{r}P) \vee s) \ \wedge \ ({}_Q\vdash (p \restriction \mathbf{r}P) \text{ unless } (q \restriction \mathbf{r}P) \vee s) \\
\Leftarrow & \{ \text{ensures } \textsc{Post-weakening}_{47}; \text{ Corollary } 3.4.3_{31} \} \\
& (\forall a : a \in \mathbf{a}P : (\mathbf{r}P)^{\mathsf{c}} \twoheadrightarrow a) \ \wedge \ ({}_P\vdash p \text{ ensures } q) \ \wedge \ ({}_Q\vdash (p \restriction \mathbf{r}P) \text{ unless } (q \restriction \mathbf{r}P) \vee s) \\
\Leftarrow & \{ \text{assumption, definition } \mathsf{Unity}_{40} \} \\
& \mathsf{Unity}.P \ \wedge \ ({}_Q\vdash (p \restriction \mathbf{r}P) \text{ unless } (q \restriction \mathbf{r}P) \vee s) \\
\Leftarrow & \{ \text{unless } \textsc{Post-weakening}_{46} \text{ and Lemma } 4.7.8_{64} \} \\
& \mathsf{Unity}.P \ \wedge \ p \restriction \mathbf{r}P \in \mathsf{Pred}.(\mathbf{r}P) \ \wedge \ ({}_Q\vdash \text{true unless}_{Q!?P} \ s) \\
= & \{ P \text{ is a UNITY program, } p \restriction V \text{ is always confined by } \mathsf{Pred}.V \} \\
& {}_Q\vdash \text{true unless}_{Q!?P} \ s
\end{array}
$$

▲

4.7.4 A Short Overview

At this point, the reader may begin to lose track as to where we are aiming at. Just to remind him: we have introduced the logic UNITY and discussed how to represent a UNITY program. We have presented the standard UNITY operators, used to express the behaviors of a program, and discussed their shortcomings. We gave a special attention to progress properties and extended UNITY with a new progress operator which we claim to have a nice compositional property —this will be made clear in the next section. In the mean time, many calculational laws concerning the extended logic have been presented. Some of the laws will be used later, but some will not. In general though, the reader will likely find those laws to be useful for designing his own programs.

As said, the next section will present more compositionality results of the new progress operator $\rightarrowtail$. It is an important section. The two sections that follow the next section are included for the completeness sake. The first will discuss program transformations, which can be used as an alternative method to design a program as in [Bac90, R.95]. The second will present a standard operational semantics for UNITY and present some soundness results of the UNITY logic with respect to the mentioned semantics.

4.8 Write Disjoint Composition

Stronger compositionality results can be obtained for programs that are write-disjoint. Recall (from Chapter 3) that two programs P and Q are said to be write-disjoint if they write to no common variable. This is denoted by $P \div Q$. If two programs P and Q are write-disjoint, Q can only influence P through P's input variables, that is, the variables read by P but not written by it. Consequently, once P and Q agree on a set of input values for P, whatever progress P makes through its write variables will be preserved in the parallel composition of P and Q. In Chapter 3 we hypothesized a law called Transparency law which states exactly this. This will be proven in this section. The Transparency law is fundamental for write-disjoint composition. Some well known design techniques that we use in practice are corollaries of this principle. A progress property is usually constructed, either using transitivity or disjunction, from a number of simpler progress properties. Using the principle we can delegate each constituent property, if we so desire, to be realized by a write-disjoint component of a program.

Parallel composition of write-disjoint programs is also attractive because it occurs frequently in practice. The pictures in Figure 2.3 show various instances of composition of write-disjoint programs. We will define some of them below.

Definition 4.8.1 WRITE-DISJOINT PROGRAMS *WD_DEF*

$$P \div Q \;=\; (\mathbf{w}P \cap \mathbf{w}Q = \emptyset)$$

Definition 4.8.2 LAYERING *LAYERING*

$$P \triangleright Q \;=\; (P \div Q) \wedge (\mathbf{w}P \supseteq \mathbf{i}Q)$$

Definition 4.8.3 FORK *FORK*

$$P \Cap Q \;=\; (P \div Q) \wedge (\mathbf{i}P = \mathbf{i}Q)$$

Definition 4.8.4 NON-INTERFERING (PARALLEL) *FPAR*

$$P \| Q \;=\; (P \div Q) \wedge (\mathbf{r}P \cap \mathbf{r}Q = \emptyset)$$

If $P \div Q$, then the parallel composition of P and Q is also called the *write-disjoint composition* of P and Q. Obviously, if $\triangleright$, $\Cap$, and $\|$ are all instances of $\div$.

In a *non-interfering parallel composition* of two programs, both programs are independent from each other. In a *fork*, the programs based their computation on the same set of input variables. For example if we have a program that computes the minimum of the values of the variables in V, and another program that computes the maximum, we can put the two programs in parallel by forking them. In a *layering* we have two layers. If $P \triangleright Q$ holds, then P is called the *lower layer* and Q the *upper layer*. The computation of the upper layer depends on the results of the lower layer.

The converse does not necessarily hold. For example, the lower layer can be a program that constructs a spanning tree from a vertex i and the upper layer is a program that broadcasts messages from i, using the constructed spanning tree. Layering works like a higher level sequential composition. However, the two layers do not have to be implemented sequentially, especially if they are non-terminating programs.

Before the Transparency law can be proven, first we need the following lemma, stating that if P and Q are write disjoint, then Q cannot destroy a predicate confined by $\mathbf{w}P$:

Lemma 4.8.5 *CONF_Local*

$$\frac{(P \div Q) \ \wedge \ p \in \mathsf{Pred}.(\mathbf{w}P)}{{}_{Q}\vdash \circlearrowleft p}$$

Proof:
The lemma follows from Lemma $3.4.6_{33}$ in Chapter 3:

$$\begin{array}{ll} & {}_{Q}\vdash \circlearrowleft p \\ = & \quad \{ \text{ definition of } \circlearrowleft (4.3.8) \ \} \\ & (\forall a : a \in \mathbf{a}Q : \{p\}\ a\ \{p\}) \\ \Leftarrow & \quad \{ \text{ Lemma } 3.4.6_{33} \ \} \\ & (\forall a : a \in \mathbf{a}Q : \mathbf{w}P \nleftarrow a \ \wedge \ p \in \mathsf{Pred}.(\mathbf{w}P)) \\ \Leftarrow & \quad \{ \ P \text{ and } Q \text{ are write-disjoint } \} \\ & p \in \mathsf{Pred}.(\mathbf{w}P) \end{array}$$

