Program refinement in UNITY

T. E. J. Vos and S. D. Swierstra
Informatica Instituut, Utrecht University

\textit{e-mail:} \{tanja, doaitse\}@cs.uu.nl

UU-CS-2001-41
December 23, 2001
# Contents

1 Introduction .................................................................................. 3

2 Terminology and notation .............................................................. 3

3 Preliminaries: states, actions, programs ........................................ 3
   3.1 Variables, values, states ......................................................... 3
   3.2 Actions ................................................................................. 3
   3.3 Programs ............................................................................... 5
   3.4 Specifications ........................................................................ 5

4 What exactly is a refinement .......................................................... 6

5 An overview of some existing work on refinements .......................... 7
   5.1 The refinement calculus ....................................................... 7
   5.2 Sanders' mixed specifications and refinement mappings ............ 8
   5.3 A.K. Singh .......................................................................... 8
   5.4 Further reading ..................................................................... 9

6 Refinement in UNITY ..................................................................... 9

7 Another notion of refinement in UNITY .......................................... 10
   7.1 Why another notion of refinement? ........................................ 11
   7.2 The formal definition of our refinement relation ..................... 12
   7.3 Property preservation ......................................................... 15
   7.4 Guard strengthening and superposition refinement ................. 17
   7.5 Non-determinism reducing refinement ................................... 18
   7.6 Atomicity refinement ......................................................... 19

8 Application of the theory ............................................................... 19
   8.1 The communication network ............................................... 19
   8.2 Distributed hylomorphisms ............................................... 20
   8.3 A refinement ordering on the distributed hylomorphisms .......... 23
   8.4 The correctness of PLUM ..................................................... 23
   8.5 Using refinements to derive the correctness of ECHO .............. 24
   8.6 Using refinements to prove the correctness of TARRY ............ 25
   8.7 Using refinements to prove the correctness of DFS ................. 28

9 Conclusion .................................................................................... 29

A Laws of $\Rightarrow$ ..................................................................... 31

B Laws of $\sim$ .............................................................................. 31

C Proofs of the refinement theorems ............................................... 32
   C.1 Preservation of unless ......................................................... 33
   C.2 Preservation of $\Rightarrow$ ....................................................... 35
1 Introduction

Program refinement has received a lot of attention in the context of stepwise development of correct programs, since the introduction of transformational programming techniques by [Wir71, Hon72, Ger75, BD77] in the seventies. This report presents a new framework of program refinement, that is based on a refinement relation between UNITY programs. The main objective of introducing this new relation is to reduce the complexity of correctness proofs for existing classes of related distributed algorithms. It is shown, however, that this relation is also suitable for the stepwise development of programs, and incorporates most of the program transformations found in existing work on refinements.

2 Terminology and notation

Function application will be represented by a dot. In definitions we shall use $d =$ meaning “is defined by”. The complement of a set $W$ is denoted by $W^c$. A relation $R$ is bitotal on $A$ and $B$ (denoted by bitotal $R.A.B$), when for every element in $A$ there exists at least one element on $B$ to which it is related, and similarly for $B$. A relation $<$ is well-founded over $A$, when it is not possible to construct an infinite sequence of decreasing values in $A$. Universal quantification will be written like $(\forall x : P x : Q x)$ meaning for all $x$ if $P$ holds for $x$ then also $Q$. If $P$ is true for all $x$ we just write $(\forall x : Q x)$. Similar notation is used for existential quantification.

3 Preliminaries: states, actions, programs

3.1 Variables, values, states

We assume we have a universe $\text{Var}$ of program variables and a universe $\text{Val}$ of values that these variables can take. Program states will be modelled as functions that are elements of $\text{Var} \rightarrow \text{Val}$, and the set of all program states will be denoted by $\text{State}$. A state-predicate is an element of $\text{State} \rightarrow \text{Bool}$. We say that a state-predicate $p$ is confined by a set of variables $V \subseteq \text{Var}$ if $p$ does not restrict the value of any variable outside $V$. Let us write $s =_V t$, if all variables in $V$ have the same values in state $s$ and $t$ (i.e. $\forall v : v \in V : s.v = t.v$). Now we can formally define predicate confinement as follows:

Definition 3.1 CONFINEMENT

$p \subseteq V \overset{d}{=} \forall s, t : s =_V t : p s = p t$

The confinement operator is monotonic in its second argument.

