Acta Informatica 30, 503–535 (1993)

Program refinement in fair transition systems

Ambuj K. Singh *

Department of Computer Science, University of California at Santa Barbara, Santa Barbara, CA 93106, USA

Received August 14, 1991 / May 4, 1993

Abstract. Refinement of programs is investigated in the context of fair transition systems. Two kinds of refinements, property preserving and fixed-point preserving, are defined. Conditions are developed under which common program transformations such as data refinement are property preserving and fixed-point preserving. These conditions and relevant program refinements are illustrated through a number of examples.

1 Introduction

Stepwise refinement of programs has been a successful method for the development of programs. Originally intended for sequential programs [2, 6, 8, 9, 11], the technique has also extended well to concurrent programs [1, 3, 4, 5, 7, 12, 16, 17, 20, 21, 22, 28]. Its main advantage is a separation of concerns since issues such as appropriate data structures, efficiency, and the peculiarities of the underlying architecture can be handled one at a time. Program derivation through stepwise refinement can be roughly classified into two kinds: property refinement and program refinement. While the former deals with the refinement of a set of properties, the latter deals with the refinement of a program text. This paper investigates program refinements in the context of a fair transition system [18].

A fair transition system is an abstract computational model characterized by a set of variables, a set of states, a set of transitions, initial conditions, and fairness requirements. Fair transition systems encompass most existing systems and have been widely used for modeling reactive systems. For our purposes in this paper, we choose the framework of Unity [7] as the representative fair transition system. The main reason behind our choice is the simplicity of Unity's logic and the ease with which refinements can be expressed and proved in it. Even though we concentrate on one specific formal framework, most of our results should translate across other frameworks based on fair transition systems.

* Work supported in part by NSF Grant CCR-9008628. A preliminary version of this paper appeared in [27]

We consider two kinds of program refinements – those that preserve all properties (safety and progress), and others that preserve the *fixed-point* (i.e., if the original program terminates, then so does the refined program, and the set of final states of the refined program is contained in the set of final states of the original program) [27]. The first kind of refinement, referred to as a *property preserving refinement*, is useful for reactive programs whereas the second kind of refinement, referred to as a *fixed-point preserving refinement*, is useful for terminating programs. We concentrate on a few commonly occurring program transformations, viz., data refinement, atomicity refinement, and strengthening of guards. We develop conditions under which these transformations are property preserving and fixed-point preserving. We also illustrate the applicability of these refinements through a number of small examples. Though the presented refinements are individually quite simple, together they can be quite a powerful tool in the development of programs [3, 5, 28].

Data refinement has received considerable attention in the literature. Hoare was the first to consider it in the context of implementation of abstract data types for sequential programs [10]. Subsequently, a number of authors have generalized his work to concurrent programs [4, 5, 20, 21]. Refinement of atomicity has also been investigated extensively. Lipton was the first to consider the refinement of atomicity for concurrent programs [15]. He considered the semantics of operations and developed conditions under which the refined program would preserve partial correctness. Lamport and Schneider have generalized Lipton's theorem to a larger class of safety properties besides partial correctness [14]. Back generalizes Lipton's conditions further by considering total correctness. He considers the framework of *action systems* and develops conditions under which a refinement of atomicity is fixed-point preserving.

Recently, Sanders [22] has proposed a mixed specification language that incorporates both a program text and a set of explicit program properties. Any fair execution of the program text that satisfies the explicit program properties is considered an acceptable behavior of the program. She develops the idea of refinements for mixed specifications and presents a number of interesting examples.

The rest of the paper is organized as follows. Section 2 discusses Unity and presents a brief introduction to its logic. Section 3 discusses the two kinds of program refinements and their properties. Section 4 considers three kinds of program transformations – data refinement, atomicity refinement, and strengthening of guards. A number of examples that illustrate the usefulness of these transformations are also given here. Finally, Sect. 5 includes some concluding remarks.

2 A brief introduction to Unity

We discuss the syntax of Unity in Sect. 2.1, the logic of Unity in Sect. 2.2, and program compositions in Sect. 2.3.

2.1 The Unity syntax

A Unity program consists of four sections – a *declare* section that declares the variables used in the program, an *always* section that consists of a set of proper equations, an *initially* section that describes the initial values of the variables, and an *assign* section that consists of a non-empty finite set of assignment statements. An assignment statement consists of one or more assignment components separated by $\|$. An assignment component is either an enumerated assignment or a quantified assignment. An enumerated assignment has a variable list on the left, a corresponding expression list in the middle, and a boolean expression on the right called the *guard* (which by default is *true*):

⟨variable-list⟩ := ⟨expression-list⟩ *if* ⟨guard⟩.

A quantified assignment specifies a quantification and an assignment that is to be instantiated with the given quantification; a quantification names a set of bound variables and a boolean expression (the range) satisfied by the instances of the bound variables.

Example 1. Examples of assignments are:

1. Exchange x, y, provided predicate b holds.
   $x, y := y, x$ *if* b
2. Add $A[i]$ into *sum* and increment i, provided i is less than N.
   $sum, i := sum + A[i], i+1$ *if* $i < N$ □

Example 2. Examples of quantified assignments are:

1. Assign 0 to all components of array $A[0..N]$.
   $\langle \| i : 0 \le i \le N :: A[i] := 0 \rangle$
2. Given arrays $A[0..N]$ and $B[0..N]$ of integers, assign the maximum of $A[i]$ and $B[i]$ to $A[i]$, for all $0 \le i \le N$.
   $\langle \| i : 0 \le i \le N :: A[i] := \max(A[i], B[i]) \rangle$ □

An assignment component is executed by first evaluating all expressions and then assigning the values of the evaluated expressions to the appropriate variables, if the associated boolean expression is *true*; otherwise, the variables are left unchanged.

The set of assignment statements in the *assign* section is written down either by enumerating every statement singly and using ▯ as the set constructor, or by using a quantification of the form ⟨▯*var* : *range* :: *statement*⟩. Symbol ▯ is called the Union operator.

A program execution starts from any state satisfying the initial conditions and goes on forever; in each step of execution some assignment statement is selected nondeterministically and executed. Nondeterministic selection is constrained by the following fairness rule: every statement is selected infinitely often [7].

Example 3. The following program assigns the maximum of variables x and y to variable z. Its *assign* section consists of two assignment statements each of which has one assignment component consisting of an enumerated assignment.

```
Program max
    initially $z = 0$
    assign
        $z := x$ if $x \ge y$
       ▯$z := y$ if $x \le y$
end
```

The following program sorts integer array $A[0..N]$, $N \geq 0$, in ascending order by swapping adjacent elements if they are out of order. Its *assign* section consists of N statements, one for every pair of adjacent positions.

```
Program sort
  assign
    $\langle \square i : 0 \leq i < N :: A[i], A[i+1] := A[i+1], A[i]$ if $A[i] > A[i+1] \rangle$
end                                                                          □
```

2.2 *The Unity logic*

Program properties are expressed using four relations on predicates – *unless, invariant, ensures, and leads-to.* The first two are used for stating safety properties whereas the last two are used for stating progress properties. The formal definitions of these relations are based on the idea of the *strongest invariant* of a program advocated by Sanders [23]. The strongest invariant of a program P is denoted $SI\text{-}P$ and characterizes the set of states reachable from the initial state. It is defined as the strongest solution of X in the equation

$$(INIT \Rightarrow X) \wedge (X \Rightarrow (\forall s : s \in assign : wp(s, X))).$$

Here *wp* denotes the *precondition* predicate transformer due to Dijkstra [8]. The predicate $SI\text{-}P$ denotes the set of states that are reachable initially as well as the states that are reachable after executing some statement in the *assign* section of program P. When program P is clear from the context, we write the strongest invariant simply as SI.

2.2.1 Unless. For any two predicates p and q, the property p *unless* q holds in a program iff for all statements s in the program the following implication holds:

$$SI \wedge p \wedge \neg q \Rightarrow wp(s, p \vee q).$$

Informally, if p is *true* at some point in the computation, then either q never holds and p holds forever from this point on, or q holds eventually and p continues to hold until q holds. Note the use of SI to restrict the attention to only reachable states.

Example 4

1. The value of x never decreases.
   $x = k$ *unless* $x > k$, for all k, or
   $x \geq k$ *unless false*, for all k.
2. Philosopher u stays *hungry* until *eating.*
   hungry.u unless eating.u
3. In Program *max*, variable z retains its old value until it gets $max(x, y)$.
   $z = k$ *unless* $z = max(x, y)$, for all k. □

Derived rules. The following rules can be derived from the definition of *unless* relation.

Reflexivity rule:

$$\frac{SI \Rightarrow (p \Rightarrow q)}{p \textit{ unless } q}$$

Irreflexivity rule:

$p \textit{ unless } \neg p$

Weakening rule:

$$\frac{p \textit{ unless } q,\ SI \Rightarrow (q \Rightarrow r)}{p \textit{ unless } r}$$

Conjunction and disjunction rules:

$$\frac{\langle \forall i :: p.i \textit{ unless } q.i \rangle}{\begin{array}{ll} \langle \forall i :: p.i \rangle \textit{ unless } \langle \forall i :: p.i \vee q.i \rangle \wedge \langle \exists i :: q.i \rangle & \{\text{conjunction}\}, \\ \langle \exists i :: p.i \rangle \textit{ unless } \langle \forall i :: \neg p.i \vee q.i \rangle \wedge \langle \exists i :: q.i \rangle & \{\text{disjunction}\} \end{array}}$$

For the case of two pairs of predicates, the rules simplify to the following.

$$\frac{p \textit{ unless } q,\ p' \textit{ unless } q'}{\begin{array}{ll} p \wedge p' \textit{ unless } (p \wedge q') \vee (p' \wedge q) \vee (q \wedge q') & \{\text{conjunction}\}, \\ p \vee p' \textit{ unless } (\neg p \wedge q') \vee (\neg p' \wedge q) \vee (q \wedge q') & \{\text{disjunction}\} \end{array}}$$

2.2.2 Invariant. For any predicate p, the property *invariant* p holds in a program iff p holds initially and the program never falsifies p, i.e.,

$$\textit{initially } p \ \wedge \ p \textit{ unless false.}$$

Alternatively, an *invariant* of a program is any predicate that is implied by its strongest invariant.

Example 5

1. Variable x is always positive.
   $\textit{invariant } x \geq 0$
2. An eating philosopher u has all the required forks.
   $\textit{invariant eating}.u \Rightarrow \textit{hasforks}.u$ □

2.2.3 Ensures. For any two predicates, p and q, the property p *ensures* q holds in a program iff p *unless* q holds in the program and there exists a statement s in the program such that

$SI \wedge p \wedge \neg q \Rightarrow wp(s, q).$

Thus, if p is *true* at some point in the computation then q holds eventually and p continues to hold until q holds.

Derived rules. The following rules can be derived from the definition of *ensures* and *unless* relations.

Reflexivity rule:

$$\frac{\textit{invariant } p \Rightarrow q}{p \textit{ ensures } q}$$

Weakening rule:

$$\frac{p \textit{ ensures } q,\ \textit{invariant } q \Rightarrow r}{p \textit{ ensures } r}$$

Conjunction rule:

$$\frac{p \textit{ unless } q,\ p' \textit{ ensures } q'}{p \wedge p' \textit{ ensures } (p \wedge q') \vee (p' \wedge q) \vee (q \wedge q')}$$

2.2.4 Leads-to. The relation *leads-to* is denoted as $\mapsto$, and is defined to be the strongest relation satisfying the following three rules.

- $p \textit{ ensures } q \Rightarrow p \mapsto q$,
- $(p \mapsto q \wedge q \mapsto r) \Rightarrow p \mapsto r$, and
- For any set W,
  $(\forall m : m \in W : p.m \mapsto q) \Rightarrow ((\exists m : m \in W : p.m) \mapsto q)$.

The first two rules imply that $\mapsto$ includes the transitive closure of *ensures* and the third rule allows us to induct over sets. Given that $p \mapsto q$ in a program, we can assert that once p becomes *true*, eventually q becomes *true*. However, unlike p *ensures* q, we cannot assert that p will remain *true* as long as q is *false*.

Example 6

1. A hungry philosopher u eventually *eats*.
   $hungry.u \mapsto eating.u$
2. If a message m is sent, then it is eventually received.
   $send.m \mapsto receive.m$
3. In Program max,
   $true \mapsto z = \max(x, y)$. □

Derived rules. The following rules can be derived from the definiton of *leads-to* and other relations.