▲

Now the transparency law:

Theorem 4.8.6 Transparency Law *REACH_TRANSPARANT*

$$\frac{P \div Q \ \wedge \ ({}_{Q}\vdash \circlearrowleft J) \ \wedge \ (J \ {}_{P}\vdash p \rightarrowtail q)}{J \ {}_{P[\!]Q}\vdash p \rightarrowtail q}$$

Proof:
By applying $\rightarrowtail$ Induction$_{56}$ it suffices to show that $\mathfrak{R} = (\lambda p, q.\ J \ {}_{P[\!]Q}\vdash p \rightarrowtail q)$ is transitive, left-disjunctive, and includes $\mathfrak{E} = (\lambda p, q.\ J \ {}_{P}\vdash p \ \mathbf{ensures}\ q)$. That $\mathfrak{R}$ is transitive and left-disjunctive follows from $\rightarrowtail$ Transitivity$_{56}$ and Disjunctivity$_{56}$. For the inclusion of $\mathfrak{E}$ we derive:

$$\begin{array}{ll} & J \ {}_{P[\!]Q}\vdash p \rightarrowtail q \\ \Leftarrow & \quad \{ \ \rightarrowtail \text{Introduction}_{56} \ \} \\ & p, q \in \mathsf{Pred}.(\mathbf{w}(P[\!]Q)) \ \wedge \ ({}_{P[\!]Q}\vdash \circlearrowleft J) \ \wedge \ ({}_{P[\!]Q}\vdash J \wedge p \ \mathsf{ensures}\ q) \\ \Leftarrow & \quad \{ \ \mathbf{w}P \subseteq \mathbf{w}P[\!]Q; \text{ Confinement Monotonicity}_{30} \ \} \end{array}$$

$$
\begin{array}{ll}
& p, q \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge\ (\ {}_{P[\!]Q}\!\vdash \circlearrowleft J) \ \wedge\ (\ {}_{P[\!]Q}\!\vdash p \wedge J \ \mathsf{ensures}\ q) \\
= & \quad \{\ \circlearrowleft \text{COMPOSITIONALITY}_{60}\ \} \\
& p, q \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge\ (\ {}_{P}\!\vdash \circlearrowleft J) \ \wedge\ (\ {}_{Q}\!\vdash \circlearrowleft J) \ \wedge\ (\ {}_{P[\!]Q}\!\vdash p \wedge J \ \mathsf{ensures}\ q) \\
\Leftarrow & \quad \{\ \mathsf{ensures}\ \text{COMPOSITIONALITY}_{60} \text{ and the definition}_{54} \text{ of } \textbf{ensures}\ \} \\
& (\ {}_{Q}\!\vdash \circlearrowleft J) \ \wedge\ (\ {}_{Q}\!\vdash p \wedge J \ \mathsf{unless}\ q) \ \wedge\ (J\ {}_{P}\!\vdash p \ \textbf{ensures}\ q) \\
\Leftarrow & \quad \{\ \mathsf{unless}\ \text{POST-WEAKENING}_{46};\ \text{definition of } \circlearrowleft;\ \circlearrowleft \text{CONJUNCTION}_{47}\ \} \\
& (\ {}_{Q}\!\vdash \circlearrowleft J) \ \wedge\ (\ {}_{Q}\!\vdash \circlearrowleft p) \ \wedge\ (J\ {}_{P}\!\vdash p \ \textbf{ensures}\ q) \\
\Leftarrow & \quad \{\ \text{Lemma } 4.8.5\ \} \\
& (\ {}_{Q}\!\vdash \circlearrowleft J) \ \wedge\ p \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge\ (P \div Q) \ \wedge\ (J\ {}_{P}\!\vdash p \ \textbf{ensures}\ q) \\
= & \quad \{\ \text{definition}_{54} \text{ of } \textbf{ensures}\ \} \\
& (\ {}_{Q}\!\vdash \circlearrowleft J) \ \wedge\ (P \div Q) \ \wedge\ (J\ {}_{P}\!\vdash p \ \textbf{ensures}\ q)
\end{array}
$$

▲

For example consider again the example with Sender and Buffer in Figure 4.14_{61}. Recall that we were discussing about the specification (4.7.1) of Sender[]Buffer:

$$(\forall X :: \mathsf{true}\ {}_{SB}\!\vdash (\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \rightarrowtail (\mathsf{out} = X))$$

The specification was refined into a number of specifications. One of them is (4.7.4):

$$\mathsf{true}\ {}_{\mathsf{Sender[]Buffer}}\!\vdash X \in \mathsf{buf} \rightarrowtail (\mathsf{out} = X)$$

By looking into the code of Buffer we conclude that this progress will be established by Buffer, no matter what Sender does. But how can we conclude this without having to look into the code of Buffer? Notice that Sender and Buffer are write-disjoint. Using the TRANSPARENCY law we can refine the above to:

$$\mathsf{true}\ {}_{\mathsf{Buffer}}\!\vdash X \in \mathsf{buf} \rightarrowtail (\mathsf{out} = X) \tag{4.8.1}$$

stating that the required progress can indeed be delegated to Buffer.

One may ask, whether a special law for write-disjoint programs is really necessary. That is, whether it can derived from the SINGH LAW given in the previous section. The latter expresses how, in general, a progress property of a program P may be influenced by another program Q (through the variables in $Q!?P$). However, the SINGH LAW does not discriminate between progress properties which are 'directly' dependent on $Q!?P$ and those that are not. For example, we cannot obtain (4.8.1) above from

$$\mathsf{true}\ {}_{\mathsf{Sender[]Buffer}}\!\vdash X \in \mathsf{buf} \rightarrowtail (\mathsf{out} = X)$$

using the SINGH_{65} law because the law *presumes* the worst case, which is that any progress made by Buffer may suffer from interference by Sender, which is not always true.

An instance of write-disjoint composition called *layering* —also called collateral composition— has been recognized by Herman [Her91] and Arora [Aro92] as an important technique to combine self-stabilizing programs. The following law is especially useful to handle layering.