Theorem 3.2 $\subseteq \text{MONOTONICITY}$

$\forall f : V \subseteq W \rightarrow (f \subseteq V) \Rightarrow (f \subseteq W)$

3.2 Actions

Actions can be (multiple) assignments or guarded (if-then) actions. Simultaneous execution of assignments is modelled by the operator $\parallel$. For example, $x, y := 1, 2 \parallel w, z := 3, 4$ equals $x, y, z, w := 1, 2, 3, 4$.

All actions in this report are assumed to be well-formed, meaning that their guard is a state-predicate, and the amount of variables at the left hand side of the := is equal to the amount of values at the right hand side.

We will assume a deep embedding of actions, i.e. the abstract syntax of actions is defined by a recursive data type $\text{ACTION}$, and their semantics is defined by a recursive function, e.g. compile, of type $\text{ACTION} \rightarrow (\text{State} \rightarrow \text{State} \rightarrow \text{Bool})$. As a consequence, we are able to obtain and reason about various components of actions. For example, we assume that we have functions $\text{guard}_{\text{of}}$ and $\text{assign}_{\text{vars}}$ that given an action returns its guard and the set of variables it assigns to respectively. Examples of these functions:

$\text{guard}_{\text{of}}(\text{if } x > 0 \land y < 10 \text{ then } x := x + 1 \parallel y := y - 1) = x > 0 \land y < 10$

$\text{assign}_{\text{vars}}(\text{if } x > 0 \land y < 10 \text{ then } x := x + 1 \parallel y := y - 1) = \{x, y\}$

Moreover, we have functions $\text{assign}$ and $\text{guard}$ that enable us to check the type of an action.

An action that is always ready to make a transition is called always enabled.
Definition 3.3  Always Enabled Action

\[ \Box_{E_{s}.A} \equiv \forall s :: (\exists t :: compile.A.s.t) \]

Multiple assignments and guarded if-then actions are always enabled. Note that this means that a guarded action with a false guard behaves like skip, i.e., the action that does not change the value of any variable.

Definition 3.4  Skip Action

For any action A, \( skip \stackrel{d}{=} if\ false\ then\ A \)

A set of variables is \( V \) ignored-by an action \( A \), denoted by \( V \rightarrow A \), if executing \( A \)'s executable in any state does not change the values of these variables. Variables in \( V \) may however be written by \( A \).

Definition 3.5  Variables Ignored-by Action

\[ V \rightarrow A \equiv \forall s,t : compile.A.s.t : s =_{V} t \]

A set of variables \( V \) is said to be invisible-to an action \( A \), denoted by \( V \Rightarrow A \), if the values of the variables in \( V \) do not influence the result of \( A \)'s executable, hence \( A \) only depends on the variables outside \( V \).

Definition 3.6  Variables Invisible-to Action

\[ V \Rightarrow A \equiv \forall s,t,s',t' : s =_{V} s' \land t =_{V} t' \land s' =_{V} t' \land compile.A.s.t : compile.A.s',t' \]

Finally, we will define two transformations on actions, namely strengthening guards and augmentation. Suppose the constructor for guarded actions of the data type ACTION is GUARD. Now we can transform an action \( A \) by strengthening its guard with state-predicate \( g \) as follows:

Definition 3.7  Strengthening Guards of Actions

\[ strengthen.guard.g.A \equiv GUARD.(g \land guard.of.A).(assign.of.A) \]

An action \( Ac \) can be combined with an assignment \( As \) to yield an augmented action:

Definition 3.8  Augmenting an Action

\[ augment.Ac.As \equiv GUARD.(guard.of.Ac).((assign.of.Ac) || As) \]

When an action \( Ac \) is transformed by augmentation to yield \( augment.Ac.As \), we say that \( Ac \) is augmented with assignment \( As \), or that \( As \) is augmented to \( Ac \). The following properties of strengthening guards and augmentation can easily be proved using the definitions given above.