Reflexivity rule:

$$\frac{\textit{invariant } p \Rightarrow q}{p \mapsto q}$$

Strengthening rule:

$$\frac{\textit{invariant } p \Rightarrow q,\ q \mapsto r}{p \mapsto r}$$

Weakening rule:

$$\frac{p \mapsto q,\ \textit{invariant } q \Rightarrow r}{p \mapsto r}$$

Impossibility rule:

$$\frac{p \mapsto \textit{false}}{\textit{invariant } \neg p}$$

Completion rule:

$$\frac{(\forall i::p.i \mapsto q.i,\ q.i\ \mathit{unless}\ b)}{(\forall i::p.i) \mapsto (\forall i::q.i) \vee b}$$

PSP (Progress-Safety-Progress) theorem:

$$\frac{p \mapsto q,\ r\ \mathit{unless}\ b}{p \wedge r \mapsto (q \wedge r) \vee b}$$

Induction rule:

$$\frac{\begin{array}{c}(\forall k::p \wedge x=k \mapsto (p \wedge x<k) \vee q)\\ \text{The domain of } x \text{ is well-founded under } <.\end{array}}{p \mapsto q} \quad \square$$

2.2.5 Fixed point. The fixed-point of a program P, usually represented by $FP\text{-}P$, denotes the collection of states that are invariant under the execution of every statement of the program. It is obtained by replacing the assignment symbol $:=$ by the equality symbol $=$ in every statement of the program and taking the conjunction over all such predicates. For example, the fixed-point of program max in Example 3 is

$$(x \geq y \Rightarrow z=x) \wedge (x \leq y \Rightarrow z=y).$$

Similarly, the fixed-point of program *sort* is

$$(\forall i::A[i] > A[i+1] \Rightarrow A[i]=A[i+1]),$$

which simplifies to $(\forall i::A[i] \leq A[i+1])$. When program P is clear from the context, we write the fixed-point simply as FP. In general, the fixed-point of a program includes states that are not reachable by the program. The set of fixed-point states that are reachable is obtained by taking the conjunction of the fixed-point and the strongest invariant of the program; for a program P, this is represented by the predicate $FP\text{-}P \wedge SI\text{-}P$.

Derived rules. The following rules can be derived from the definition of fixed-point and other relations.

Stability at fixed-point:

$FP \wedge p\ \mathit{unless}\ \mathit{false}$

Point predicate[1] rule 1:

For any point predicate p,

$p\ \mathit{unless}\ \mathit{false} \equiv \mathit{invariant}\ p \Rightarrow FP$

Point predicate rule 2:

For any point predicate p,

$p \mapsto \neg p \equiv \mathit{invariant}\ (p \Rightarrow \neg FP)$ $\square$

The preceding definitions of *unless* and *ensures* rely on the definition of the strongest invariant of a program. However, when reasoning about a program

[1] A *point predicate* is a predicate that is true at exactly one point in the state space.

we are seldom interested in computing the strongest invariant. One way to infer program properties without calculating the strongest invariant explicitly is through the use of the *substitution axiom* [7]. This axiom allows a program invariant I to be replaced by *true* and vice-versa in the context of the program. The validity of this axiom stems from the fact that the strongest invariant of a program implies any invariant of the program. Consequently, any property inferred by applying the substitution axiom can also be inferred by a direct application of the definition of the *unless, ensures*, and *leads-to* relations [23].

2.3 *Program composition*

Program composition in Unity is achieved either by union or superposition.

2.3.1 Program composition by Union. Let F and G be programs with compatible *declare* sections (i.e., the declarations of the variables are non-conflicting), compatible *always* sections (i.e., the two sets of equations are consistent), and compatible *initially* sections (i.e., the initial values of the variables are non-conflicting). Then, their composition is a new program denoted $F \square G$; every section of this program is obtained by a union of the corresponding sections of F and G. Since the states reachable by a program may change when it is composed with another program, reasoning about a composite program requires considerable caution. For example, it may be incorrect to infer the property *p unless q* for a composite program if it holds for each of the individual programs. We solve this problem by introducing the notion of a context [16]. The context of a program F is another program representing the environment in which program F executes. All program properties are specified relative to a context, which is specified within a pair of square brackets following the statement of the property. For example, *p unless q in F* [*G*] denotes that program F satisfies the property *p unless q* when placed in a context G. The formal definition of program properties relative to a context mirrors the definition of the properties without a context. For example, the *unless* and *ensures* properties are defined as follows.

$$\begin{aligned}&p \textit{ unless } q \text{ in } F[G] \equiv (\forall s : s \in F : SI\text{-}(F \square G) \wedge p \wedge \neg q \Rightarrow wp(s, p \vee q))\\&p \textit{ ensures } q \text{ in } F[G] \equiv p \textit{ unless } q \text{ in } F[G] \wedge (\exists s : s \in F : SI\text{-}(F \square G) \wedge p \wedge \neg q \Rightarrow\\&\qquad wp(s, q))\end{aligned}$$

Note that SI-$(F \square G)$ denotes the strongest invariant of the composite program $F \square G$. Its usage in the above definition amounts to considering all program states that are reachable by the actions of F and G as opposed to considering only the states reachable by actions of F. All the theorems stated earlier for *unless, ensures, and leads-to* continue to hold for the above definitions. A special program *nil* is defined to be the identity of composition, i.e., $F \square nil = nil \square F = F$. The definitions of *unless* and *ensures* in Sect. 2.1 follow as a special case of the above definitions under the substitution $G := nil$. Henceforth, we assume that the assertion *P in F* is an abbreviation for the assertion *P in F* [*nil*].

The substitution axiom can be used to rewrite a program property with the help of an assertion I only if I is an invariant of the composition of the program and the context mentioned in the property. Reasoning about a composite program is carried out by the following restatement of the *union theorem* [7].

p *unless* q in $(F \square G)[H] \equiv p$ *unless* q in $F[G \square H] \wedge p$ *unless* q in $G[F \square H]$,
and

p *ensures* q in $(F \square G)[H] \equiv (p$ *ensures* q in $F[G \square H] \wedge p$ *unless* q in $G[F \square H])$
$\vee (p$ *unless* q in $F[G \square H] \wedge p$ *ensures* q in $G[F \square H])$.

Leads-to properties do not compose in general. However, the following theorem defines a limited composition of these properties [25].

Composition of *leads-to*:

$$\frac{p \mapsto q \text{ in } F[G],\ r \Rightarrow FP\text{-}G}{p \mapsto q \vee \neg r \text{ in } F \square G}$$

2.3.2 Program composition by superposition. Superposition is another mechanism to structure programs in Unity. Suppose we are given a program F and a statement r that does not assign to any of the variables of F. Then, the statement r can be superposed on program F in two ways – either it can be combined with a statement s of F to yield an augmented statement $s \| r$, or it can be added by itself to F, thus resulting in the composite program $F \square r$. In either case, all the *unless*, *ensures*, and *leads-to* properties of the original program are preserved. Moreover, the fixed-point of the transformed program implies the fixed-point of the original program and all invariants of the original program are preserved. This result is referred to as the *superposition theorem.*

3 Program refinement in Unity

There are two kinds of program refinements in Unity: those that preserve all *unless* (safety) and *leads-to* (progress) properties [7], and others that preserve the fixed-point (i.e., if the original program terminates, then so does the refined program and moreover, the fixed-point of the refined program implies the fixed-point of the original program). The first kind of refinement is useful for reactive programs whereas the second kind is useful for terminating programs. Formally, the two refinements are defined as follows.

Let F and G be two programs. We say that G is a *property preserving* refinement of F iff for all predicates p, q the following two assertions hold.

- $(p$ *unless* q holds in $F) \Rightarrow (p$ *unless* q holds in $G)$, and
- $(p \mapsto q$ holds in $F) \Rightarrow (p \mapsto q$ holds in $G)$.

Note that the preservation of *ensures* properties is not required in this definition. This is because *leads-to* (and not *ensures*) is used as the basic relation for specifying progress properties, and preservation of *leads-to* is all that is required in most situations. Similarly, we say that G is a *fixed-point preserving* refinement of F iff the following two assertions hold.

- $(true \mapsto FP\text{-}F$ in $F) \Rightarrow (true \mapsto FP\text{-}G$ in $G)$, and
- $(FP\text{-}G \wedge SI\text{-}G) \Rightarrow (FP\text{-}F \wedge SI\text{-}F)$.

An invariant of a program provides a simple way to obtain a refinement that is both property preserving and fixed-point preserving. Suppose we have a program F and an invariant $f = g$ that holds in the program. Then, any occurrence of f in the text of the program can be replaced by g to obtain a program

G that is a property preserving and fixed-point preserving refinement of F. The explanation is as follows. Since the invariant $f=g$ holds at all the reachable states, all the program properties of F continue to hold in G. This implies that G is a property preserving refinement of F. It is also possible to show that $SI\text{-}F \wedge FP\text{-}F \equiv SI\text{-}G \wedge FP\text{-}G$. Consequently, if $true \mapsto FP\text{-}F$, or equivalently, $true \mapsto SI\text{-}F \wedge FP\text{-}F$ in program F then we can assert the following in program G:

$true \mapsto SI\text{-}F \wedge FP\text{-}F$, or equivalently,
$true \mapsto SI\text{-}G \wedge FP\text{-}G$, or equivalently,
$true \mapsto FP\text{-}G$.

This implies that G is a fixed-point preserving refinement of F. We refer to the above result again as the substitution axiom since we are now substituting in a program's text instead of in a program's properties.

It is not too difficult to construct fixed-point preserving refinements that are not property preserving. The existence of property preserving refinements that are not fixed-point preserving is more interesting. The relationship between the two kinds of refinements is considered next.

Theorem 1. *A property preserving refinement that does not introduce any new variables (i.e., does not modify the state space) is also fixed-point preserving.*

Proof. Consider a program F and its property preserving refinement G. Let p be a point predicate such that $p \Rightarrow (SI\text{-}F \wedge SI\text{-}G)$. Now, observe the following.

$p \Rightarrow FP\text{-}F$
$\Rightarrow$ {stability at fixed-point}
p *unless false* in F
$\Rightarrow$ {G preserves all *unless* properties of F}
p *unless false* in G
$=$ {point predicate rule 1, $p \Rightarrow SI\text{-}G$}
$p \Rightarrow FP\text{-}G$
$=$ {point predicate rule 2, $p \Rightarrow SI\text{-}G$}
$\neg (p \mapsto \neg p$ in $G)$
$\Rightarrow$ {G preserves all *leads-to* properties of F}
$\neg (p \mapsto \neg p$ in $F)$
$=$ {point predicate rule 2, $p \Rightarrow SI\text{-}F$}
$p \Rightarrow FP\text{-}F$

Thus, $(SI\text{-}F \wedge SI\text{-}G) \Rightarrow (FP\text{-}F \equiv FP\text{-}G)$. Next, we show that $SI\text{-}G \Rightarrow SI\text{-}F$. The first condition for fixed-point preserving refinements then follows because G preserves all *leads-to* properties of F and the second condition for fixed-point preserving refinements follows from predicate calculus.

true
$\Rightarrow$ {property of $\mapsto$}
false $\mapsto$ *false* in F
$\Rightarrow$ {$SI\text{-}F$ is invariant in F}
$\neg\, SI\text{-}F \mapsto false$ in F
$\Rightarrow$ {G preserves all *leads-to* properties of F}
$\neg\, SI\text{-}F \mapsto false$ in G

$\Rightarrow$ {impossibility theorem}
invariant $SI\text{-}F$ in G
$\Rightarrow$ {definition of strongest invariant}
$SI\text{-}G \Rightarrow SI\text{-}F$

This completes the proof. □

Example 7. For an example of a property preserving refinement that is not fixed-point preserving, consider any terminating program F and add to it a statement $t::x:=x+1$, where x is a fresh variable. It follows from the Superposition theorem that program $F \square t$ is a property preserving refinement of F. However, since $F \square t$ does not terminate, program $F \square t$ is not a fixed-point preserving refinement of F. □

The following theorem also relates the two kinds of refinements.

Theorem 2. *If G is a property preserving refinement of F and if both F and G terminate, then G is also a fixed-point preserving refinement of F.*

Proof. The first condition of fixed-point preserving refinements is satisfied because G terminates. For the second condition, we have to show that

$$(FP\text{-}G \wedge SI\text{-}G) \Rightarrow (FP\text{-}F \wedge SI\text{-}F).$$

Observe the following.