Theorem 4.8.7 SPIRAL LAW *REACH_SPIRAL*

$$\frac{(P \div Q) \ \wedge \ ({}_{P}\vdash \circlearrowright (J \wedge q)) \ \wedge \ ({}_{Q}\vdash \circlearrowright J) \qquad (J \ {}_{P}\vdash p \rightarrowtail q) \ \wedge \ (J \wedge q \ {}_{Q}\vdash \mathsf{true} \rightarrowtail r)}{J \ {}_{P[]Q}\vdash p \rightarrowtail q \wedge r}$$

If $P \triangleright Q$ holds, $P[]Q$ is a layering with P as the lower layer and Q as the upper layer. According to the law, a progress property $p \rightarrowtail r$ in $P[]Q$ can be split into $p \rightarrowtail q$ in the lower layer P, and $q \vdash \mathsf{true} \rightarrowtail r$ in the upper layer Q. The SPIRAL law is used to implement a sequential division of tasks. For example if we want to do a broadcast, we can think of a two-steps process: first, construct a spanning tree, and then do the actual broadcast. Usually we have separate programs for both tasks. The SPIRAL Law provides the required justification for this kind of separation, where in this case P constructs the spanning tree and Q performs the broadcast under the assumption that q describes the existence of this spanning tree.

4.9 Program Transformations

Parallel composition or sequential composition are, as the name implies, program composition in which two programs are combined to form a larger one. There are also program transformations in which a program is transformed into another one. In [CM88] only the addition of assignments to fresh variables is mentioned. There are of course more useful transformations but at the time not much was known about how exactly they affect the behavior of a program. Recent results were given by Singh in [Sin93] who exposed data refinement, guard strengthening, and refinement of atomicity, and investigated the kind of program properties preserved by these transformations. There is also the work by Udink, Herman, and Kok in [UHK94] which presented action duplication, data refinement, guard strengthening, and action substitution using invariants, and proved that these transformations preserved some form of local safety and progress properties.

In this section several program transformations will be discussed. The main question that we wish to address is how the transformations affect the $\rightarrowtail$ properties of a program. We expect that the results in [CM88] and [Sin93] for $\mapsto$ will translate well to results in $\rightarrowtail$. In addition, we find the laws of superposition (additions of assignments to fresh variables) in [CM88] to be somewhat informally stated. We will re-state them, with proper details. The reader may also find it interesting to see how the transformation laws presented later can be neatly proven from the laws at the action level presented in Chapter 3. The proofs are collected separately in Section 4.12. Program transformation is however not a main issue in this thesis. The technique is not going to be used in the applications presented later in this thesis. Therefore we are not going to be too elaborate. The results are also not mechanically verified yet.

If P is a UNITY program, adding variables to P, or a skip action, preserves the properness of P. That is, the resulting program also satisfies Unity.P. Obviously, this simple transformation preserves whatever unless, ensures, $\rightarrowtail$, and invariant properties

of P. In addition, strengthening the initial condition of a program also preserves such properties.

Recall than an action a can be extended with an assignment b to some fresh variables by composing b 'in front of' a: $b;a$. Assume that V are the variables read by a. Naturally, we expect that extending a with b preserves any Hoare triple specification of a, as long as the specification does not refer to the fresh variables. This is justified by Corollary $3.4.4_{32}$ from Chapter 3 which states that $\{p\}\ a\ \{q\}$ implies $\{p \restriction V\}\ b;a\ \{q \restriction V\}$. The same should also hold at the program level. In UNITY, the addition of assignments to fresh variables is called *superposition*. In addition, if we can extend the actions in a program with assignments to fresh variables, we can also add new actions which only assign to fresh variables.

Theorem 4.9.1 PRIMITIVE PROPERTIES AFTER SUPERPOSITION
Let P and Q be UNITY programs and $f \in \mathsf{Action} \to \mathsf{Action}$ such that: $\mathbf{a}Q = \{f.a;a \mid a \in \mathbf{a}P\} \cup \{f.a \mid a \in A\}$ for some $A \subseteq \mathsf{Action}$. Let $\mathfrak{R}$ be either unless or ensures. We have:

$$\frac{(\forall a : a \in \mathbf{a}P : \mathbf{r}P \nleftarrow f.a)\ \wedge\ ({}_{P}\vdash p\ \mathfrak{R}\ q)}{{}_{Q}\vdash (p \restriction \mathbf{r}P)\ \mathfrak{R}\ (q \restriction \mathbf{r}P)}$$

◀

Notice that Q is obtained from P by extending each action a of P with an assignment $f.a$ (which can also be skip). The first conjunct in the assumption of the above law expresses the fact that $f.a$ only assigns to fresh variables. A similar law also exists for $\rightarrowtail$. It can easily be proven using $\rightarrowtail$ INDUCTION$_{56}$.

Theorem 4.9.2 $\rightarrowtail$ AFTER SUPERPOSITION
Let P and Q be UNITY programs and $f \in \mathsf{Action}{\to}\mathsf{Action}$ such that $\mathbf{w}P \subseteq \mathbf{w}Q$ and: $\mathbf{a}Q = \{f.a;a \mid a \in \mathbf{a}P\} \cup \{f.a \mid a \in A\}$ for some $A \subseteq \mathsf{Action}$. We have:

$$\frac{(\forall a : a \in \mathbf{a}P : \mathbf{r}P \nleftarrow f.a)\ \wedge\ (J\ {}_{P}\vdash p \rightarrowtail q)}{J \restriction \mathbf{r}P\ {}_{Q}\vdash p \rightarrowtail q}$$

◀

An action a can be strengthened with a guard g by extending it to if g then a. The meaning of this action, according to the convention made Section 4.2 is:

$$(\lambda s,t.\ (g.s \Rightarrow a.s.t)\ \wedge\ (\neg g.s \Rightarrow (s = t)))$$

It is known that strengthening the actions of a program with guards preserves its safety properties:

Theorem 4.9.3 SAFETY UNDER A STRONGER GUARD
Let P and Q be UNITY programs, $g \in \mathsf{Action}{\to}\mathsf{Pred}$, and $A \subseteq \mathbf{a}P$ such that $\mathbf{a}Q = (\mathbf{a}P \backslash A) \cup \{\mathsf{if}\ g.a\ \mathsf{then}\ a \mid a \in A\}$ We have:

$$\frac{{}_P\vdash p\ \mathsf{unless}\ q}{{}_Q\vdash p\ \mathsf{unless}\ q}$$

◀

Notice that Q is obtained from P by adding a guard $g.a$ to each action a from A.

If an action a in P is strengthened with a guard g, whatever progress by $\rightarrowtail$ in P will be preserved in the new program, *if* the other actions cannot destroy g, and if it will eventually hold. This is expressed by the following theorem[9].