Theorem 3.9  Preservation of \( \rightarrow \)

\[ \begin{align*}
  V \rightarrow A \\
  V \rightarrow strengthen.guard.g.A 
\end{align*} \]

Theorem 3.10  Preservation of \( \rightarrow \)

\[ \begin{align*}
  V \rightarrow A \land g \in V \land g \in V \\
  V \rightarrow strengthen.guard.g.A 
\end{align*} \]

Theorem 3.11

\[ strengthen.guard.g.(augment.Ac.As) = augment.(strengthen.guard.g.Ac).As \]

Theorem 3.12  Preservation of \( \rightarrow \)

\[ \begin{align*}
  V \leftarrow Ac \land V \leftarrow As \land is.assign.As \\
  V \leftarrow augment.Ac.As 
\end{align*} \]
3.3 Programs

UNITY programs $P$ are modeled by a quadruple $(aP, iniP, rP, wP)$; $aP$, is the set of actions separated by the symbol $\mathsf{[]}$; $iniP$ is the initial condition of the program; $rP$ is the set of read variables; and $wP$ the set of write variables.

A UNITY program must satisfy four syntactic requirements regarding its well-formedness: (1) The program should have at least one action; (2) A write variable is also readable; (3) The actions of a program should only write to the declared write variables; (4) The actions of a program should only depend on the declared read variables.

Using the notions of ignored-by and invisible-to we can define a well-formed "UNITY program" as an object satisfying the following predicate Unity.

Definition 3.13 Unity

$\text{Unity}(P) \equiv (aP \neq \emptyset) \land (wP \subseteq rP) \land (\forall A : A \in aP : (wP)_{A} \leftrightarrow A) \land (\forall A : A \in aP : (rP)_{A} \rightarrow A)$

A program execution of such a program is infinite, in each step an action is selected nondeterministically and executed. Selection is weakly fair, meaning that every action is selected infinitely often.

3.4 Specifications

As usual, reasoning about actions is done by means of Hoare triples [Hoa69]. If $p$ and $q$ are state-predicates, and $A$ is an action, then $\{p\} A \{q\}$ means that if $A$ is executed in any state satisfying $p$, it will end in a state satisfying $q$.

Definition 3.14 Hoare Triple

$\{p\} A \{q\} \equiv \forall s, t : p \land \text{compile}.A.s.t : q.t$

To reason about programs we will use the UNITY specification and proof logic from [CM89] augmented by [Pra95]. Safety properties can be specified by the following operators:

Definition 3.15 Unless (Safety Property)

$\triangleright p \text{ unless } q \equiv \forall A : A \in aP : \{p \land \neg q\} A \{p \lor q\}$

Definition 3.16 Stable Predicate

$\triangleright \neg p \equiv p \text{ unless } \neg \text{false}$

The following is a theorem about unless that we will need later in this report.

Theorem 3.17 Anti-Reflexivity

$\triangleright p \text{ unless } \neg p$

One-step progress properties are specified by:

Definition 3.18 Ensures (Progress Property)

$\triangleright p \text{ ensures } q \equiv (\triangleright p \text{ unless } q) \land (\exists A : A \in aP : \{p \land \neg q\} A \{q\})$

To specify general progress properties we will use Prasetya's [Pra95] reach ($\rightarrow$) and convergence ($\rightsquigarrow$) operators. The $\rightarrow$-operator is defined as the least disjunctive and transitive closure of ensures:

Definition 3.19 Reach Operator

$(\lambda p, q. J, \triangleright p \rightarrow q)$ is defined as the smallest relation $\rightarrow$ satisfying:
Lifting: \[ p \subseteq \mathcal{C} w P \land q \subseteq \mathcal{C} w P \land (\mu \vdash J) \land (\mu \vdash J \land p \text{ ensures } q) \]

Transitivity: \[ p \rightarrow q \land q \rightarrow r \]

Disjunction: \[ \forall i : W \cdot i : p_i \rightarrow q \]

where \( W \in \alpha \rightarrow \text{Val} \) characterises a non-empty set.

Many properties about \( \Rightarrow \) can be found in [Pra95], the properties we need in this report are listed in Appendix A.

The \( \leadsto \)-operator defines a restricted form of self-stabilisation, a notion first introduced by Dijkstra in [Dij74]. Roughly speaking, a self-stabilising program is a program which is capable of recovering from arbitrary transient failures of the environment in which the program is executing. Obviously such programs are very useful, although the requirement to allow arbitrary failures may be too strong. A more restricted form of self-stabilisation, called convergence, allows a program to recover only from certain failures. In [Pra95], a convergence operator is defined in terms of \( \Rightarrow \):

**Definition 3.20** \( \text{CONVERGENCE} \)

\[ J, p \vdash \text{q} \equiv q \subseteq \mathcal{C} w P \land (\exists q' :: (J, p \vdash q' \land q) \land (\mu \vdash (J \land q' \land q))) \]

Again some properties taken from [Pra95] are listed in Appendix B. Most properties are analogous to those of \( \Rightarrow \). There is, however, one property that is satisfied by \( \leadsto \) but not by \( \Rightarrow \) nor \( \equiv \), viz.