$true \mapsto FP\text{-}F$ in F	, assumption
$true \mapsto (FP\text{-}F \wedge SI\text{-}F)$ in F	, $SI\text{-}F$ is invariant in F
$true \mapsto (FP\text{-}F \wedge SI\text{-}F)$ in G	, G preserves *leads-to* properties of F
$\neg (FP\text{-}F \wedge SI\text{-}F) \mapsto (FP\text{-}F \wedge SI\text{-}F)$ in G	, strengthening
$FP\text{-}G \wedge \neg (FP\text{-}F \wedge SI\text{-}F)$ *unless false* in G	, stability at fixed-point
$FP\text{-}G \wedge \neg (FP\text{-}F \wedge SI\text{-}F) \mapsto$ *false* in G	, PSP theorem on above two
invariant $FP\text{-}G \Rightarrow (FP\text{-}F \wedge SI\text{-}F)$ in G	, impossibility theorem
$SI\text{-}G \Rightarrow (FP\text{-}G \Rightarrow (FP\text{-}F \wedge SI\text{-}F))$	, definition of strongest invariant
$(FP\text{-}G \wedge SI\text{-}G) \Rightarrow (FP\text{-}F \wedge SI\text{-}F)$	, predicate calculus □

Theorem 3. *The relations "is a property preserving refinement of" and "is a fixed-point preserving refinement of" are preorders (i.e., reflexive and transitive).*

Proof. Follows from the reflexivity and transitivity of implication. □

4 Some useful program refinements

In this section we discuss some program refinements that are useful in the formal derivation of programs. In Sect. 4.1 we consider the subject of data refinements. In Sect. 4.2 we consider the strengthening of guards. Finally, in Sect. 4.3 we consider the question of atomicity refinement.

4.1 Data refinement

Data refinement is a very effective tool in program derivation as it provides a programmer the freedom to express an algorithm using a convenient abstract data type. Later, the chosen abstract data type can be implemented by an available data type on the target machine while preserving the correctness of the original algorithm. In this section we develop conditions under which data refinement can be carried out for reactive programs.

Let $F\square s$ be a program using a variable x of abstract data type X. Assume that statement $s::x:=f(x)$ *if* $p(x)$, which performs operation f on the abstract object x provided guard p holds, is the *only* statement modifying x. We wish to examine conditions under which variable x can be implemented by a fresh variable y of the concrete data type Y. For this purpose, let $t::y:=g(y)$ *if* $q(y)$ be a statement that performs operation g on concrete object y provided guard q holds. Intuitively, in order for program $F\square t$ to *simulate* program $F\square s$, there should exist a function h (called the abstraction function) from Y to X such that $h(y)$ equals x at all times. Before we state the theorem, we discuss some preliminaries.

Notation. For any expression e, define e' to be the expression obtained by a syntactic substitution of term x by the term $h(y)$ and define F' to be the program obtained by syntactically substituting x by $h(y)$ everywhere. We say that program G is a property preserving refinement of program F *under the coupling* $x=h(y)$ if the following two conditions hold for all predicates b, c.

- (b *unless* c holds in F) $\Rightarrow$ (b' *unless* c' holds in G), and
- ($b\mapsto c$ holds in F) $\Rightarrow$ ($b'\mapsto c'$ holds in G).

Similarly, we say that program G is a fixed-point preserving refinement of program F *under the coupling* $x=h(y)$ if the following two conditions hold.

- ($true\mapsto FP\text{-}F$ in F) $\Rightarrow$ ($true\mapsto FP\text{-}G$ in G), and
- ($FP\text{-}G \wedge SI\text{-}G \wedge x=h(y)$) $\Rightarrow$ ($FP\text{-}F \wedge SI\text{-}F$). $\square$

Define conditions $A0$, $A1$, and $A2$ as follows.

- $\{h(m)\,|\,m$ is an initial value for $y\}=\{n\,|\,n$ is an initial value for $x\}$. ... ($A0$)
- $p(h(y)) \Rightarrow h(g(y))=f(h(y))$, for all y. ... ($A1$)
- $p(h(y)) \equiv q(y)$, for all y. ... ($A2$) $\square$

Theorem 4. *Program $F'\square t$ is a property preserving refinement of $F\square s$ under the coupling $x=h(y)$ if conditions $A0$, $A1$, and $A2$ hold. Furthermore, if there exists a function r from the program variables to a well-founded set such that*

$$FP\text{-}(F\square s) \wedge p(x) \wedge g(y) \neq y \wedge r=k \Rightarrow wp(y:=g(y), r<k) \quad \dots (A3)$$

then $F'\square t$ is a fixed-point preserving refinement as well. $\square$

Conditions $A0$–$A2$ ensure that $h(y)$ equals x at all times in the transformed program. Condition $A3$ ensures that if the original program terminates then so does the transformed program (function r defines an upper bound on the number of state transitions).

Proof. Let statement u be defined as $y:=g(y)$ *if* $p(x)$. The proof is in four steps. In the first step, we show that program $F\square(s\|u)$ is a property preserving refinement of $F\square s$. We also show that $F\square(s\|u)$ is a fixed-point preserving refinement of $F\square s$ if Condition $A3$ holds. In the second step, we replace x by $h(y)$ in the text of F and u based on conditions $A0$ and $A1$ and show that the resulting program $F'\square(s\|u')$ is a property and fixed-point preserving refinement of program $F\square(s\|u)$. In the third step, we use Condition $A2$ and the substitution axiom to obtain a property and fixed-point preserving refinement $F'\square(s\|t)$. Finally, in the fourth step, we delete statement s and show that the resulting program $F'\square t$ is a property and fixed-point preserving refinement under the coupling $x=h(y)$. The desired theorem then follows from the transitivity of the refinements.

Step 1. From the superposition theorem $F\square(s\|u)$ is a property preserving refinement of $F\square s$. It remains to prove that $F\square(s\|u)$ is a fixed-point preserving refinement of $F\square s$. Let FP_0 be the fixed-point of $F\square s$ and let $FP_1 \equiv FP_0 \wedge FP\text{-}u$ be the fixed-point of program $F\square(s\|u)$. Clearly, FP_1 implies FP_0. Since superposition preserves all invariants, the strongest invariant of $F\square(s\|u)$ implies the strongest invariant of $F\square s$. This implies that the second condition for fixed-point preserving refinements is satisfied. Our remaining proof obligation is to show that if $F\square s$ terminates then so does $F\square(s\|u)$.

Assume that $true \mapsto FP_0$ in $F\square s$. Since *leads-to* properties are preserved in $F\square(s\|u)$ and since *leads-to* is transitive, it will suffice to show that $FP_0 \mapsto FP_1$ in $F\square(s\|u)$. Observe the following.

$$\begin{array}{ll}
FP_0 \wedge \neg FP\text{-}u \wedge r=k \Rightarrow wp(y:=g(y),\, r<k) & \text{, property } A3 \\
FP_0 \wedge \neg FP\text{-}u \wedge r=k \Rightarrow wp(u,\, r<k) & \text{, definition of statement } u \\
FP_0 \wedge \neg FP\text{-}u \wedge r=k \Rightarrow wp(u,\, FP_0 \wedge r<k) & \text{, } u \text{ does not modify } FP_0 \\
FP_0 \wedge \neg FP\text{-}u \wedge r=k \Rightarrow wp(s\|u,\, FP_0 \wedge r<k) & \text{, stability at fixed point} \\
\multicolumn{2}{l}{(\forall v : v\in F : FP_0 \wedge \neg FP\text{-}u \wedge r=k \Rightarrow wp(v, r<k \vee (FP_0 \wedge \neg FP\text{-}u \wedge r=k)))\ldots(A4)} \\
 & \text{, stability at fixed point} \\
FP_0 \wedge \neg FP\text{-}u \wedge r=k \textit{ ensures } FP_0 \wedge r<k \text{ in } F\square(s\|u) & \text{, definition of } \textit{ensures} \\
FP_0 \wedge \neg FP\text{-}u \wedge r=k \mapsto FP_0 \wedge r<k \text{ in } F\square(s\|u) & \text{, definition of } \mapsto \\
FP_0 \wedge \neg FP_1 \wedge r=k \mapsto FP_0 \wedge r<k \text{ in } F\square(s\|u) & \text{, definition of } FP_1 \\
FP_0 \wedge \neg FP_1 \wedge r=k \mapsto (FP_0 \wedge \neg FP_1 \wedge r<k) \vee FP_1 & \text{, consequent weakening} \\
FP_0 \wedge \neg FP_1 \mapsto FP_1 & \text{, induction over } k \\
FP_0 \wedge FP_1 \mapsto FP_1 & \text{, reflexivity} \\
FP_0 \mapsto FP_1 & \text{, disjunction}
\end{array}$$

Step 2. We first prove that $h(y)=x$ is an invariant of program $F\square(s\|u)$.

$$\begin{array}{ll}
h(y)=x \Rightarrow f(h(y))=f(x) & \text{, Leibniz's rule} \\
p(x) \wedge h(y)=x \Rightarrow f(h(y))=f(x) & \text{, predicate calculus} \\
p(x) \wedge h(y)=x \Rightarrow h(g(y))=f(x) & \text{, condition } A1 \\
h(y)=x \Rightarrow wp(s\|u,\, h(y)=x) & \text{, definitions of } s \text{ and } u \\
(\forall u : v\in F : h(y)=x \Rightarrow wp(v,\, h(y)=x)) & \text{, } F \text{ does not modify } x, y \\
h(y)=x \textit{ unless false} \text{ in } F\square(s\|u) & \text{, definition of } \textit{unless} \\
\textit{invariant } h(y)=x \text{ in } F\square(s\|u) & \text{, condition } A0
\end{array}$$

Next, we substitute using invariant $h(y)=x$ in the text of program F and statement u. The resulting program is $F'\square(s\|u')$. It follows from the substitution axiom that this refinement is a property and fixed-point preserving refinement of program $F\square(s\|u)$.

Steps 3 and 4. Shown in the appendix. (End of Proof)

Though Theorem 4 assumes y to be a single variable, it can be easily extended to the case of y being a vector of variables. The theorem is motivated by Hoare's correctness conditions for implementation of abstract data types for sequential programs [10]. As in there, we assume that the abstract state is functionally dependent on the concrete state. This functional dependence has been later generalized to an arbitrary *abstraction relation* between the abstract and the concrete states [6, 20, 21].

Observation 1. A simple way to satisfy condition $A3$ is to ensure that

$$p(h(y)) \Rightarrow (h(g(y)) = h(y) \Rightarrow g(y) = y),$$

for then the antecedent of Condition $A3$ reduces to false. $\square$

If the program under consideration includes mutiple statements s_i that modify the abstract variable, then it can be refined by including a new statement t_i corresponding to each abstract statement s_i. The correctness conditions for the refinement now include Condition $A0$, and a set of conditions $A1$–$A3$ corresponding to each pair of statements (s_i, t_i). (All the conditions corresponding to $A3$ should however refer to the same function r). The proof is once more in four steps. In the first step statements $u.i$, one for each s_i, are added to the program. The proof of correctness of this step is same as before except that in property $A4$ the universal quantification has also to range over statements $(s_j \| u.j)$, $j \neq i$. The proof obligation for these new statements is easily met because each statement s_j is at a fixed point and statement $u.j$ can only decrease r at each state change. The remaining three steps and their proofs of correctness are similar to the proofs provided for Theorem 4.

Given a property preserving refinement of an abstract object x by a concrete object y through an abstraction function h, it is possible to add a statement $t::y := g(y)$ *if* $q(y)$, and obtain a property preserving refinement provided $q(y) \Rightarrow h(g(y)) = h(y)$ for all y. Statement t refines a *skip* statement $s::x := x$ *if* $q(y)$ of the abstract program. Condition $A1$ is satisfied because f is the identity function and $q(y) \Rightarrow h(g(y)) = h(y)$. Condition $A2$ is satisfied vacuously. Such statements that preserve $h(y)$ by merely restructuring the concrete object are called *restructuring* statements. As an example, consider implementing a bank account x by a checking account y and a savings account z through the abstraction function $h(y, z) = y + z$, i.e., the sum of accounts y and z is meant to simulate abstract account x. In this case, if conditions $A0$–$A2$ (which link program variables x, y and z and are needed for the proof of correctness of the refinement) hold, then a restructuring statement $s::y, z := y-1, z+1$ *if* $y \geq 1$ that transfers one dollar from y to z can be added without affecting the correctness of the refinement. The addition of restructuring statements allows the concrete object to repeat a state by stuttering without changing the abstract representation [1, 13].

Restructuring statements cannot be added freely in the case of fixed-point preserving refinements because the new program may never reach a fixed-point. For example, for the banking account instance if we add a restructuring statement $u::y, z := y+1, z-1$ *if* $z \geq 1$ in addition to statement t, then the new program may not terminate even if the original program terminates. This is because there is no function that bounds the number of state changes before the new program reaches a fixed-point.

Example 8. Consider the following program in which variable x is a natural number.

```
Program simple
    initially $x=0$
    assign
        $x:=x+1$ if $x>5$
        $\square z:=x^2$
end
```

We wish to replace variable x by a fresh variable y of the type *queue*. The abstraction function h that we choose here maps a queue to the number of elements in the queue, i.e., $h(y)=size(y)$.