Theorem 4.9.4 PROGRESS UNDER A STRONGER GUARD
Let P and Q be UNITY programs, $g \in \mathsf{Action}{\to}\mathsf{Pred}$, and $A \subseteq \mathbf{a}P$ such that $\mathbf{w}P \subseteq \mathbf{w}Q$ and $\mathbf{a}Q = (\mathbf{a}P \backslash A) \cup \{\mathsf{if}\ g.a\ \mathsf{then}\ a \mid a \in A\}$. Let Q_{-a} be the same program as Q, except that the action a is deleted. We have:

$$\frac{(\forall a : a \in A : ({}_{Q_{-a}}\vdash \circlearrowright J \wedge g.a)\ \wedge\ (J\ {}_Q\vdash \mathsf{true} \rightarrowtail g.a))\ \wedge\ (J\ {}_P\vdash p \rightarrowtail q)}{J\ {}_Q\vdash p \rightarrowtail q}$$

◀

Notice that the condition $J\ {}_Q\vdash \mathsf{true} \rightarrowtail g.a$ states that in the new program Q, eventually the guard $g.a$ becomes true. The condition ${}_{Q_{-a}}\vdash \circlearrowright J \wedge g$ states that no other action but a can falsify the guard $g.a$.

If ${}_P\vdash J \wedge p\ \mathsf{unless}\ q$ holds, and in addition J implies that $g = h$, then replacing an action $\mathsf{if}\ g\ \mathsf{then}\ a$ in P with $\mathsf{if}\ h\ \mathsf{then}\ a$ preserves $J \wedge p\ \mathsf{unless}\ q$. Typically J is an invariant, or at least a stable predicate. A similar result also exists for $\mathsf{ensures}$ and $\rightarrowtail$.

Theorem 4.9.5 ACTIONS SUBSTITUTION
Let P, Q and R be UNITY programs such that

$$Q = P[\!]([\{\mathsf{if}\ g_1\ \mathsf{then}\ \mathsf{assign}.x.f_1\}, \mathbf{ini}P, \mathbf{r}P, \mathbf{w}P)$$
$$R = P[\!]([\{\mathsf{if}\ g_2\ \mathsf{then}\ \mathsf{assign}.x.f_2\}, \mathbf{ini}P, \mathbf{r}P, \mathbf{w}P)$$

and $x \in \mathbf{w}P$. Let $\mathfrak{R}$ be either unless or $\mathsf{ensures}$. If J satisfies $[J \Rightarrow (\lambda s.\ f_1.s, g_1.s = f_2.s, g_2.s)]$ then we have:

$$\frac{{}_Q\vdash J \wedge p\ \mathfrak{R}\ q}{{}_R\vdash J \wedge p\ \mathfrak{R}\ q}\quad \text{and} \quad \frac{J\ {}_Q\vdash p \rightarrowtail q}{J\ {}_R\vdash p \rightarrowtail q}$$

◀

As an example consider two programs P and Q which communicate through the assignment (of P) $\mathsf{if}\ g\ \mathsf{then}\ y := x$ where x is intended to be a variable of P and y of Q. The assignment can be viewed as an action by P to send a new datum it keeps in

[9] A stronger result was given by Singh in [Sin93].

x to the variable y. As it is, P can send a new datum whenever g is true. Suppose that we want to implement this communication on a synchronous machine. That is, sending a new datum is only possible if not only P, but also Q is ready to receive the datum. Below we show an implementation of this. The read, write, and init sections are omitted for the sake of simplicity.

```
prog   P'
assign
       ...
[]     if Prdy ∧ Qrdy then if g then  y, Prdy := x, false
[]     if ¬Prdy ∧ ¬Qrdy then Prdy := true

prog   Q'
assign
       ...
[]     if Prdy ∧ ¬Qrdy then Qrdy := true
[]     if ¬Prdy ∧ Qrdy then Qrdy := false
```

The variables Prdy and Qrdy are assumed fresh with respect to P and Q. P' is ready to send a new datum if Prdy is true and Q' is ready to receive one if Qrdy is true. Only when Prdy and Qrdy are both true then a communication can take place. Notice that the protocol is in fact the 4-phase hand-shake protocol as in Figure 4.14.

We feel that $P' [\!] Q'$ somehow 'implements' $P [\!] Q$. But how can one justify this? We observe that we can obtain $P' [\!] Q'$ from P and Q through a series of previously described transformations, and recall that the transformations preserve —under some conditions— unless and $\rightarrowtail$ properties.

First, we can transform P by adding an assignment Prdy := true and extending the action if g then $y := x$ with if g then Prdy := false. Notice that these are assignments to Prdy, which is fresh. We obtain the following program:

```
prog   P1
assign
       ...
[]     (if g then Prdy := false) ; (if g then y := x)
[]     Prdy := true
```

By adding assignments to the fresh variable Qrdy we can also transform Q to the following:

```
prog   Q1
assign
       ...
[]     Qrdy := true
[]     Qrdy := false
```

By adding guards we can obtain P' from P_1 and Q' from Q_1.

By the transformation laws given earlier, it can be concluded that ${}_{P[\!]Q}\vdash p$ unless q implies ${}_{P'[\!]Q'}\vdash p$ unless q and J ${}_{P[\!]Q}\vdash p \rightarrowtail q$ implies J ${}_{P'[\!]Q'}\vdash p \rightarrowtail q$, if J, p, and q are all

confined by Pred.($\mathbf{r}P$) and if each new guard eventually holds and can only be falsified by the action it stands guard for. If the reader observes the code of P' and Q' and considers the fact that Pry and Qrdy are fresh variables, he should be able to conclude that the latter condition is met.

One can continue —or take different transformations— by, for example, sharpening the condition that determines the readiness of Q to receive and hence giving Q more control in synchronizing with P.

4.10 The Semantics of UNITY

Eventually, one may want to relate the logic defined by UNITY and some operational semantics. In doing so, one usually hopes to investigate how far the logic reflects the 'real' world. This raises the question of soundness and completeness of the logic with respect to the given operational semantics. Another reason is that in some cases, reasoning may be easier if conducted at the operational level. So, some ability to go back and forth between the logical and operational level will be appreciated.

In this section, an operational semantics for UNITY will be given. The semantic domains are quite straightforwardly chosen, namely all possible sequences of states which can be generated by a UNITY program (this is a standard model). An operational notion of unless and $\mapsto$ (leads-to) will be defined based on these semantics. It has been proven that these two operators are *sound* with respect to their operational counterparts, but not complete. It has been shown that Sanders' version of unless and $\mapsto$ are *complete* [San91, Pac92]. There is no reason for us to repeat these results. However, we wish to mention here that we have *mechanically verified* the soundness results. Quite unfortunately, due to time constraints, we did not succeed in verifying the completeness results.