**CONJUNCTIVITY.**

### 4 What exactly is a refinement

Whereas the word *refinement* has been used in technical contexts in several related but subtly different ways, we can only give an overview after we have agreed on what is considered to be a refinement and, more important, what refinements are being considered. In Webster’s college dictionary [Inc95], refinement is defined as:

**refinement** n. 1. fineness or elegance of feeling, taste, manners, language, etc. 2. an instance of this. 3. the act or process of refining. 4. the quality or state of being refined. 5. a subtle point of distinction. 6. an improved form of something. 7. a detail or device added to improve something.

and all senses but 1 accord with the uses in computer science related contexts. We shall start by making a clear distinction between *program* refinement on the one hand and *property* refinement on the other.

Property refinement occurs within the context of the UNITY methodology for developing distributed programs. Here, a high level UNITY specification — which, within the UNITY methodology, is a property and not a program — is refined by adding more detail to it (i.e. 7 of Webster’s definition). The specification is improved in the sense that, being more detailed by exploiting some solution strategy, it gets easier to derive the final UNITY program that satisfies the initial specification. This kind of property refinement, or specification refinement is in some work also referred to as *reification* [Jac91].

Program refinement is the activity of transforming a complete program in order to improve something (i.e. 6 and 7. of Webster’s definition). This something can be the program itself (i.e. efficiency, costs, representation, etcetera), or the complexity of the correctness proof of the program. Although the *definition* that states when one program is considered to be a refinement of another differs among existing work on program refinements (see the sections below), the type or kind of program refinements (or program transformations) that are studied are generally the same. Before we discuss existing work on program refinement, we shall give the meanings of these different kinds of refinements.

**data refinement** is a program transformation where a (high-level, abstract) data structure is replaced by another (lower-level, concrete) data structure. It was first introduced in [Hoa72], and is very useful for improving the efficiency of programs.
atomicity refinement is a program transformation where a program with a coarse grain of atomicity is transformed into another program that uses a finer grain of atomicity. It is a useful transformation rule. On the one hand, proving algorithms with a coarse grain of atomicity is easier since fewer interleavings have to be considered. On the other hand, distributed algorithms that use a fine grain of atomicity are potentially faster as more processes may execute concurrently.

strengthening guards is a program transformation of which the name speaks for itself.

superposition refinement is a program transformation that, as we already discussed in Section 6, adds new functionality to an program in the form of additional variables and assignments to these variables.

The existing work that shall be discussed in the following sections is concerned with program refinements of distributed or concurrent programs.

5 An overview of some existing work on refinements

5.1 The refinement calculus

The refinement calculus originates with Ralph Back [Bac78, Bac80] and was reintroduced by Joseph Morris [Mor89] and Carrol Morgan [Mor88, MG90, Mor90]. The calculus provides a framework for systematic program development.

The main idea behind the refinement calculus is considering both specifications and code to be programs. A notion of refinement is then defined on these programs as a reflexive and transitive relation that preserves total correctness. More specifically, a program $P$ is refined by another program $P'$ (denoted by $P \leq P'$ or $P \subseteq P'$) if, when both $P$ and $P'$ are started in the same state:

- if $P$ terminates so does $P'$
- the set of final states of $P'$ is contained in the set of final states of $P$

This notion of refinement is defined using Dijkstra’s weakest pre-condition calculus [Dij76]. Note that this definition of refinement is not a property preserving refinement. All we know when $P \leq P'$ is that the input-output correctness is preserved; it does not guarantee that the behaviour of $P'$ during execution, and thus its temporal properties, will be the same as the behaviour of $P$. Since the refinement calculus was originally designed for sequential programs total correctness was sufficient. The refinement calculus has however been lifted to work on both parallel [Bac89, Bac90, Ser90, BS91, Bac93] and reactive (or distributed) [Bac90, vW92b, BvW94, BS96] systems, by using action systems [BKS83, BKS84, BKS88] to model parallel and distributed systems as sequential programs. Although preserving total correctness is also sufficient for parallel systems, stepwise refinement of reactive or distributed systems also requires preservation of temporal properties. Consequently, in [Bac90, vW92b, BvW94, BS96] the notion of refinement was extended such that the preservation of temporal properties was guaranteed.