The incrementing of x is replaced by the appending (the symbol ';' denotes concatenation) of some arbitrary element e to the queue. Condition $A0$ is satisfied by setting y to *null* initially. Condition $A1$ is satisfied because $size(y;\ e)=size(y)+1$. Condition $A2$ is satisfied vacuously. Condition $A3$ is satisfied because $size(y;\ e)\neq size(y)$ (see Observation 1). Thus, we obtain the following refined program that is a property and fixed-point preserving refinement.

```
Program simple
    initially $y=null$
    assign
        $y:=y;\ e$ if $size(y)>5$
        $\square z:=(size(y))^2$
end                          □
```

Example 9. In this example taken from [7], we consider replacing a shared variable by unbounded FIFO channels. Consider a variable x that is shared between two processes F and G. Process F accesses x only by statement s and process G accesses x only by statement t; these statements are defined as follows:

$s::x:=x\oplus d$ *if* p, and
$t::vs,\ x:=f(vs,\ x),\ g(x)$ *if* $b(x)\wedge q$.

Variable vs represents local variables of G. Both x and d are assumed to be of the same type and $\oplus$ is an arbitrary function of the type $X\times X\to X$. It is further assumed that predicates p, q do not mention x. It is apparent from examining the statements that process F only modifies x while process G tests, reads, and modifies x.

We wish to replace the shared variable x by two variables: one, a channel from process F to G called c and the other, a local copy of x at G called y. Thus, $type(y)=X$ and $type(c)=X^*$. We wish to transform statement s to a statement s' in which variable d is appended to the channel variable c. Similarly, we wish to transform statement t to a statement t' in which process G accesses variables y and c instead of variable x.

$s'::c:=c;\ d$ *if* p, and
$t'::vs,\ y:=f(vs,\ y\oplus c),\ g(y)$ *if* $b(y\oplus c)\wedge q$.

The question then arises: under what conditions does this transformation preserve all *unless* and *leads-to properties?* The answer lies in the conditions $A0$, $A1$, and $A2$ of Theorem 4 presented earlier. Here x represents the abstract object and the pair $(y,\ c)$ represents the concrete object. We choose an abstraction function h as follows:

$h(y, c) = y \oplus c,$

where function $\oplus$ is generalized to accept strings in its second argument as follows:

$$y \oplus c = \begin{cases} y & \text{if } c = null \\ (y \oplus head(c)) \oplus tail(c) & \text{otherwise.} \end{cases}$$

Condition $A0$ of Theorem 4 is satisfied by choosing initial values for y and c such that initial value of x = initial value of $y \oplus c$. Condition $A2$ of the theorem is satisfied as $b(y \oplus c) = b(h(y, c))$. Henceforth, we concentrate on the satisfaction of condition $A1$. In order to satisfy it, we have to show that

1. $h(y, c; d) = h(y, c) \oplus d$, and
2. $b(y \oplus c) \Rightarrow h(g(y), c) = g(h(y, c))$.

The proof of satisfaction of the first condition is as follows:

$$\begin{aligned} & h(y, c; d) \\ = & \{\text{definition of } h\} \\ & y \oplus (c; d) \\ = & \{\text{definition of } \oplus, \text{ induction}\} \\ & (y \oplus c) \oplus d \\ = & \{\text{definition of } h\} \\ & h(y, c) \oplus d \end{aligned}$$

In order to prove that the second condition is satisfied, we assume property $B0$ defined as follows:

$$b(x \oplus c) \Rightarrow (g(x \oplus c) = g(x) \oplus c), \text{ for all } x, c. \quad \ldots (B0)$$

Based on this property, the proof of the second condition is as follows.

$$\begin{aligned} & \text{second condition} \\ = & \{\text{definition}\} \\ & b(y \oplus c) \Rightarrow (h(g(y), c) = g(h(y, c))) \\ = & \{\text{definition of } h\} \\ & b(y \oplus c) \Rightarrow (g(y) \oplus c = g(y \oplus c)) \\ = & \{\text{assumption } B0\} \\ & true \end{aligned}$$

This proves that the transformation of statements s, t to statements s', t' preserves all *unless* and *leads-to* properties if Condition $B0$ is satisfied. Note that property $B0$ follows from the simpler property $B0'$ defined below. (The proof is by induction on the length of sequence c.)

$$g(x \oplus d) = g(x) \oplus d \quad \ldots (B0')$$

Having completed and proved the first refinement, we now add a restructuring statement u to the transformed program.

$u :: y, c := y \oplus head(c), tail(c)$ *if* $c \neq null$

This transformation fulfills the conditions of restructuring statements because

$$c \neq null \Rightarrow h(y, c) = h(y \oplus head(c), tail(c)).$$

At this point let us recapitulate what we have done so far. We set out with the task of replacing shared variable x by an asynchronous channel from F to G. Statement s' in process F represents the transmission of a data item to channel c and so fits in with the message passing paradigm. Statement u (which is a part of process G) represents the reception of a data item (along with an update of local variable y) and therefore, also fits in with the message passing paradigm. However statement t' (which is a part of process G) is not yet in the right form as it mentions the channel variable c. Thus, this statement will require further transformation. We will return to this example in the next subsection after we have stated and proved another theorem about program transformations. $\square$

4.2 *Strengthening of guards*

Strengthening the guard of a statement obviously preserves all safety properties of a program because any state that is reachable in the refined program is also reachable in the original program. In this section we develop conditions under which this program transformation preserves other desired program properties.

Notation. Given a program, we say that a function g of the program variables is *non-increasing* if the execution of any statement of the program does not increase the value of g, i.e., $g=k$ *unless* $g<k$. $\square$

Theorem 5. *Let F be a program and let $s::A$ if p be a statement. Let statement $t::A$ if $p \wedge q$ be obtained by strengthening the guard of statement s. Then, program $F \square t$ is a property and a fixed-point preserving refinement of the program $F \square s$ if the following two conditions hold in F in the context of statement s.*

- $p \mapsto q$
- There exists a non-increasing function g from the program variables to a well-founded set such that

$$(g=k \wedge q) \textit{ unless } (\neg p \vee g<k), \text{ for all } k. \quad \square$$

Proof. The proof is in four parts: first, we show that $F \square t$ preserves all the safety properties of $F \square s$; second, we show that $F \square t$ preserves all the progress properties of $F \square s$; next, we show that the conjunction of the fixed-point and the strongest invariant of $F \square t$ implies the conjunction of the fixed-point and the strongest invariant of $F \square s$; finally, we show that if $F \square t$ terminates then $F \square s$ also terminates. Before we proceed with the proof, we note that since statement t is obtained by strengthening the guard of statement s, $SI\text{-}(F \square t) \Rightarrow SI\text{-}(F \square s)$ and $FP\text{-}(F \square s) \Rightarrow FP\text{-}(F \square t)$.

Part 1.

$$\begin{array}{ll}
b \textit{ unless } c \text{ in } F \square s & \text{, assumption} \\
(\forall u : u \in F \square s : SI\text{-}(F \square s) \wedge b \wedge \neg c \Rightarrow wp(u, b \vee c)) & \text{, definition of } \textit{unless} \\
(\forall u : u \in F \square t : SI\text{-}(F \square s) \wedge b \wedge \neg c \Rightarrow wp(u, b \vee c)) & \text{, } t \text{ has a stronger guard than } s \\
(\forall u : u \in F \square t : SI\text{-}(F \square t) \wedge b \wedge \neg c \Rightarrow wp(u, b \vee c)) & \text{, } SI\text{-}(F \square t) \Rightarrow SI\text{-}(F \square s) \\
b \textit{ unless } c \text{ in } F \square t & \text{, definition of } \textit{unless}
\end{array}$$

Part 2. Shown in the appendix.

Part 3.

$p \wedge \neg q \Rightarrow FP\text{-}t$	, observing t
$FP\text{-}F \wedge p \wedge \neg q \Rightarrow FP\text{-}F \wedge FP\text{-}t$	, predicate calculus
$FP\text{-}F \wedge p \wedge \neg q \Rightarrow FP\text{-}(F \square t)$	, $FP\text{-}(F \square t) \equiv FP\text{-}F \wedge FP\text{-}t$
$FP\text{-}F \wedge p \wedge \neg q$ *unless false* in $F \square t$	, stability at fixed-point
$p \mapsto q$ in $F \square t$	, property $C5$ in the appendix
$FP\text{-}F \wedge p \wedge \neg q \mapsto$ *false* in $F \square t$	, PSP theorem
$SI\text{-}(F \square t) \Rightarrow \neg(FP\text{-}F \wedge p \wedge \neg q)$	, impossibility theorem ... $(C7)$

Let R denote the fixed point of assignment A that is a part of statements s and t. Now observe the following.

$$\begin{aligned}
& FP\text{-}(F \square t) \wedge SI\text{-}(F \square t) \\
= & \{\text{definition of } FP\text{-}(F \square t)\} \\
& FP\text{-}F \wedge FP\text{-}t \wedge SI\text{-}(F \square t) \\
= & \{FP\text{-}t \equiv (p \wedge q \Rightarrow R)\} \\
& FP\text{-}F \wedge (p \wedge q \Rightarrow R) \wedge SI\text{-}(F \square t) \\
= & \{\text{predicate calculus}\} \\
& FP\text{-}F \wedge ((p \wedge \neg q) \vee (p \Rightarrow R)) \wedge SI\text{-}(F \square t) \\
= & \{FP\text{-}s \equiv (p \Rightarrow R)\} \\
& FP\text{-}F \wedge ((p \wedge \neg q) \vee FP\text{-}s) \wedge SI\text{-}(F \square t) \\
= & \{\text{predicate calculus}\} \\
& (FP\text{-}F \wedge p \wedge \neg q \wedge SI\text{-}(F \square t)) \vee (FP\text{-}F \wedge FP\text{-}s \wedge SI\text{-}(F \square t)) \\
= & \{\text{simplifying using } C7\} \\
& FP\text{-}F \wedge FP\text{-}s \wedge SI\text{-}(F \square t) \\
= & \{\text{definition of } FP\text{-}(F \square s)\} \\
& FP\text{-}(F \square s) \wedge SI\text{-}(F \square t) \\
\Rightarrow & \{SI\text{-}(F \square t) \Rightarrow SI\text{-}(F \square s)\} \\
& FP\text{-}(F \square s) \wedge SI\text{-}(F \square s) \quad \text{(End of Part 3)}
\end{aligned}$$

Part 4.

$true \mapsto FP\text{-}(F \square s)$ in $F \square s$	, assumption
$true \mapsto FP\text{-}(F \square s)$ in $F \square t$	, preservation of *leads-to*
$true \mapsto FP\text{-}(F \square t)$ in $F \square t$	, $FP\text{-}(F \square s) \Rightarrow FP\text{-}(F \square t)$ (End of Proof)

Corollary 1. *Let statement s be A if p, statement t be A if* $p \wedge q$, *and F be any program. Then, program* $F \square t$ *is a property and fixed-point preserving refinement of* $F \square s$ *if the following two conditions hold in F in the context of statement s:*

- $p \mapsto q$, and
- *q unless* $\neg p$.

Proof. Define g to be a constant function. Thus, g is non-increasing and bounded from below. Consequently, both the conditions of Theorem 5 are satisfied. □

Corollary 2 [7] *Let statement s be A if p, statement t be A if q, and F be any program. Then, program* $F \square t$ *is a property and fixed-point preserving refinement of* $F \square s$ *if the following three conditions hold:*

- *invariant* $q \Rightarrow p$ in $F \square s$,
- $p \mapsto q$ in $F[s]$, and
- *q unless* $\neg p$ in $F[s]$.

Proof. Let statement u be A *if* $p \wedge q$. Then, it follows from Corollary 1 that $F \square u$ is a property and fixed-point preserving refinement of $F \square s$. Because $q \Rightarrow p$ is an invariant of $F \square s$, $q \Rightarrow p$ holds initially in $F \square s$ and $q \Rightarrow p$ *unless false* holds in $F \square s$. Since the initial conditions remain unchanged and $F \square u$ preserves all the safety properties of $F \square s$, it follows that $q \Rightarrow p$ is also an invariant of $F \square u$. Therefore, $p \wedge q \equiv q$ in $F \square u$. Consequently, the guard of statement u can be changed from $p \wedge q$ to q by the substitution axiom, thus yielding program $F \square t$. The theorem follows. □

Theorem 5 was first proved in [26] and used in [28] to synchronize processes that accessed a common resource. Processes accessed the resource when a certain predicate was set to *true* by an underlying mutual exclusion algorithm. This predicate was added to the guard of each statement that accessed the resource. The correctness of the refinement followed from the starvation-freedom property of the mutual exclusion algorithm.