Recall that an execution of a UNITY program is an infinite sequence of actions such that each action occurs infinitely often (fairness). Let exec.P denote the set of all possible UNITY execution of P. We will represent an infinite sequence over A with a function from $\mathbb{N}$ to A. Let tr be the set of all possible sequence of states which can be generated by the executions in exec. A member of tr.P is called a *trace* of P.

Definition 4.10.1 Execution Set *EXEC_DEF*

$$\sigma \in \mathsf{exec}.P \;=\; (\forall i :: \sigma.i \in \mathbf{a}P) \;\wedge\; (\forall i, a : a \in \mathbf{a}P : (\exists j : i \leq j : \sigma.j = a))$$

Definition 4.10.2 Trace Set *TRACE_DEF*

$$\tau \in \mathsf{tr}.P \;=\; \mathbf{ini}P.(\tau.0) \;\wedge\; (\exists \sigma : \sigma \in \mathsf{exec}.P : (\forall i :: (\sigma.i).(\tau.i).(\tau.(i+1))))$$

◀

Let ${}_P\vdash p\ \mathfrak{U}\ q$ mean that for any trace τ of P, if $\tau.i$ satisfies $p \wedge \neg q$, then $\tau.(i+1)$ satisfies $p \vee q$. Let ${}_P\vdash p\ \mathfrak{L}\ q$ mean that for any trace τ of P, if $\tau.i$ satisfies p, then there exists a j, $i \leq j$, such that $\tau.j$ satisfies q. Indeed, $\mathfrak{U}$ and $\mathfrak{L}$ are intended to be the

operational interpretation of unless and $\mapsto$.

Definition 4.10.3 OPERATIONAL UNLESS *tUNLESS_ADEF1*

$$ {}_P\vdash p\ \mathfrak{U}\ q \ = \ (\forall i, \tau : \tau \in \mathsf{tr}.P : (p \wedge \neg q).(\tau.i) \ \Rightarrow \ (p \vee q).(\tau.(i+1))) $$

Definition 4.10.4 OPERATIONAL LEADS-TO *tLEADSTO_ADEF1*

$$ {}_P\vdash p\ \mathfrak{L}\ q \ = \ (\forall i, \tau : \tau \in \mathsf{tr}.P : p.(\tau.i) \ \Rightarrow \ (\exists j : i \leq j : q.(\tau.j))) $$

◀

unless and $\mapsto$ are sound with respect to the above operational interpretation. They are however not *complete*. The Sanders' definition of unless and $\mapsto$ (given in Section 4.6) are complete with respect to the above interpretation. The soundness of Sanders' definition follows from the soundness of the standard unless and $\mapsto$. We did not verify any completeness result. If the reader is interested, an elegant completeness proof can be found in [Pac92].

Theorem 4.10.5 SOUNDNESS OF unless *UNLESS_IMP_tUNLESS*

$$ ({}_P\vdash p \ \mathsf{unless}\ q) \ \Rightarrow \ ({}_P\vdash p\ \mathfrak{U}\ q) $$

Theorem 4.10.6 SOUNDNESS OF $\mapsto$ *LEADSTO_IMP_tLEADSTO*

$$ ({}_P\vdash p \mapsto q) \ \Rightarrow \ ({}_P\vdash p\ \mathfrak{L}\ q) $$

◀

Now, how about the operational meaning of $\rightarrowtail$? We could not come with any satisfactory answer. The only thing that we know is, as given equation $(4.5.3)_{54}$ in Section 4.5, that $\rightarrowtail$ includes $\mapsto$:

$$ (\mathsf{true}\ {}_P\vdash p \rightarrowtail q) \ \Rightarrow \ ({}_P\vdash p \mapsto q) $$

Or, slightly more general, we can prove:

$$ (J\ {}_P\vdash p \rightarrowtail q) \ \Rightarrow \ ({}_P\vdash J \wedge p \mapsto q) \qquad (4.10.1) $$

Since $J\ {}_P\vdash p \rightarrowtail q$ also implies ${}_P\vdash J$ unless false, It follows then, that $J\ {}_P\vdash p \rightarrowtail q$ implies ${}_P\vdash J\ \mathfrak{U}$ false and ${}_P\vdash J \wedge p\ \mathfrak{L}\ q$. We do not expect equivalence though, because $\rightarrowtail$ is quite different from $\mapsto$. This has been suggested early in Section 4.5, but let us go over an example presented there again. Consider the following program:

```
prog   P
read   {a, x}
write  {x}
init   true
assign if a = 0 then x := 1
[]     if a = 1 then x := 1
[]     if a = 2 then x := x + 1
```

In the above program we have $(b = 0) \wedge a < 2 \mapsto (x = 1)$. We expect that the $\rightarrowtail$ version of this property, namely $(b = 0) \wedge a < 2 \vdash \mathsf{true} \rightarrowtail (x = 1)$, also holds. But this is not true. The property $(b = 0) \wedge a < 2 \mapsto (x = 1)$ can be concluded because we have $(b = 0) \wedge (a = 0)$ **ensures** $(x = 1)$ and $(b = 0) \wedge (a = 1)$ **ensures** $(x = 1)$ and we can join them using the disjunctivity of $\mapsto$. Unfortunately we cannot do the same with $\rightarrowtail$. If we do that, we will get an unsound logic. Consider the program TikToe in Figure 4.9. It is redisplayed below:

```
prog    TikToe
read    {a, b}
write   {a}
init    true
assign  if a = 0 then a := 1
[]      if a = 1 then a := 0
[]      if b ≠ 0 then a := a + 1
```

The programs P and TikToe are write-disjoint. Suppose $(b = 0) \wedge a < 2 \ {}_P\vdash \mathsf{true} \rightarrowtail (x = 1)$ holds. The predicate $(b = 0) \wedge a < 2$ is also stable in Tiktoe. By the TRANSPARENCY$_{72}$ principle we conclude that $(b = 0) \wedge a < 2 \ {}_{P[]\mathsf{TikToe}}\vdash \mathsf{true} \rightarrowtail (x = 1)$ also holds. But this simply cannot be true. Consider the execution:

```
[ if a = 0 then a := 1 ; if a = 0 then x := 1 ;
  if a = 1 then a := 0 ; if a = 1 then x := 1 ;
  if a = 2 then x := x + 1 ; if b ≠ 0 then a := a + 1 ]*
```

which is a fair execution of $P[]$TikToe, but with this execution x will never be equal to 1 if initially $x \neq 1 \wedge a < 2 \wedge (b = 0)$.