The development of a program within the refinement calculus framework consists of a sequence of correctness (or in the case of distributed systems, temporal properties) preserving refinement steps, starting with an initial high-level specification and ending with an efficient executable program. These correctness preserving refinement steps are formulated as program transformations rules $t \in \text{programs} \rightarrow \text{programs}$ and added to the refinement calculus framework by proving theorems of the form:

\[
\forall P \in \text{programs} :: \quad \text{Verification Conditions hold for } P \quad \Rightarrow \quad P \leq t.P
\]

In other words if certain verification conditions are satisfied, then applying rule $t$ to program $P$ is a correctness (and in the case of distributed systems, temporal properties) preserving refinement step. Many transformation rules can be found in [Bac88, Bac89, BvW89, BvW90, Bac90, Ser90, BS91, vW92a, vW92b, Bac93, BS96, SW97, BvW98, BKS98], concerning among others, data refinement, guard strengthening, superposition refinement, and atomicity refinement (or changing the granularity).

Some other references on uses of the refinement calculus for distributed systems include [SW94a, SW94b, SW96], where the refinement steps are applied backwards in order to obtain a formal approach to reverse engineering distributed algorithms. In [Wal96, BW96, WS96, Wal98a, Wal98b, BW98] action systems and their refinements are formalised and applied within the B-method [Abr96].

\text{In [Bac81] a notion of partial correctness preserving refinement is studied.}
5.2 Sanders’ mixed specifications and refinement mappings

Sanders [San90] has introduced a mixed specification technique (called \textit{mspecs}) to define a notion of program refinement in UNITY. An \textit{mspec} incorporates both program text and a set of program properties. More specifically, an \textit{mspec} consists of a \textit{declare} section that contains a list of variables together with their types (the Cartesian product of these variables is referred to as the state space of the \textit{mspec}); an \textit{initially} section that contains a predicate that specifies the allowed initial values of the variables; an \textit{assign} section that contains a set of conditional assignment statements that, in an operational view, constrain the behaviour of the program by specifying allowed state changes; a \textit{property} section containing a set of program properties (expressed in a modified version of the UNITY logic) that further constrain the allowed state changes, and for the progress properties, the allowed sequence of state changes.

Consequently, if the \textit{assign} section is empty, an \textit{mspec} is a standard UNITY specification, and if the \textit{properties} section is empty an \textit{mspec} is a standard UNITY program. An \textit{mspec} is called implementable when all properties in the property section can be proved to hold for the actions in the \textit{assign} section.

A benefit of specifying UNITY programs with a mixed specification is the following. Some desired program properties, like e.g. safety properties, are easier and more intuitively expressed using statements instead of logic, while others (usually progress properties) are better specified using logics [Lam83, Lam89]. In an \textit{mspec} one can benefit from both possibilities, which is good since getting a specification right in the first place is crucial and not always easy.

A notion of refinement is defined on \textit{mspecs} which is based on a refinement mapping [Lam83, LS84, AL88, Lam91, Lam96] \(M\) from the state space of the refinement to the state space of the original. It is denoted by \((G \text{ refines } F)_{M}\), and informally means:

- all initial conditions of \(G\) are mapped by \(M\) to the initial conditions on \(F\)
- if a state change from \(y_{0}\) to \(y_{1}\) is permitted by the assignments in the \textit{assign} section of \(G\), then either a state change from \(M_{y_{0}}\) to \(M_{y_{1}}\) is permitted by the assignments in the \textit{assign} section of \(F\), or \(M_{y_{0}} = M_{y_{1}}\).
- all properties of \(F\) are implied by the properties of \(G\)

Using this definition, several theorems are proved that state under which conditions a property that holds in an \textit{mspec} can be considered to hold in a refinement. To give an indication of what these theorems look like, the \(\rightarrow\) preservation theorem is copied below [San90, page 13]:

\[
\forall i : (\forall r_{i} \text{ ensures } q_{i} \text{ is used in the proof of } r_{i} \rightarrow q_{i} ) : \quad r_{i} \circ M \text{ ensures } q_{i} \circ M
\]

\[
\rightarrow \quad p \circ M \rightarrow q \circ M
\]

Similar theorems are given for preservation of \textit{unless}, \textit{ensures}, and fixed points. Moreover, a theorem is proved that states when the program transformation of replacing a shared variable by a message communication system is a property preserving (data) refinement. Stepwise derivation of programs within this framework now consists of a sequence of refined \textit{mspecs}, starting with an \textit{mspec} containing a high level of abstraction, and ending up with an \textit{mspec} that is implementable.