Example 9 (Continued from previous subsection). In the last section we discussed the implementation of variable x shared between processes F and G by a channel c from F to G and a local variable y at G. We transformed the pair of statements s (in F) and t (in G) to three statements s' (in F), t' (in G), and u (in G). The transformation was proved to be correct provided Condition $B0$ (or the stronger condition $B0'$) held. The transformed statements s' and u were in the right form whereas statement t' needed further refinements. Here, we use Theorem 5 to transform statement t' into statement v defined as follows:

$v :: vs, y := f(vs, y), g(y)$ *if* $b(y) \wedge q$

This statement is in the right form as it mentions variables local to process G.

In order to carry out the transformation from t' to v, we assume the following two conditions:

$b(x) \Rightarrow b(x \oplus d)$, for all x, d, and ... $(B1)$
$b(x) \Rightarrow f(vs, x \oplus d) = f(vs, x)$, for all x, d, vs. ... $(B2)$

The proof of correctness is in two steps: first, statement t' is transformed to statement $t'' :: vs, y := f(vs, y \oplus c), g(y)$ *if* $b(y) \wedge q$ and later, statement t'' is transformed to statement v. These two steps are detailed next.

First, consider the refinement of statement t' to t''. It can be shown by induction on the length of c and Condition $B1$ that $b(y) \Rightarrow b(y \oplus c)$. Therefore, by Corollary 2, the guard of statement t' can be strengthened to $b(y) \wedge q$ provided the following two conditions hold in the remainder of the program (i.e., the program without statement t'):

1. $y \oplus c = m \mapsto y = m$, and
2. $b(y)$ *unless false*.

For a proof of Condition 1, observe that on account of statement u, $y \oplus head(c) = k$ *ensures* $y = k$. The required condition follows from the repeated application of this progress property. For a proof of Condition 2, observe that statement u is the only statement in the remainder of the program that modifies variable y. Furthermore, from Condition $B1$, $b(y) \Rightarrow b(y \oplus head(c))$. Consequently, $b(y)$ *unless false* holds over the remainder of the program. This completes the proof of correctness of the transformation of statement t' to statement t''.

Next, we consider the refinement of statement t'' to statement v. It follows from repeated application of conditions $B1$ and $B2$ that

$b(y) \Rightarrow f(vs, y \oplus c) = f(vs, y)$.

Consequently, by the substitution axiom, the expression $f(vs, y \oplus c)$ may be substituted by the expression $f(vs, y)$ in statement t''. As a result, we obtain statement v, as desired.

Summing up the sequence of refinements, the statements s and t can be replaced by the statements s', u, and v provided Conditions $B0$, $B1$, and $B2$ hold. Similar conditions were defined by Chandy and Misra in [7] as the *Asynchrony Condition.* Our stepwise derivation of the condition provides a useful illustration of the program refinement theorems discussed here, in addition to providing some insight into their theorem. Recently, Sanders has also presented a development of the Asynchrony Condition in the framework of *mixed specifications* [22].

If, in addition, we want the above program refinement to be fixed-point preserving, then it can be shown, on the basis of Observation 1, that the following two conditions suffice:

$p \Rightarrow x \oplus d \neq x$, for all x, d, and ... $(B3)$

$b(x \oplus c) \wedge q \Rightarrow (g(x) \oplus c = x \oplus c \Rightarrow g(x) = x)$, for all x, c. ... $(B4)$ □

The following example taken from [7] illustrates an application of the refinements developed here.

Example 10. Let statements s and t be defined as follows:

$s :: x := x - d$ *if* p, and

$t :: vs, x := vs + 1, x + e$ *if* $x < 0 \wedge q$, where $d \geq 0$ and $e \geq 0$.

Conditions $B0'$ (which implies $B0$), $B1$, and $B2$ for this example translate to the following three properties respectively.

$(x - d) + e = (x + e) - d$,

$x < 0 \Rightarrow x - d < 0$, and

$x < 0 \Rightarrow vs + 1 = vs + 1$.

Because all these conditions hold, statements s, t can be transformed to the following set of statements that use a channel c and a local variable y while preserving all *unless and leads-to* properties.

$c := c; d$ *if* p

$[]\, y, c := y - head(c), tail(c)$ *if* $c \neq null$

$[]\, vs, y := vs + 1, y + e$ *if* $y < 0 \wedge q$

It follows from conditions $B3$ and $B4$ defined earlier that this refinement also preserves the fixed-point provided $p \Rightarrow d > 0$ is an invariant of the program. □

4.3 Refining atomicity

Concurrent algorithms that use a fine grain of atomicity are more efficient as more processes may execute concurrently. On the other hand, it is easier to prove algorithms with a coarse grain of atomicity as a less number of interleavings may need to be considered. Consequently, one method for program development is to derive a program with a coarse grain of atomicity and later refine it into another program that uses a finer grain of atomicity. Techniques that allow the atomicity of an algorithm to be reduced without compromising its correctness have been investigated by a number of authors. Lipton examined the semantics of operations and proposed the idea of *left-* and *right-commutativity* [15]. Later he used these definitions to define *left* and *right movers* and developed conditions under which a single coarse grain atomic operation could be replaced by a sequence of fine grain operations consisting of some right movers followed by a single operation followed by some left movers. Such a replacement would preserve the partial correctness and the deadlock freedom properties.

Lamport and Schneider generalized Lipton's theorem and presented a set of conditions under which a larger class of safety properties (not just partial correctness and deadlock freedom) would be preserved in the transformed program [14]. In a related work, Back considers the refinement of atomicity in *action systems* [3]. His approach is also based on the idea of left and right movers. However, the transformed fine-grained action system now preserves the total correctness (in our terminology, a fixed-point preserving refinement) of parallel programs. Similar results concerning refinement of atomicity appear in a joint work by Back and Sere [5]. Theorems similar to those discussed above can be developed in the context of Unity (or other state transition systems) by developing an appropriate notion of left- and right-commutativity and by defining left and right movers. However, instead of duplicating the existing work, we focus here on atomicity refinements that preserve all program properties, i.e., we concentrate on atomicity refinements that are property-preserving.

The need to preserve general program properties severely constrains the kinds of refinements that may be applied. In particular, it is no longer possible to replace a coarse-grained operation by a sequence of left movers and right movers. It appears that refinements now can no longer decompose the state transformation achieved by a statement since that would amount to violating some safety properties. For example, consider an assignment statement $s::A$ *if* b consisting of an assignment A and a guard b. It does not appear that the assignment A can be decomposed without violating some *unless* properties. However, it may be possible to decompose the guard b that defines when the assignment can be applied. Such modifications will be useful when the guard is quite complex, perhaps involving quantifications. (The usage of the term atomicity refinement for such guard simplifications may appear to be misleading but such refinements do reduce the number of shared variables that need to be accessed by a statement.)

In the remainder of this section, we concentrate on refinements that decompose the guards of statements while preserving the assignment components. We consider two kinds of complex guards – one in which the guard is a disjunction of predicates and the other in which the guard is a conjunction of predicates.

4.3.1 *Transforming existential quantification in guards*

Theorem 6. *Let F be any program and $s::A$ if $(\exists i::p.i)$ and $t::\langle \square i::t.i\rangle$, where $t.i::A$ if $p.i$, be any group of statements. Then, the program $F\square t$ is a property and fixed-point preserving refinement of the program $F\square s$ if $p.i$ unless $\neg(\exists i::p.i)$ holds in $F[s]$ for each i.* $\square$

Proof. First, we show that the strongest invariants of $F\square s$ and $F\square t$ are identical.

$$\begin{aligned} & X \Rightarrow wp(s, X)\\ =&\{\text{definition of } s\}\\ & X \Rightarrow wp(A \text{ if } (\exists i::p.i), X)\\ =&\{\text{definition of } wp\}\end{aligned}$$

$$\begin{aligned}
&X \Rightarrow (((\exists i::p.i) \Rightarrow wp(A, X)) \wedge (\neg(\exists i::p.i) \Rightarrow X))\\
=&\{\text{predicate calculus}\}\\
&(X \Rightarrow ((\exists i::p.i) \Rightarrow wp(A, X))) \wedge (X \Rightarrow (\neg(\exists i::p.i) \Rightarrow X))\\
=&\{\text{predicate calculus}\}\\
&(\forall i::X \Rightarrow (p.i \Rightarrow wp(A, X)))\\
=&\{(X \Rightarrow (\neg p.i \Rightarrow X)) \equiv true\}\\
&(\forall i::(X \Rightarrow (p.i \Rightarrow wp(A, X))) \wedge (X \Rightarrow (\neg p.i \Rightarrow X))\\
=&\{\text{predicate calculus}\}\\
&(\forall i::X \Rightarrow ((p.i \Rightarrow wp(A, X))) \wedge (\neg p.i \Rightarrow X))\\
=&\{\text{definition of } wp\}\\
&(\forall i::X \Rightarrow wp(A \textit{ if } p.i, X))\\
=&\{\text{definition of } t.i\}\\
&(\forall i::X \Rightarrow wp(t.i, X))
\end{aligned}$$

Based on the above derivation,

$$\begin{aligned}
&(\forall u: u \in F \square s : X \Rightarrow wp(u, X))\\
=&\{\text{predicate calculus}\}\\
&(\forall u: u \in F : X \Rightarrow wp(u, X)) \wedge (X \Rightarrow wp(s, X))\\
=&\{\text{previous derivation}\}\\
&(\forall u: u \in F : X \Rightarrow wp(u, X)) \wedge (\forall i::X \Rightarrow wp(t.i, X))\\
=&\{\text{predicate calculus and definition of } t\}\\
&(\forall u: u \in F \square t : X \Rightarrow wp(u, X))
\end{aligned}$$

It follows therefore that $SI\text{-}(F \square s) \equiv SI\text{-}(F \square t)$.

The remainder of the proof is in four parts: first, we show that $F \square t$ preserves all the safety properties of $F \square s$; second, we show that $F \square t$ preserves all the progress properties of $F \square s$; next, we show that the conjunction of the fixed-point and the strongest invariant of $F \square t$ implies the conjunction of the fixed-point and the strongest invariant of $F \square s$; finally, we show that if $F \square t$ terminates then $F \square s$ also terminates.

Part 1. Let b *unless* c be a property of $F \square s$. Then, $SI\text{-}(F \square s) \wedge b \wedge \neg c \Rightarrow wp(u, b \vee c)$ for all statements u in F and $SI\text{-}(F \square s) \wedge b \wedge \neg c \Rightarrow wp(s, b \vee c)$. Since $SI\text{-}(F \square s) \equiv SI\text{-}(F \square t)$, the desired proof of b *unless* c in $F \square t$ will follow if we show that $SI\text{-}(F \square s) \wedge b \wedge \neg c \Rightarrow wp(t.i, b \vee c)$ holds for each i. This proof is as follows.

$$\begin{aligned}
&SI\text{-}(F \square s) \wedge b \wedge \neg c \Rightarrow wp(s, b \vee c) && \text{, } b \text{ unless } c \text{ in } F \square s\\
&SI\text{-}(F \square s) \wedge b \wedge \neg c \Rightarrow wp(A \textit{ if } (\exists i::p.i), b \vee c) && \text{, definition of } s\\
&SI\text{-}(F \square s) \wedge b \wedge \neg c \wedge p.i \Rightarrow wp(A, b \vee c) && \text{, definition of } wp\\
&SI\text{-}(F \square s) \wedge b \wedge \neg c \wedge \neg p.i \Rightarrow (b \vee c) && \text{, predicate calculus}\\
&SI\text{-}(F \square s) \wedge b \wedge \neg c \Rightarrow wp(A \textit{ if } p.i, b \vee c) && \text{, above two}\\
&SI\text{-}(F \square s) \wedge b \wedge \neg c \Rightarrow wp(t.i, b \vee c) && \text{, definition of } t.i
\end{aligned}$$

(End of Part 1)

Parts 2, 3, and 4. Shown in the appendix. (End of Proof)

A simplified version of Theorem 6 was used in [28] to map a program onto a set of processes. There, we had a program fragment $\langle \square j::A.j \textit{ if } (\exists n::p.j.n)\rangle$ in which statement $A.j$ represented a task, n ranged over a given set of processes, and predicate $p.j.n$ was true if and only if task $A.j$ was mapped to process n. By applying the above theorem, we obtained a refined fragment $\langle \square n::\langle \square j::A.j \textit{ if } p.j.n\rangle\rangle$ in which the term of the outer quantification represented the code of process n.