So, $\rightarrowtail$ is not as disjunctive as $\mapsto$ can be: it has to be less disjunctive because, as we have seen, this is crucial for the TRANSPARENCY law. As said, we could not come up with a satisfactory operational semantics for $(\lambda p, q.\ J \ {}_P\vdash\ p \rightarrowtail q)$. We suspect however, that this is the largest subrelation of $(\lambda p, q.\ {}_P\vdash\ J \wedge p\mathfrak{L}q)$ which satisfies the TRANSPARENCY law.

4.11 Related Work

In [UHK94], Udink, Herman, and Kok defined a new progress operator. The new operator is somewhere between **ensures** and $\mapsto$. It has a very nice compositionality property but it is a rather complicated operator. However, the authors also provided a class of program transformations which preserve safety and progress under the new operator. Reasoning can be carried out in terms of transformations. In fact, the transformations discussed in Section 4.9 were much inspired by the work in [UHK94].

The issue of compositionality in distributed programming has received quite a lot of attention. In [Zwi88] Zwiers proposed a compositional logic for synchronously communicating processes. In [dBvH94] de Boer and van Hulst proposed a compositional logic for asynchronous systems, and in [PJ91] Pandya and Joseph proposed yet another compositional logic for both synchronous and asynchronous systems. The focus

of these papers are focussed around the assumed means of communication, namely channels. Partial correctness is considered. In UNITY however, total correctness is very important, since otherwise no progress can be concluded. In this thesis, attention is focused on the compositionality of progress in general, but indeed further research is required to develop some basic theory for UNITY regarding channel-based communication. Such an investigation will surely benefit the results of the above mentioned papers.

Closely related work was done by de Boer and his colleagues in [dBKPJ93] where they gave a compositional semantics of local blocks. A local block is a part of a program in which it does some internal computation. Such a computation is not visible from outside, and therefore cannot be directly influenced either. Recall that in Section 4.8 we discussed write-disjoint programs. If two programs P and Q are write-disjoint, then the write variables of P are in a sense local, because they cannot be written by Q — although Q may still be able to observe them. In [UK93a] Udink and Kok investigated the relation between various operational semantics for UNITY and the preservation of UNITY properties under program refinement. In their subsequent paper [UK93b], semantics that preserves program refinement within a context were proposed.

4.12 Postponed Proofs

Theorem 4.9.1
Let P and Q be UNITY programs and $f \in \mathsf{Action}{\rightarrow}\mathsf{Action}$ such that: $\mathbf{a}Q = \{f.a; a \mid a \in \mathbf{a}P\} \cup \{f.a \mid a \in A\}$ for some $A \subseteq \mathsf{Action}$. Let $\mathfrak{R}$ be either unless or ensures. We have:

$$\frac{(\forall a : a \in \mathbf{a}P : \mathbf{r}P \nleftarrow f.a) \ \wedge \ ({}_{P}\vdash p \ \mathfrak{R} \ q)}{{}_{Q}\vdash (p \restriction \mathbf{r}P) \ \mathfrak{R} \ (q \restriction \mathbf{r}P)}$$

Proof:
We will only show the case $\mathfrak{R} =$ unless. The case of ensures can be proven in a much similar way. We have to show:

$${}_{Q}\vdash (p \restriction \mathbf{r}P) \ \mathsf{unless} \ (q \restriction \mathbf{r}P)$$

By the definition$_{42}$ of unless it suffices to show that for all $b \in \mathbf{a}Q$ we have:

$$\{(p \restriction \mathbf{r}P) \wedge \neg(q \restriction \mathbf{r}P)\} \ b \ \{(p \restriction \mathbf{r}P) \vee (q \restriction \mathbf{r}P)\}$$

If $b = f.a; a$, for some $a \in \mathbf{a}P$, we derive:

$$\begin{array}{ll} & \{(p \restriction \mathbf{r}P) \wedge \neg(q \restriction \mathbf{r}P)\} \ f.a; a \ \{(p \restriction \mathbf{r}P) \vee (q \restriction \mathbf{r}P)\} \\ = & \quad \{ \text{ projection distributes over predicate operators } \} \\ & \{(p \wedge \neg q) \restriction \mathbf{r}P\} \ f.a; a \ \{(p \vee q) \restriction \mathbf{r}P\} \\ \Leftarrow & \quad \{ \text{ Corollary } 3.4.4_{32} \ \} \end{array}$$

$$
\begin{array}{ll}
& \mathbf{r}P \twoheadleftarrow f.a \ \wedge \ (\mathbf{r}P)^{\mathsf{c}} \twoheadleftarrow a \ \wedge \ \{p \wedge \neg q\}\ a\ \{p \vee q\} \\
= & \quad \{\ P \text{ is a UNITY program}\ \} \\
& \mathbf{r}P \twoheadleftarrow f.a \ \wedge \ \{p \wedge \neg q\}\ a\ \{p \vee q\} \\
= & \quad \{\ \text{assumption}\ \} \\
& \{p \wedge \neg q\}\ a\ \{p \vee q\}
\end{array}
$$

The last follows from ${}_P\vdash p$ unless q.

If $b = f.a$, for some $a \in A$, then it can also be written as $f.a$; skip and an argument quite similar to the one applies.

▲

Theorem 4.9.4

Let P and Q be UNITY programs, $g \in \mathsf{Action}{\rightarrow}\mathsf{Pred}$, and $A \subseteq \mathbf{a}P$ such that $\mathbf{w}P \subseteq \mathbf{w}Q$ and $\mathbf{a}Q = (\mathbf{a}P \backslash A) \ \cup \ \{\mathsf{if}\ g.a\ \mathsf{then}\ a \mid a \in A\}$. Let Q_{-a} be the same program as Q, except that the action a is deleted. We have:

$$
\frac{(\forall a : a \in A : ({}_{Q_{-a}}\vdash \circlearrowright J \wedge g.a) \ \wedge \ (J\ {}_Q\vdash \mathsf{true} \rightarrowtail g.a)) \ \wedge \ (J\ {}_P\vdash p \rightarrowtail q)}{J\ {}_Q\vdash p \rightarrowtail q}
$$

▶

Proof:

Using $\rightarrowtail$ INDUCTION$_{56}$ it suffices to show that $\mathfrak{R} = (\lambda p, q.\ J\ {}_Q\vdash p \rightarrowtail q)$ is transitive, left-disjunctive, and includes $\mathfrak{E} = (\lambda p, q.\ J\ {}_P\vdash p\ \mathfrak{ensures}\ q)$, assuming that ${}_{Q_{-a}}\vdash \circlearrowright J \wedge g.a$ and $J\ {}_Q\vdash \mathsf{true} \rightarrowtail g.a$ hold for all $a \in A$.