5.3 A.K. Singh

In [SO89, Sin89, Mis90, Sin91, Sin93] refinement of UNITY programs is investigated. Notions of property preserving and total correctness preserving (or fixed-point preserving, as it is called in [Sin93]) refinements are defined\(^3\) as follows: [Sin93, page 311]:

Let \(F\) and \(G\) be two programs. \(G\) is a \textit{property-preserving} refinement of \(F\) iff for all predicates \(p, q\), the following two assertions hold:

- \(\forall r \vdash p \text{ unless } q \Rightarrow \quad \forall \rightarrow r \vdash p \text{ unless } q\)
- \(\forall r \vdash p \rightarrow q \Rightarrow \quad \forall \rightarrow r \vdash \rightarrow q\)

Similarly, \(G\) is a \textit{fixed-point preserving} refinement of \(F\) iff

- \(\forall r \vdash \text{true } \rightarrow \text{FP}G \Rightarrow \quad \forall \rightarrow r \vdash \text{true } \rightarrow \text{FP}G\)
- \(\text{FP}G \land \text{S1}G \Rightarrow \quad \text{FP}F \land \text{S1}F\)

where \(\text{FP}P\) is the fixed point of a program \(P\), i.e. it characterises the collection of states that are invariant under the execution of every statement in \(P\); \(\text{S1}P\) denotes the strongest invariant of a program \(P\), i.e. it denotes the set of states reachable from the initial state.

\(^3\)The modified version was defined to eliminate the need of the substitution axiom [San91]

\(^3\)The definitions of \textit{unless}, \textit{ensures}, and \(\rightarrow\) of Sanders’ logic [San91] are used.
Having defined these two notions of refinement, theorems are proved stating under which conditions certain program transformations are property-preserving and fixed-point preserving refinements. To give an indication of what these theorems look like, a theorem, stating the verification conditions under which strengthening the guard of a program is a property and fixed-point preserving refinement, looks like: [Sin89, page 1] [Mis90] [Sin93, page 519]

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

- $\mathit{r} \vdash p \implies q$
- There exists a non-increasing function $g$ from the program variables to a well-founded set such that $\mathit{r} \vdash (g = k \land q)$ unless $(-p \lor g < k)$, for all $k$

In [Sin93] similar theorems are proved for program transformations like data refinement and atomicity refinement, and applied to a number of examples.

5.4 Further reading

For some other work on refinement concepts within the UNITY (or a UNITY-like) framework, the reader can for example read [ZGG90, Jon90, Kor91, Udi95, UK96, Din97, GKS98].

6 Refinement in UNITY

Within the UNITY framework [CM89] two refinements are distinguished: restricted union superposition, and augmentation superposition. It is recognised in [CM89] that the lack of appropriate syntactic mechanisms limits the algebraic treatment of superposition. Consequently, the description of superposition refinement in [CM89] is rather informal. Since in this report we assume a deep embedding of actions, we have more appropriate syntactic mechanisms which enable us to give a less informal treatment of superposition.

In [CM89], the *restricted union superposition* rule states that an action $A$ may be added to an underlying program provided that $A$ does not assign to the underlying variables. Here we split this into two parts:

- first, defining the actual transformation of the program;
- second, proving under which conditions this transformation preserves the properties of the underlying program.

Let $A$ be an action from the universe $\text{ACTION}$, and let $iA$ be a state-predicate describing the initial values of the superposed variables, then a program $P$ can be refined by restricted union superposition using the transformation formally defined by:

**Definition 6.1 Restricted union superposition**

$$\text{RU}_{\text{superpose}} \text{DEF}$$

Let $A \in \text{ACTION}, iA$ be a state-predicate, and $P$ be a program:

$$\text{RU}_S.P.A.iA = P \parallel ([\{A\}, iA, (\text{assign}\_\text{vars}.A), (\text{assign}\_\text{vars}.A))]$$

Theorems stating that properties are preserved under restricted union superposition are stated below for arbitrary programs $P$, actions $A$, and state-predicates $p, q, J$. Note that instead of requiring that the superposed action $A$ does not write to the underlying variables, it is sufficient to require that the write variables of the underlying program are ignored by the action $A$.