4.3.2 Transforming universal quantification in guards

Theorem 7. *Let F be any program and $s::p:=true$ if $\neg p \wedge (\forall i::q.i)$ be any statement. Consider a refinement of statement s in which the predicates $q.i$ are computed asynchronously with the help of fresh variables $y.i$ as follows: variables $y.i$ are initialized to false and statement s is replaced by program $G=(s' \| t')\Box\langle\Box i::u.i\rangle$, where statements s', t', and $u.i$ are defined as follows:*

$s'::p:=true \text{ if } \neg p \wedge (\forall i::y.i)$,
$t'::\langle \| i::y.i:=false \text{ if } \neg p \wedge (\forall i::y.i)\rangle$, and
$u.i::y.i:=true \text{ if } \neg p \wedge q.i$.

Then program $F\Box G$ is a property and a fixed-point preserving refinement of program $F\Box s$ if program F satisfies $\neg p \wedge q.i$ unless false, for each i, in the context of program $H=(s\|t)\Box\langle\Box j::u.j\rangle$, where statement t is defined as $\langle\| i::y.i:=false$ if $\neg p \wedge (\forall i::q.i)\rangle$. □

Proof. The refinement from $F\Box s$ to $F\Box G$ consists of two subrefinements: first from $F\Box s$ to $F\Box H$, and the second from $F\Box H$ to $F\Box G$. Because of transitivity of refinements, it is sufficient to prove that each of the two subrefinements is a property and fixed-point preserving refinement. These proofs are detailed next.

First, consider the subrefinement of $F\Box s$ to $F\Box H$. In this refinement statement t is synchronously superposed to statement s and statements $u.i$ are asynchronously superposed to the program. Thus, by the superposition theorem, program $F\Box H$ preserves all *unless and leads-to* properties of $F\Box s$. Therefore, this is a property preserving refinement. Also from the superposition theorem, the fixed-point of $F\Box H$ implies the fixed-point of $F\Box s$. Moreover, since superposition preserves all invariants, the strongest invariant of $F\Box H$ implies the strongest invariant of $F\Box s$. This proves the second condition for fixed-point preserving refinements. Our remaining proof obligation is to show that if $F\Box s$ terminates, then so does $F\Box H$. Let $R.i$ denote the fixed-point of statement $u.i$. Then,

$FP\text{-}(F\Box s) \equiv FP\text{-}F \wedge (p \vee (\exists i::\neg q.i))$,
$FP\text{-}(F\Box(s\|t)) \equiv FP\text{-}(F\Box s)$,
$R.i \equiv p \vee \neg q.i \vee y.i$, and
$FP\text{-}(F\Box H) \equiv FP\text{-}(F\Box s) \wedge (\forall i::R.i)$.

We prove the following two properties of program $F\Box H$ for each i,

$FP\text{-}(F\Box s) \wedge R.i$ *unless false*, and ... $(E0)$
$FP\text{-}(F\Box s) \mapsto FP\text{-}(F\Box s) \wedge R.i.$ ... $(E1)$

Based on these properties, observe the following.

$true \mapsto FP\text{-}(F\Box s)$ in $F\Box s$	, assumption
$true \mapsto FP\text{-}(F\Box s)$ in $F\Box H$	, preservation of *leads-to*
$FP\text{-}(F\Box s) \mapsto FP\text{-}(F\Box s) \wedge (\forall i::R.i)$ in $F\Box H$	, completion theorem on $E0, E1$
$true \mapsto FP\text{-}(F\Box s) \wedge (\forall i::R.i)$ in $F\Box H$	, transitivity on above two
$true \mapsto FP\text{-}(F\Box H)$	, definition □

Proof of $E0$.

$FP\text{-}(F\Box(s\|t)) \wedge R.i$ *unless false* in $F\Box(s\|t)\Box u.i[\langle\Box j:j\neq i:u.j\rangle]$
, stability at fixed-point

$FP\text{-}(F\Box s) \wedge R.i$ *unless false* in $F\Box(s\|t)\Box u.i[\langle\Box j:j\neq i:u.j\rangle]$
, $FP\text{-}(F\Box(s\|t) \equiv FP\text{-}(F\Box s)$

$FP\text{-}(F\Box s) \wedge R.i$ *unless false in* $\langle\Box j:j\neq i:u.i\rangle[F\Box(s\|t)\Box u.i]$
, observing $u.j$

$FP\text{-}(F\Box s) \wedge R.i$ *unless false* in $F\Box H$, union theorem

Proof of $E1$.

$FP\text{-}(F\square(s\|t))$ *unless false* in $F\square(s\|t)[\langle\square j::u.j\rangle]$, stability at fixed point
$FP\text{-}(F\square s)$ *unless false* in $F\square(s\|t)[\langle\square j::u.j\rangle]$, $FP\text{-}(F\square(s\|t))\equiv FP\text{-}(F\square s)$
$FP\text{-}(F\square s)$ *unless false* in $\langle\square j::u.j\rangle\,[F\square(s\|t)]$, observing $u.j$
$FP\text{-}(F\square s)$ *unless false* in $F\square(s\|t)\square\langle\square j::u.j\rangle$, union theorem
$FP\text{-}(F\square s)$ *ensures* $FP\text{-}(F\square s)\wedge R.i$ in $u.i\,[F\square(s\|t)\square\langle\square j: j\neq i: u.j\rangle]$
, observing $u.i$
$FP\text{-}(F\square s)$ *ensures* $FP\text{-}(F\square s)\wedge R.i$ in $F\square H$, union theorem
$FP\text{-}(F\square s)\mapsto FP\text{-}(F\square s)\wedge R.i$ in $F\square H$, definition of *leads-to*

This completes the proof of correctness of the first subrefinement.

Next, consider the second subrefinement from $F\square H$ to $F\square G$ in which statements s and t are replaced by statements s' and t' respectively. In both of these transformations the guard $\neg p\wedge(\forall i::q.i)$ has been replaced by $\neg p\wedge(\forall i::y.i)$. Therefore, our proof obligation will follow from Corollary 2 in Sect. 4 if we prove the following properties.

1. *invariant* $\neg p\wedge(\forall i::y.i)\Rightarrow\neg p\wedge(\forall i::q.i)$ in $F\square H$,
2. $\neg p\wedge(\forall i::y.i)$ *unless* $p\vee(\exists i::\neg q.i)$ in $F\square\langle\square i::u.i\rangle$ in the context $s\|t$, and
3. $\neg p\wedge(\forall i::q.i)\mapsto\neg p\wedge(\forall i::y.i)$ in $F\square\langle\square i::u.i\rangle$ in the context $s\|t$.

These three properties are proved in the appendix. (End of proof)

Theorem 7 originates from the two-phase handshake used in network protocols and can be understood as follows. The statement that sets program variable p to *true* represents a *controller* process and the statements that set predicates $q.i$ represent *subordinate* processes. After setting $q.i$ to *true*, a subordinate process waits for an acknowledgement from the controller (in the form of variable p being set to *true*) before resetting it to *false*. The controller polls each subordinate (through local variable $y.i$) and sets p to *true* when it finds every $y.i$ to be *true*. The correctness of the refinement is based on the fact that the subordinate processes do not reset predicates $q.i$ while they are being polled by the controller. It is possible to introduce more asynchrony into the above refinement by setting variables $y.i$ to *false* asynchronously. However, it has to be ensured that the next phase of polling $q.i$ does not begin until every $y.i$ from the previous phase have been reset.

The refinement of a synchronous computation of guards by an asynchronous computation is similar to the idea of delay insensitivity discussed in the context of electronic circuits by Seitz in [24] and Martin in [19], and in the context of programs by Chandy and Misra in [7]. A program is said to be delay insensitive if for all the assignment statements, the right hand side of the assignment does not change as long as the left hand side of the assignment does not equal the right hand side. Thus, delay insensitivity allows an electronic circuit to be designed without using a common clock. For example, consider the statement $s::p:=p\vee(q\wedge r)$ that represents a combinatorial logic-gate. (Note that this statement is equivalent to the statement $p:=true$ *if* $\neg p\vee(q\wedge r)$.) For this statement to be delay insensitive, the program that s is a part of should satisfy the following property:

$\neg p\wedge q\wedge r$ *unless* p. … (P)

In other words, a state in which p is *false* and q and r are *true* persists until p is set to *true*. On the other hand, the conditions for refinement of atomicity derived earlier ensure that the expression on the right hand side of a statement can be computed piecemeal. Delay insensitivity in general does not ensure this piecemeal evaluation; it ensures asynchrony among different statements as opposed to atomicity refinement which ensures asynchrony in the execution of a statement. For example, consider the statement s defined earlier and let us try using property P as the condition for atomicity refinement. In that case, it is possible that predicates q and r keep oscillating between *true* and *false* such that they are never *true* at the same time. Consequently, condition P is vacuously satisfied. However, if q and r are evaluated asynchronously then it is possible that both will be found to be *true* and the subsequent setting of p to *true* will be incorrect. This shows that condition P is not acceptable as the correctness requirement. As is evident from Theorem 7, the correct conditions for the refinement of atomicity in this case are:

$$\neg p \wedge q \textit{ unless false} \text{ and } \neg p \wedge r \textit{ unless false.}$$

5 Concluding remarks

In spite of numerous logics and methodologies, concurrent programming has remained a very difficult and complex task. There are two possible explanations for this complexity. First, most of the concurrent programs are unstructured and written in an ad-hoc way. This makes the separation of concerns next to impossible and consequently, the proofs are inextricably entangled. Second, most of the concurrent programs contain a lot of irrelevant information that has nothing to do with the underlying algorithm's correctness. It is far easier to prove the algorithms that the programs are based on than the programs themselves. Program derivation through stepwise refinements is one possible solution to the aforementioned problems. Such a formal derivation achieves the separation of concerns by postponing efficiency and architectural decisions until late in the design process. In this paper we discussed the program refinement phase of stepwise refinements by considering property preserving and fixed-point preserving refinements. We developed a small theory of these refinements and discussed conditions under which some common program transformations are correct. We hope that these theorems will be a first step towards building an adequate set of tools for program transformations. Once a sufficient number of theorems have been developed, it may be possible to implement recurring themes in parallel program derivation (viz. abstract data type implementation, process synchronization and scheduling) automatically while preserving the correctness of the original program.

The absence of *local variables* in Unity implies that every variable is treated as a *shared* variable. Consequently, program refinements need to consider properties across all variables and not just those variables at the interface. We are currently examining the introduction of local variables in Unity and the development of a theory of program refinements that distinguishes between local and interface variables.

Acknowledgements. The ideas and theorems discussed here were motivated by discussions with Jay Misra. The paper improved considerably from the comments of the anonymous referees and Ying Liu, and discussions with the Unity group at The University of Texas at Austin.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. In: Proceedings of the 3rd IEEE Symposium on Logic in Computer Science, pp. 165–175, 1988
2. Back, R.J.R.: Correctness preserving program refinements: proof theory and applications. Technical Report Mathematical Center Tracts, 131, Center for Mathematics and Computer Science (CWI), Amsterdam 1980
3. Back, R.J.R.: A method for refining atomicity in parallel algorithms. In: Odijk, E., Rein, M., Syre, J.C. (eds.) PARLE '89 – Parallel architectures and languages Europe, Vol. II (Lect. Notes Comput. Sci. 366, pp. 199–216) Berlin, Heidelberg, New York: Springer 1989
4. Back, R.J.R.: Refinement calculus, part II: parallel and reactive programs. In: Bakker, J.W. de, Roever, W.P. de, Rozenberg, G. (eds.) Stepwise refinement of distributed systems: models, formalisms, correctness (Lect. Notes Comput. Sci., Vol. 430, pp. 67–93) Berlin, Heidelberg, New York: Springer 1990
5. Back, R.J.R., Sere, K.: Stepwise refinement of parallel algorithms. Sci. Comput. Program. **13**, 133–180 (1989–90)
6. Back, R.J.R., von Wright, J.: Refinement calculus, part I: sequential nondeterministic processes. In: Bakker, J.W. de, Roever, W.P. de, Rozenberg, G. (eds.) Stepwise refinement of distributed systems: models, formalisms, correctness (Lect. Notes Comput. Sci., Vol. 430, pp. 42–66) Berlin, Heidelberg, New York: Springer 1990
7. Chandy, K.M., Misra, J.: Parallel program design: a foundation. Reading, MA: Addison Wesley 1988
8. Dijkstra, E.W.: A discipline of programming. Englewood Cliffs, NJ: Prentice Hall 1976
9. Gries, D.: The science of programming. Berlin, Heidelberg, New York: Springer 1981
10. Hoare, C.A.R.: Proofs of correctness of data representations. Acta Inf. **4**, 271–281 (1972)
11. Jones, C.B.: Systematic software development using VDM. Englewood Cliffs, NJ: Prentice Hall 1986
12. Lam, S.S., Shankar, A.U.: Protocol verification via projections. IEEE Trans. Software Eng. **10(4)**, 325–342 (1984)
13. Lamport, L.: A simple approach to specifying concurrent systems. Commun. ACM **32(1)**, 32–47 (1989)
14. Lamport, L., Schneider, F.: Pretending atomicity. Technical Report TR-89-1005, Dept. of Computer Science, Cornell University, May 1989
15. Lipton, R.J.: Reduction: a method for proving properties of parallel programs. Commun. ACM **14(12)**, 717–721 (1975)
16. Liu, Y., Singh, A.K., Bagrodia, R.: A decomposition based approach to design of efficient parallel programs. In: Etiemble, D., Syre, J.-C. (eds.) PARLE '92. Parallel architectures and languages Europe. (Lect. Notes Comput. Sci., Vol. 605, pp. 21–36) Berlin, Heidelberg, New York: Springer 1992
17. Lynch, N.A., Tuttle, M.: Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, pp. 137–151, 1987
18. Manna, Z., Pnueli, A.: How to cook a temporal proof system for your pet language. In: Proceedings of the 9th ACM Symposium on Principles of Programming Languages, pp. 141–154, 1983
19. Martin, A.J.: Compiling communicating processes into delay-insensitive VLSI circuits. Distrib. Comput **1**, 226–234 (1986)
20. Morgan, C., Gardiner, P.H.B.: Data refinement by calculation. Acta Inf. **27**, 481–503 (1990)
21. Morris, J.M.: Laws of data refinement. Acta Inf. **26**, 287–308 (1989)