The transitivity and left-disjunctivity of $\mathfrak{R}$ follow from $\rightarrowtail$ TRANSITIVITY$_{56}$ and DISJUNCTION$_{56}$. As for the inclusion of $\mathfrak{E}$, assume $\mathfrak{E}.p.q$. Hence, by the definitions of $\mathfrak{E}$, $\mathfrak{ensures}$, and ensures we have:

$$
\begin{array}{ll}
p, q \in \mathsf{Pred}.(\mathbf{w}P) & (4.12.1) \\
{}_P\vdash \circlearrowright J & (4.12.2) \\
{}_P\vdash J \wedge p\ \mathsf{unless}\ q & (4.12.3) \\
\{J \wedge p \wedge \neg q\}\ a\ \{q\} & (4.12.4)
\end{array}
$$

for some $a \in \mathbf{a}P$. If $a \notin A$ (and hence $a \in \mathbf{a}Q$) we derive:

$$
\begin{array}{ll}
& J\ {}_Q\vdash p \rightarrowtail q \\
\Leftarrow & \quad \{\ \rightarrowtail \text{INTRODUCTION}_{56}\ \} \\
& p, q \in \mathsf{Pred}.(\mathbf{w}Q) \ \wedge \ ({}_Q\vdash \circlearrowright J) \ \wedge \ ({}_Q\vdash J \wedge p\ \mathsf{ensures}\ q) \\
= & \quad \{\ \mathbf{w}P \subseteq \mathbf{w}Q, \text{CONFINEMENT MONOTONICITY}_{30}, (4.12.1)\ \} \\
& ({}_Q\vdash \circlearrowright J) \ \wedge \ ({}_Q\vdash J \wedge p\ \mathsf{ensures}\ q) \\
\Leftarrow & \quad \{\ \text{definition of ensures}, a \in \mathbf{a}Q\ \} \\
& ({}_Q\vdash \circlearrowright J) \ \wedge \ ({}_Q\vdash J \wedge p\ \mathsf{unless}\ q) \ \wedge \ \{J \wedge p \wedge \neg q\}\ a\ \{q\} \\
= & \quad \{\ (4.12.4)\ \} \\
& ({}_Q\vdash \circlearrowright J) \ \wedge \ ({}_Q\vdash J \wedge p\ \mathsf{unless}\ q)
\end{array}
$$

$\Leftarrow$ { definition$_{44}$ of $\circlearrowleft$, Theorem 4.9.3$_{76}$ }
$({}_P\vdash \circlearrowleft J) \ \wedge \ ({}_P\vdash J \wedge p \text{ unless } q)$

The first is (4.12.2) and the second is (4.12.3). If $a \in A$ we derive first:

$J \ {}_Q\vdash p \rightarrowtail q$
$\Leftarrow$ { $\rightarrowtail$ CANCELLATION$_{56}$ }
$q \in \mathsf{Pred}.(\mathbf{w}Q) \ \wedge \ (J \ {}_Q\vdash p \rightarrowtail (p \wedge g.a) \vee q) \ \wedge \ (J \ {}_Q\vdash p \wedge g.a \rightarrowtail q)$
$\Leftarrow$ { $\rightarrowtail$ PSP$_{56}$ }
$p, q \in \mathsf{Pred}.(\mathbf{w}Q) \ \wedge \ ({}_Q\vdash J \wedge p \text{ unless } q) \ \wedge \ (J \ {}_Q\vdash \text{true} \rightarrowtail g.a) \ \wedge \ (J \ {}_Q\vdash p \wedge g.a \rightarrowtail q)$
$=$ { $\mathbf{w}P \subseteq \mathbf{w}Q$, CONFINEMENT MONOTONICITY$_{30}$, (4.12.1) }
$({}_Q\vdash J \wedge p \text{ unless } q) \ \wedge \ (J \ {}_Q\vdash \text{true} \rightarrowtail g.a) \ \wedge \ (J \ {}_Q\vdash p \wedge g.a \rightarrowtail q)$
$\Leftarrow$ { Theorem 4.9.3$_{76}$ }
$({}_P\vdash J \wedge p \text{ unless } q) \ \wedge \ (J \ {}_Q\vdash \text{true} \rightarrowtail g.a) \ \wedge \ (J \ {}_Q\vdash p \wedge g.a \rightarrowtail q)$

The first conjunct is (4.12.3) and the second is an assumption. Let Q_a be defined as:

$$Q_a = (\{\text{if } g.a \text{ then } a\}, \mathbf{ini}Q, \mathbf{r}Q, \mathbf{w}Q)$$

Note that $Q = Q_{-a}[\!]Q_a$. For the third conjunct we derive:

$J \ {}_Q\vdash p \wedge g.a \rightarrowtail q$
$\Leftarrow$ { $\rightarrowtail$ INTRODUCTION$_{56}$, confinement is preserved by $\wedge$ }
$p, g.a, q \in \mathsf{Pred}.(\mathbf{w}Q) \ \wedge \ ({}_Q\vdash \circlearrowleft J) \ \wedge \ ({}_Q\vdash J \wedge p \wedge g.a \text{ ensures } q)$
$=$ { $\mathbf{w}P \subseteq \mathbf{w}Q$, CONFINEMENT MONOTONICITY$_{30}$, (4.12.1) }
$g.a \in \mathsf{Pred}.(\mathbf{w}Q) \ \wedge \ ({}_Q\vdash \circlearrowleft J) \ \wedge \ ({}_Q\vdash J \wedge p \wedge g.a \text{ ensures } q)$
$\Leftarrow$ { Theorem 4.9.3$_{76}$, definition$_{44}$ of $\circlearrowleft$, (4.12.2) }
$g.a \in \mathsf{Pred}.(\mathbf{w}Q) \ \wedge \ ({}_Q\vdash J \wedge p \wedge g.a \text{ ensures } q)$
$\Leftarrow$ { $Q = Q_{-a}[\!]Q_a$, ensures COMPOSITION$_{60}$ }
$g.a \in \mathsf{Pred}.(\mathbf{w}Q) \ \wedge \ ({}_{Q_{-a}}\vdash J \wedge p \wedge g.a \text{ unless } q) \ \wedge \ ({}_{Q_a}\vdash J \wedge p \wedge g.a \text{ ensures } q)$
$\Leftarrow$ { unless SIMPLE CONJUNCTION$_{46}$, definition$_{44}$ of $\circlearrowleft$ }
$g.a \in \mathsf{Pred}.(\mathbf{w}Q) \wedge ({}_{Q_{-a}}\vdash J \wedge p \text{ unless } q) \wedge ({}_{Q_{-a}}\vdash \circlearrowleft J \wedge g.a) \wedge ({}_{Q_a}\vdash J \wedge p \wedge g.a \text{ ensures } q)$
$=$ { $\mathbf{a}Q_{-a} \subseteq \mathbf{a}Q$, assumption }
$g.a \in \mathsf{Pred}.(\mathbf{w}Q) \ \wedge \ ({}_Q\vdash J \wedge p \text{ unless } q) \ \wedge \ ({}_{Q_a}\vdash J \wedge p \wedge g.a \text{ ensures } q)$
$=$ { Theorem 4.9.3$_{76}$, (4.12.3) }
$g.a \in \mathsf{Pred}.(\mathbf{w}Q) \ \wedge \ ({}_{Q_a}\vdash J \wedge p \wedge g.a \text{ ensures } q)$
$=$ { definition of ensures and Q_a }
$g.a \in \mathsf{Pred}.(\mathbf{w}Q) \ \wedge \ \{J \wedge p \wedge g.a \wedge \neg q\} \text{ if } g.a \text{ then } a \ \{q\}$