**Theorem 6.2 Preservation of unless and ensures**

$$\text{RU}_{\text{superpose}} \text{PRESERVES UNLESS}$$

$$\frac{p \bigland w P \land q \bigland w P \leftrightarrow A}{\mathit{r} \vdash p \text{ unless } q} \quad \frac{\mathit{r} \vdash p \text{ unless } q}{\mathit{ru}_{\text{superpose}}.P.A.iA \vdash p \text{ unless } q}$$

$$\text{RU}_{\text{superpose}} \text{PRESERVES ENSURES}$$

$$\frac{p \bigland w P}{\mathit{r} \vdash p \text{ ensures } q} \quad \frac{\mathit{r} \vdash p \text{ ensures } q}{\mathit{ru}_{\text{superpose}}.P.A.iA \vdash p \text{ ensures } q}$$

9
Theorem 6.3 Preservation of $\Rightarrow$

\[
\begin{align*}
J \not\vdash P \land wP & \not\vdash A \\
J \not\vdash p \implies q & \Rightarrow J_{(\underaccent{\hat{\gamma}})} \vdash p \implies q \\
J \not\vdash p \implies q & \Rightarrow J_{(\underaccent{\hat{\gamma}})} \vdash p \implies q
\end{align*}
\]

In [CM89], the augmentation superposition rule states that an assignment $As$ that does not assign to the underlying variables can be augmented to any assignment or assignment-part of actions of the underlying program. Again, we first define the actual transformation on the program, and second, prove theorems stating when properties are preserved. Let $As$ be an assignment from the universe $ACTION$, and let $iA$ be a state-predicate describing the initial values of the superposed variables, then a program $P$ can be refined by augmentation superposition using the transformation rule formally defined by:

**Definition 6.4 Augmentation Superposition**

Let $As \in ACTION, iA$ be a state-predicate, $ACs \subseteq ACTION$, and $P$ be a program:

\[
\begin{align*}
\text{AUG}_{S,P,ACs,As,iA} & = (\{ Ac \mid Ac \in AP \land Ac \not\in ACs \} \\
& \cup \{ \text{augment.Ac.As} \mid Ac \in AP \land Ac \in ACs \}, \\
\text{iniP} \land iA, \\
\text{rP} \cup (\text{assign.vars.As}), \\
\text{wP} \cup (\text{assign.vars.As}) \})
\end{align*}
\]

Theorems stating that properties are preserved under augmentation superposition are listed below for arbitrary $As \in ACTION$, state-predicates $iA$, programs $P$, and $ACs \subseteq ACTION$. Note again that instead of requiring that the assignment $As$ does not write to the underlying variables, it is sufficient to require that the write variables of the underlying program are ignored by $As$.

**Theorem 6.5 Preservation of unless and ensures**

\[
\begin{align*}
J \not\vdash P \land wP & \not\vdash A \land \text{is assign.As} \\
J \not\vdash p \text{ unless } q & \Rightarrow J_{(\underaccent{\hat{\gamma}})} \vdash p \text{ unless } q \\
J \not\vdash p \text{ ensures } q & \Rightarrow J_{(\underaccent{\hat{\gamma}})} \vdash p \text{ ensures } q
\end{align*}
\]

**Theorem 6.6 Preservation of $\Rightarrow$**

\[
\begin{align*}
J \not\vdash P \land wP & \not\vdash A \land \text{is assign.As} \\
J \not\vdash p \implies q & \Rightarrow J_{(\underaccent{\hat{\gamma}})} \vdash p \implies q \\
J \not\vdash p \implies q & \Rightarrow J_{(\underaccent{\hat{\gamma}})} \vdash p \implies q
\end{align*}
\]

7 Another notion of refinement in UNITY

Like Sanders, but unlike Back and Singh, our refinement relation is *not* defined to be property or correctness preserving, and accordingly additional theorems have to be proved that state conditions under which properties of a program are preserved in its refinement. These conditions, however, do not look like the ones in Sanders, but relate to the verification conditions of the theorems in Back and Singh that argue about the property preservation of specific program transformation rules. The main difference between our refinement relation and the ones described in the previous sections, is that its purpose it not the stepwise derivation of correct programs but the reduction of complexity of correctness proofs of existing classes of related algorithms. The next section shall exemplify this.
7.1 Why another notion of refinement?