22. Sanders, B.: Stepwise refinement of mixed specifications of concurrent programs. In: Proceedings of IFIP Conference on Programming Concepts and Methods, Israel. Amsterdam: Elsevier 1990
23. Sanders, B.: Eliminating the substitution axiom from Unity. Formal Aspects Comput. **3**, 189–205 (1991)
24. Seitz, C.: System timing. In: Mead, C., Conway, L. (eds.) Introduction to VLSI Systems. Reading, MA: Addison Wesley 1980
25. Singh, A.K.: Leads-to and program union. Notes on Unity: 06-89, The University of Texas at Austin, Texas, May 1989
26. Singh, A.K.: On strengthening the guard. Notes on Unity: 07-89. The University of Texas at Austin, Texas, June 1989
27. Singh, A.K.: Program refinement in fair transition systems. In: Aarts, E.H., Leeuwen, J. van, Rem, M. (eds.) PARLE '91. Parallel architectures and languages Europe (Lect. Notes Comput. Sci., Vol. 506, pp. 128–147) Berlin, Heidelberg, New York: Springer 1991
28. Singh, A.K., Overbeek, R.: Derivation of efficient parallel programs: an example from genetic sequence analysis. Int. J. Parallel Program. **18(6)**, 446–484 (1989)

Appendix

Proof of Theorem 4

Step 3. Based on assertion $A2$, we replace $p(h(y))$ by $q(y)$ in the guard of statement u'. The resulting program, $F' \square (s \| t)$, is a property and fixed-point preserving refinement of program $F' \square (s \| u')$ on account of the substitution axiom.

Step 4. In this step, we delete statement s and obtain the desired program $F' \square t$. We prove that this program is a property and fixed-point preserving refinement of $F' \square (s \| t)$ under the coupling $x = h(y)$.

First, we show that $x = h(y)$ is an invariant of program $F' \square (s \| t)$. We showed earlier that $x = h(y)$ is an invariant of program $F \square (s \| u)$. Because program $F' \square (s \| t)$ was obtained from this program by appealing to the substitution axiom, $x = h(y)$ is also an invariant of program $F' \square (s \| t)$, as required.

Next, we show that program $F' \square t$ is a property preserving refinement of program $F' \square (s \| t)$ under the coupling $x = h(y)$. Consider program $F' \square (s \| t)$ and a property p *unless* q that holds in it. From the substitution axiom, the property p' *unless* q' also holds in it. Because program $F' \square t$ and predicates p', q' do not mention x, it can be shown (by a result similar to the superposition theorem) that the property p' *unless* q' also holds in the program $F' \square t$. A similar result can be proved for any *leads-to* property $p \mapsto q$ that holds in $F' \square (s \| t)$. This proves that $F' \square t$ is a property preserving refinement of $F' \square (s \| t)$ under the coupling $x = h(y)$.

Finally, we show that program $F' \square t$ is a fixed-point preserving refinement of program $F' \square (s \| t)$ under the coupling $x = h(y)$. Assume that $true \mapsto FP\text{-}(F' \square (s \| t))$ in program $F' \square (s \| t)$. Because program $F' \square t$ does not mention x, it can be shown that $true \mapsto FP\text{-}(F' \square t)$ in program $F' \square t$. This proves the first condition for fixed-point preserving refinements. For a proof of the second condition, note the following.

- $FP\text{-}(F' \square (s \| t)) \equiv FP\text{-}(F' \square t) \wedge x = h(y)$, and
- $SI\text{-}(F' \square (s \| t)) \equiv SI\text{-}(F' \square t) \wedge x = h(y)$.

Therefore, $(FP\text{-}(F'\Box t) \wedge SI\text{-}(F'\Box t) \wedge x=h(y)) \Rightarrow (FP\text{-}(F'\Box(s\|t) \wedge SI\text{-}(F'\Box(s\|t))$, as required. This proves that $F'\Box t$ is a fixed-point preserving refinement of $F'\Box(s\|t)$ under the coupling $x=h(y)$. $\square$

Proof of Theorem 5

Part 2. Let $b \mapsto c$ be a property of $F\Box s$. We prove that $b \mapsto c$ in $F\Box t$ by induction on the proof of $b \mapsto c$ in $F\Box s$.

Base case: b *ensures* c in $F\Box s$.

From the definition of *ensures*, b *unless* c in $F\Box s$ and there exists a statement u in $F\Box s$ such that $SI\text{-}(F\Box s) \wedge b \wedge \neg c \Rightarrow wp(u, c)$. If u belongs to F then the proof follows as the refinement preserves all the safety properties and statements in F remain unchanged. Otherwise, $u=s$, i.e.,

$$SI\text{-}(F\Box s) \wedge b \wedge \neg c \Rightarrow wp(s, c) \quad \ldots\ (C0)$$

Since statement s consists of an assignment guarded by predicate p, it follows that

$$SI\text{-}(F\Box s) \wedge b \wedge \neg p \Rightarrow c. \quad \ldots\ (C1)$$

We prove the following two properties of program $F\Box t$.

$$\begin{array}{ll} b \wedge p \mapsto (b \wedge q) \vee c, \text{ and} & \ldots\ (C2) \\ b \wedge q \wedge g=k \mapsto c \vee g<k, & \ldots\ (C3) \end{array}$$

Based on the above properties, observe the following in $F\Box t$:

$$\begin{array}{ll}
SI\text{-}(F\Box s) \wedge b \wedge \neg p \Rightarrow c & \text{, property } C1 \\
SI\text{-}(F\Box t) \wedge b \wedge \neg p \Rightarrow c & \text{, } SI\text{-}(F\Box t) \Rightarrow SI\text{-}(F\Box s) \\
SI\text{-}(F\Box t) \Rightarrow (b \wedge \neg p \Rightarrow c) & \text{, predicate calculus} \\
\textit{invariant } b \wedge \neg p \Rightarrow c & \text{, property of } SI \quad \ldots\ (C4) \\
b \wedge \neg p \mapsto c & \text{, implication rule} \\
b \mapsto (b \wedge q) \vee c & \text{, disjunction with } C2 \\
g=k \textit{ unless } g<k & \text{, } g \text{ is non-increasing} \\
b \wedge g=k \mapsto (b \wedge q \wedge g=k) \vee c \vee g<k & \text{, PSP theorem} \\
b \wedge g=k \mapsto c \vee g<k & \text{, transitivity with } C3 \\
b \textit{ unless } c & \text{, preservation of safety properties} \\
b \wedge g=k \mapsto c \vee (b \wedge g<k) & \text{, PSP theorem} \\
b \mapsto c & \text{, induction over } k \quad \square
\end{array}$$

Proof of C2.

$$\begin{array}{ll}
p \mapsto q \text{ in } F[t] & \text{, assumption} \\
\neg(p \wedge q) \Rightarrow FP\text{-}t & \text{, } t \text{ is } A \textit{ if } p \wedge q \\
p \mapsto q \vee (p \wedge q) \text{ in } F\Box t & \text{, composition of } \textit{leads-to} \\
p \mapsto q \text{ in } F\Box t & \text{, predicate calculus} \quad \ldots\ (C5) \\
b \textit{ unless } c \text{ in } F\Box t & \text{, preservation of safety properties} \\
b \wedge p \mapsto (b \wedge q) \vee c \text{ in } F\Box t & \text{, PSP theorem}
\end{array}$$

Proof of C3.

$$\begin{array}{ll}
SI\text{-}(F\Box s) \wedge b \wedge \neg c \Rightarrow wp(s, c) & \text{, property } (C0) \\
SI\text{-}(F\Box s) \wedge b \wedge q \wedge \neg c \Rightarrow wp(t, c) & \text{, definitions of } s, t
\end{array}$$

$SI\text{-}(F\square t) \wedge b \wedge q \wedge \neg c \Rightarrow wp(t, c)$, $SI\text{-}(F\square t) \Rightarrow SI\text{-}(F\square s)$
$SI\text{-}(F\square t) \wedge b \wedge q \wedge g = k \wedge \neg c \Rightarrow wp(t, c \wedge g \leq k)$, g is non-increasing
$SI\text{-}(F\square t) \wedge b \wedge q \wedge g = k \wedge \neg c \Rightarrow wp(t, (b \wedge \neg p) \vee c \vee g < k)$
, property of *wp*
$b \wedge q \wedge g = k$ *ensures* $(b \wedge \neg p) \vee c \vee g < k$ in $t[F]$, definition of *ensures* ... ($C6$)
b *unless* c in $F\square s$, assumption
b *unless* c in $F[s]$, union theorem
$q \wedge g = k$ *unless* $\neg p \vee g < k$ in $F[s]$, assumption
$b \wedge q \wedge g = k$ *unless* $(b \wedge \neg p) \vee c \vee g < k$ in $F[s]$, *conjunction*
$b \wedge q \wedge g = k$ *unless* $(b \wedge \neg p) \vee c \vee g < k$ in $F[t]$, $SI\text{-}(F\square t) \Rightarrow SI\text{-}(F\square s)$
$b \wedge q \wedge g = k$ *ensures* $(b \wedge \neg p) \vee c \vee g < k$ in $F\square t$, union theorem with $C6$
$b \wedge q \wedge g = k \mapsto c \vee g < k$ in $F\square t$, property $C4$

This concludes the base case.

Induction step:

- $b \mapsto r$ in $F\square s$, r *ensures* c in $F\square s$.
  $b \mapsto r$ in $F\square t$, induction hypothesis
  $r \mapsto c$ in $F\square t$, proof similar to base case
  $b \mapsto c$ in $F\square t$, transitivity

- $b \equiv (\exists i::b.i)$ and $b.i \mapsto c$ in $F\square s$, for all i
  $b.i \mapsto c$ in $F\square t$, induction hypothesis
  $(\exists i::b.i) \mapsto c$ in $F\square t$, disjunction over i
  $b \mapsto c$ in $F\square t$, $b \equiv (\exists i::b.i)$ $\square$

Proof of Theorem 6

Part 2. Let $b \mapsto c$ be a property of $F\square s$. We prove that $b \mapsto c$ in $F\square t$ by induction on the proof of $b \mapsto c$ in $F\square s$.

Base case: b *ensures* c in $F\square s$.

From the definition of *ensures*, b *unless* c in $F\square s$ and there exists a statement u in $F\square s$ such that $SI\text{-}(F\square s) \wedge b \wedge \neg c \Rightarrow wp(u, c)$. If u belongs to F then the proof follows as the refinement preserves all the safety properties and statements in F remain unchanged. Otherwise, $u = s$, i.e., $SI\text{-}(F\square s) \wedge b \wedge \neg c \Rightarrow wp(s, c)$. Therefore, noting that $SI\text{-}(F\square s) \equiv SI\text{-}(F\square t)$,

$SI\text{-}(F\square t) \wedge b \wedge \neg(\exists i::p.i) \Rightarrow c$, and ... ($D0$)
$SI\text{-}(F\square t) \wedge b \wedge \neg c \wedge p.i \Rightarrow wp(A, c)$, for each i. ... ($D1$)

We prove the following two properties:

$b \wedge p.i$ *unless* $c \vee (b \wedge \neg(\exists i::p.i))$ in $F\square\langle\square j : j \neq i : t.j\rangle$ in the context $t.i$ and ... ($D2$)
$b \wedge p.i$ *ensures* $c \vee (b \wedge \neg(\exists i::p.i))$ in $t.i$ in the context $F\square\langle\square j : j \neq i : t.j\rangle$. ... ($D3$)

Based on the above properties, observe the following in $F\square t$:

$b \wedge p.i$ *ensures* $c \vee (b \wedge \neg(\exists i::p.i))$, union theorem on $D2$, $D3$

$b \wedge p.i \mapsto c \vee (b \wedge \neg(\exists i :: p.i))$	, definition of $\mapsto$
$b \wedge (\exists i :: p.i) \mapsto c \vee (b \wedge \neg(\exists i :: p.i))$	, disjunction over i
$(b \wedge \neg(\exists i :: p.i)) \mapsto c$	, implication rule with $D0$
$(b \wedge (\exists i :: p.i)) \mapsto c$	, transitivity on above two
$b \mapsto c$	, disjunction on above two $\square$

*Proof of D*2.