The first conjunct follows from $J \ {}_Q\vdash \text{true} \rightarrowtail g.a$ and $\rightarrowtail$ CONFINEMENT$_{57}$. For the second, we derive:

$\{J \wedge p \wedge g.a \wedge \neg q\} \text{ if } g.a \text{ then } a \ \{q\}$
$=$ { definition Hoare triple and if $g.a$ then a }

$$
\begin{array}{ll}
 & (\forall s,t :: (J \wedge p \wedge g.a \wedge \neg q).s \wedge (g.a.s \Rightarrow a.s.t) \wedge (\neg g.a.s \Rightarrow s = t) \;\Rightarrow\; q.t) \\
= & \quad \{ \text{ definition of predicate operators } \} \\
 & (\forall s,t :: (J \wedge \wedge p \wedge \neg q).s \wedge g.a.s \wedge (g.a.s \Rightarrow a.s.t) \wedge (\neg g.a.s \Rightarrow s = t) \;\Rightarrow\; q.t) \\
\Leftarrow & \quad \{ \text{ predicate calculus } \} \\
 & (\forall s,t :: (J \wedge p \wedge \neg q).s \wedge a.s.t \;\Rightarrow\; q.t) \\
= & \quad \{ \text{ definition of Hoare triple } \} \\
 & (4.12.4)
\end{array}
$$

▲

Theorem 4.9.5
Let P, Q and R be UNITY programs such that

$$Q = P[\!](\{\mathsf{if}\ g_1\ \mathsf{then\ assign}.x.f_1\}, \mathbf{ini}P, \mathbf{r}P, \mathbf{w}P)$$
$$R = P[\!](\{\mathsf{if}\ g_2\ \mathsf{then\ assign}.x.f_2\}, \mathbf{ini}P, \mathbf{r}P, \mathbf{w}P)$$

and $x \in \mathbf{w}P$. Let $\mathfrak{R}$ be either unless or ensures. If J satisfies $[J \Rightarrow (\lambda s.\ f_1.s, g_1.s = f_2.s, g_2.s)]$ then we have:

$$\frac{{}_Q\!\vdash J \wedge p\ \mathfrak{R}\ q}{{}_R\!\vdash J \wedge p\ \mathfrak{R}\ q} \quad \text{and} \quad \frac{J\ {}_Q\!\vdash p \rightarrowtail q}{J\ {}_R\!\vdash p \rightarrowtail q}$$

Proof:
We will only proof the unless case. The ensures case can be proven in much the same way and the $\rightarrowtail$ case can subsequently be proven easily using $\rightarrowtail$ INDUCTION$_{56}$. It suffices to show that:

$$\{J \wedge p \wedge \neg q\}\ b\ \{(J \wedge p) \vee q\}$$

holds for all $b \in \mathbf{a}R$. If $b \in \mathbf{a}P$ then it is trivially implied by ${}_Q\!\vdash J \wedge p$ unless q. If $b = \mathsf{if}\ g_2\ \mathsf{then\ assign}.x.f_2$ we derive:

$$
\begin{array}{ll}
 & \{J \wedge p \wedge \neg q\}\ \mathsf{if}\ g_2\ \mathsf{then\ assign}.x.f_2\ \{(J \wedge p) \vee q\} \\
= & \quad \{ \text{ definition of Hoare triple, if-then construct, and assignment } \} \\
 & (\forall s,t ::\ (J \wedge p \wedge \neg q).s\ \wedge\ (g_2.s \Rightarrow (t.x = f_2.s) \wedge (t \restriction \{x\}^{\mathsf{c}} = s \restriction \{x\}^{\mathsf{c}})) \wedge \\
 & \qquad (\neg g_2.s \Rightarrow (s = t))\ \Rightarrow\ ((J \wedge p) \vee q).t) \\
= & \quad \{ \text{ definition of predicate operators } \} \\
 & (\forall s,t ::\ J.s\ \wedge\ (p \wedge \neg q).s\ \wedge\ (g_2.s \Rightarrow (t.x = f_2.s) \wedge (t \restriction \{x\}^{\mathsf{c}} = s \restriction \{x\}^{\mathsf{c}})) \wedge \\
 & \qquad (\neg g_2.s \Rightarrow (s = t))\ \Rightarrow\ ((J \wedge p) \vee q).t) \\
= & \quad \{ [J \Rightarrow (\lambda s.\ f_1.s, g_1.s = f_2.s, g_2.s)] \} \\
 & (\forall s,t ::\ J.s\ \wedge\ (p \wedge \neg q).s\ \wedge\ (g_1.s \Rightarrow (t.x = f_1.s) \wedge (t \restriction \{x\}^{\mathsf{c}} = s \restriction \{x\}^{\mathsf{c}})) \wedge \\
 & \qquad (\neg g_2.s \Rightarrow (s = t))\ \Rightarrow\ ((J \wedge p) \vee q).t) \\
= & \quad \{ \text{ definition of Hoare triple and if-then construct, assignment } \} \\
 & \{J \wedge p \wedge \neg q\}\ \mathsf{if}\ g_1\ \mathsf{then\ assign}.x.f_1\ \{(J \wedge p) \vee q\}
\end{array}
$$

▲