$SI\text{-}(F \square t) \wedge b \wedge \neg c \wedge p.i \Rightarrow wp(A, c)$	, condition $D1$
$SI\text{-}(F \square t) \wedge b \wedge p.i \wedge \neg c \wedge p.j \Rightarrow wp(A, (b \wedge p.i) \vee c)$	, property of *wp* and predicate calculus
$SI\text{-}(F \square t) \wedge b \wedge p.i \wedge \neg c \wedge \neg p.j \Rightarrow (b \wedge p.i) \vee c$	, predicate calculus
$SI\text{-}(F \square t) \wedge b \wedge p.i \wedge \neg c \Rightarrow wp(A \textit{ if } p.j, (b \wedge p.i) \vee c)$	, above two
$SI\text{-}(F \square t) \wedge b \wedge p.i \wedge \neg c \Rightarrow wp(t.j, (b \wedge p.i) \vee c)$	, definition of $t.j$
$b \wedge p.i \textit{ unless } c$ in $\langle \square j : j \neq i : t.j \rangle [F \square t.i]$	, definition of *unless* ... $(D4)$
$b \textit{ unless } c$ in $F \square s$	, $b \textit{ ensures } c$ in $F \square s$
$b \textit{ unless } c$ in $F[s]$	, union theorem
$p.i \textit{ unless } \neg(\exists i :: p.i)$ in $F[s]$	, assumption
$b \wedge p.i \textit{ unless } c \vee (b \wedge \neg(\exists i :: p.i))$ in $F[s]$	, conjunction
$b \wedge p.i \textit{ unless } c \vee (b \wedge \neg(\exists i :: p.i))$ in $F[t]$	, $SI\text{-}(F \square s) \equiv SI\text{-}(F \square t)$
$b \wedge p.i \textit{ unless } c \vee (b \wedge \neg(\exists i :: p.i))$ in $F \square \langle j : j \neq i : t.j \rangle [t.i]$	, union theorem with $D4$

*Proof of D*3.

$SI\text{-}(F \square t) \wedge b \wedge \neg c \wedge p.i \Rightarrow wp(A, c)$	, condition $D1$
$SI\text{-}(F \square t) \wedge b \wedge p.i \wedge \neg c \wedge p.i \Rightarrow wp(A, c)$	, predicate calculus
$SI\text{-}(F \square t) \wedge b \wedge p.i \wedge \neg c \wedge \neg p.i \Rightarrow c$	, antecedent $\equiv$ *false*
$SI\text{-}(F \square t) \wedge b \wedge p.i \wedge \neg c \Rightarrow wp(A \textit{ if } p.i, c)$	, above two
$SI\text{-}(F \square t) \wedge b \wedge p.i \wedge \neg c \Rightarrow wp(t.i, c)$	, definition of $t.i$
$b \wedge p.i \textit{ ensures } c$ in $t.i[F \square \langle \square j : j \neq i : t.j \rangle]$	, definition of *ensures*
$b \wedge p.i \textit{ ensures } c \vee (b \wedge \neg(\exists i :: p.i))$ in $t.i[F \square \langle \square j : j \neq i : t.j \rangle]$	, weakening

This concludes the base case.

Induction step:

- $b \mapsto r$ in $F \square s$, $r \textit{ ensures } c$ in $F \square s$.

$b \mapsto r$ in $F \square t$	, induction hypothesis
$r \mapsto c$ in $F \square t$	, proof similar to base case
$b \mapsto c$ in $F \square t$	, transitivity

- $b \equiv (\exists i :: b.i)$ and $b.i \mapsto c$ in $F \square s$, for all i

$b.i \mapsto c$ in $F \square t$	, induction hypothesis
$(\exists i :: b.i) \mapsto c$ in $F \square t$	, disjunction over i
$b \mapsto c$ in $F \square t$	, $b \equiv (\exists i :: b.i)$. (End of Part 2)

Part 3. Let R be the fixed-point of assignment A. Observe the following

$$\begin{aligned}
&FP\text{-}(F \square t)\\
=&\{\text{definition of fixed-point}\}\\
&FP\text{-}F \wedge (\forall i::p.i \Rightarrow R)\\
=&\{\text{predicate calculus}\}\\
&FP\text{-}F \wedge ((\exists i::p.i) \Rightarrow R)\\
=&\{\text{definition of fixed-point}\}\\
&FP\text{-}(F \square s)
\end{aligned}$$

The required proof obligation then follows as $SI\text{-}(F \square t) \equiv SI\text{-}(F \square s)$. (End of Part 3)

Part 4.

$true \mapsto FP\text{-}(F \square s)$ in $F \square s$, assumption
$true \mapsto FP\text{-}(F \square s)$ in $F \square t$, preservation of *leads-to*
$true \mapsto FP\text{-}(F \square t)$ in $F \square t$, $FP\text{-}(F \square s) \equiv FP\text{-}(F \square t)$ as shown earlier $\square$

Proof of Theorem 7

Proof of 1. We prove the following properties of program $F \square H$.

$p \wedge \neg y.i$ *unless* $\neg p \wedge \neg y.i$ ... $(E2)$
$\neg p \wedge \neg y.i \wedge q.i$ *unless* $(\neg p \wedge y.i \wedge q.i) \vee (p \wedge \neg y.i)$ ... $(E3)$
$\neg p \wedge \neg y.i \wedge \neg q.i$ *unless* $\neg y.i \wedge (p \vee q.i)$ ... $(E4)$
$\neg p \wedge y.i \wedge q.i$ *unless* $p \wedge \neg y.i$ ... $(E5)$

Based on these properties, observe the following in program $F \square H$:

$(p \vee q.i) \wedge \neg y.i$ *unless* $(\neg p \wedge \neg y.i \wedge \neg q.i) \vee (\neg p \wedge y.i \wedge q.i)$
, disjunction on $E2, E3$
$\neg y.i$ *unless* $\neg p \wedge y.i \wedge q.i.$, disjunction with $E4$
$\neg y.i \vee (\neg p \wedge q.i)$ *unless false* , disjunction with $E5$
initially $\neg y.i$, observing $F \square H$
invariant $\neg y.i \vee (\neg p \wedge q.i)$, above two
invariant $\neg p \wedge y.i \Rightarrow q.i$, weakening the invariant
invariant $\neg p \wedge (\forall i::y.i) \Rightarrow \neg p \wedge (\forall i::q.i)$, conjunction over i

Proof of $E2$.

p *unless* $\neg p$ in $F[H]$, irreflexivity
$\neg y.i$ *unless false* in $F[H]$, $y.i$ is a fresh variable
$p \wedge \neg y.i$ *unless* $\neg p \wedge \neg y.i$ in $F[H]$, conjunction on above two
$p \wedge \neg y.i$ *unless false* in $H[F]$, $p \Rightarrow FP\text{-}H$
$p \wedge \neg y.i$ *unless* $\neg p \wedge \neg y.i$ in $F \square H$, union theorem

Proof of $E3$.

$\neg p \wedge q.i$ *unless false* in $F[H]$, assumption
$\neg y.i$ *unless false* in $F[H]$, $y.i$ is a fresh variable

$\neg p \wedge \neg y.i \wedge q.i$ *unless false* in $F[H]$, conjunction on above two
$\neg p \wedge \neg y.i \wedge q.i$ *unless* $(\neg p \wedge y.i \wedge q.i) \vee (p \wedge \neg y.i)$ in $H[F]$
, observing H
$\neg p \wedge \neg y.i \wedge q.i$ *unless* $(\neg p \wedge y.i \wedge q.i) \vee (p \wedge \neg y.i)$ in $F \square H$
, union theorem

Proof of E4.

$\neg p \wedge \neg q.i$ *unless* $p \vee q.i$ in $F[H]$, irreflexivity
$\neg y.i$ *unless false* in $F[H]$, $y.i$ is a fresh variable
$\neg p \wedge \neg y.i \wedge \neg q.i$ *unless* $\neg y.i \wedge (p \vee q.i)$ in $F[H]$
, conjunction on above two
$\neg p \wedge \neg y.i \wedge \neg q.i$ *unless false* in $H[F]$, $\neg q.i \Rightarrow FP\text{-}H$
$\neg p \wedge \neg y.i \wedge \neg q.i$ *unless* $\neg y.i \wedge (p \vee q.i)$ in $F \square H$
, union theorem

Proof of E5.

$\neg p \wedge q.i$ *unless false* in $F[H]$, assumption
$y.i$ *unless false* in $F[H]$, $y.i$ is a fresh variable
$\neg p \wedge y.i \wedge q.i$ *unless false* in $F[H]$, conjunction on above two
$\neg p \wedge y.i \wedge q.i$ *unless* $p \wedge \neg y.i$ in $H[F]$, observing H
$\neg p \wedge y.i \wedge q.i$ *unless* $p \wedge \neg y.i$ in $F \square H$, union theorem

Proof of 2.

$\neg p$ *unless* p in $F[H]$, irreflexivity
$(\forall i::y.i)$ *unless false* in $F[H]$, $y.i$ is a fresh variable
$\neg p \wedge (\forall i::y.i)$ *unless* p in $F[H]$, conjunction on above two
$\neg p \wedge (\forall i::y.i)$ *unless* p in $\langle \square i::u.i\rangle [F \square (s \| t)]$, observing $u.i$
$\neg p \wedge (\forall i::y.i)$ *unless* p in $F \square \langle \square i::u.i\rangle [s \| t]$, union theorem
$\neg p \wedge (\forall i::y.i)$ *unless* $p \vee (\exists i::\neg q.i)$ in $F \square \langle \square i::u.i\rangle [s \| t]$
, weakening

Proof of 3.

$\neg p \wedge q.i$ *unless false* in $F[H]$, assumption
$y.i$ *unless false* in $F[H]$, $y.i$ is a fresh variable
$\neg p \wedge q.i \wedge y.i$ *unless false* in $F[H]$, conjunction on above two
$\neg p \wedge q.i \wedge y.i$ *unless false* in $\langle \square i::u.i\rangle [F \square (s \| t)]$, observing $u.i$
$\neg p \wedge q.i \wedge y.i$ *unless false* in $F \square \langle \square i::u.i\rangle [s \| t]$, union theorem ... $(E6)$
$\neg p \wedge q.i \wedge \neg y.i$ *unless* $\neg p \wedge q.i \wedge y.i$ in $\langle \square i::u.i\rangle [F \square (s \| t)]$
, observing $u.i$
$\neg p \wedge q.i \wedge y.i$ *unless* $\neg p \wedge q.i \wedge y.i$ in $\langle \square i::u.i\rangle [F \square (s \| t)]$
, reflexivity
$\neg p \wedge q.i$ *unless* $\neg p \wedge q.i \wedge y.i$ in $\langle \square i::u.i\rangle [F \square (s \| t)]$
, disjunction on above two
$\neg p \wedge q.i$ *ensures* $\neg p \wedge q.i \wedge y.i$ in $\langle \square i::u.i\rangle [F \square (s \| t)]$
, observing $u.i$
$\neg p \wedge q.i$ *unless false* in $F[H]$, assumption
$\neg p \wedge q.i$ *ensures* $\neg p \wedge q.i \wedge y.i$ in $F \square \langle \square i::u.i\rangle [s \| t]$
, union theorem

$\neg p \wedge q.i \mapsto \neg p \wedge q.i \wedge y.i$ in $F \square \langle \square i::u.i \rangle [s \| t]$, definition of *leads-to*

$\neg p \wedge (\forall i::q.i) \mapsto \neg p \wedge (\forall i::q.i \wedge y.i)$ in $F \square \langle \square i::u.i \rangle [s \| t]$
, completion theorem with $E6$

$\neg p \wedge (\forall i::q.i) \mapsto \neg p \wedge (\forall i::y.i)$ in $F \square \langle \square i::u.i \rangle [s \| t]$
, weakening □