Guard strengthening and superposition are transformations for the step-wise development of programs, the formalisation of which was discussed in Section 6. This section exemplifies why these program refinements are sometimes insufficient to refine a program, and hence motivates the introduction of our new refinement relation.

Suppose we have a class of similar algorithms that seemingly establish the same progress in various ways. Most of the time, algorithms in such a class differ by having different mechanisms or control structures that influence their control flow and degree of non-determinism. Sometimes, however, adding such a mechanism or control structures, does not consist of one transformation, but a sequence (or composition) of transformations which as a whole are a property preserving transformation but on their own they are not. Consider, for example the following simple UNITY program which is in the class of algorithms that, started with initial values \( x = 0 \) and \( y = 0 \), increments both \( x \) and \( y \) until they have the value 10.

```
prog P
read \{x, y\}
write \{x, y\}
init x = 0 \land y = 0
assign if x \leq 10 then x := x + 1   P_x
[ ] if y \leq 10 then y := y + 1   P_y
```

Figure 1: Program \( P \) that increments both \( x \) and \( y \) until they have the value 10

It is easy to prove that \( \text{true} \vdash x = 0 \land y = 0 \rightarrow x = 10 \land y = 10 \) (see Figure 3). Another algorithm in this class is one that reduces the non-determinism of \( P \) in such a way that the value of \( x \) and \( y \) are incremented in an alternating way. Obviously, this more deterministic program also satisfies (for some \( J \) \( J \vdash x = 0 \land y = 0 \rightarrow x = 10 \land y = 10 \), and can be constructed by introducing a variable \( x_{\text{turn}} \) of type \( \text{bool} \) - the value of which indicates that it is \( x \)'s turn - and transforming \( P \) as follows:

```
prog Q
read \{x, y, x\_{\text{turn}}\}
write \{x, y, x\_{\text{turn}}\}
init x = 0 \land y = 0 \land x\_{\text{turn}} = \text{true}
assign if x < 10 \land x\_{\text{turn}} then x := x + 1 || x\_{\text{turn}} := \neg x\_{\text{turn}}   Q_x
[ ] if y < 10 \land \neg x\_{\text{turn}} then y := y + 1 || x\_{\text{turn}} := \neg x\_{\text{turn}}   Q_y
```

Figure 2: Program \( Q \); reducing \( P \)'s non-determinism

The machinery for superposition refinements in UNITY, formalised in Section 6, is inadequate for proving that this transformation is a property preserving one. This is because if we augment the \( P_x \) with assignment \( x\_{\text{turn}} := \neg x\_{\text{turn}} \) to yield the program \( \text{AUG}_x.S.P_x \{x\_{\text{turn}} := x\_{\text{turn}} \} (x\_{\text{turn}} := \neg x\_{\text{turn}}, (x\_{\text{turn}} = \text{true})) \), then we cannot subsequently augment action \( P_y \) of \( \text{AUG}_y.S.P_x \{x\_{\text{turn}} := x\_{\text{turn}} \} (x\_{\text{turn}} := \text{false}, (x\_{\text{turn}} = \text{true})) \) with the assignment \( x\_{\text{turn}} := \neg x\_{\text{turn}} \) and prove that the properties are preserved, since the write variables of \( \text{AUG}_x.S.P_x \{x\_{\text{turn}} := x\_{\text{turn}} \} (x\_{\text{turn}} := \text{true}) \) (i.e. \( w.P \cup \{x\_{\text{turn}}\} \)) are not ignored by the assignment \( x\_{\text{turn}} := \neg x\_{\text{turn}} \). Consequently, the formalisation of the UNITY superposition rules are not sufficient to prove preservation of properties under these kind of non-determinism reducing refinements. However, these refinements are very powerful for reducing the complexity of a correctness proof for a class of distributed programs. Non-deterministic programs are often easier to prove than more deterministic ones, since simplicity is gained by avoiding unnecessary determinism. To illustrate this we have displayed the proof of \( x = 0 \land y = 0 \rightarrow x = 10 \land y = 10 \) for programs \( P \) and \( Q \) in Figures 3 and 4 respectively. It is not hard to see that the proof in Figure 3 is simpler than the proof in Figure 4. One reason for this is that, because of \( P \)'s freedom to increment \( x \) and \( y \) whenever it wants (i.e. non-determinism), we are able
true $\gamma \vdash x = 0 \land y = 0 \Rightarrow x = 10 \land y = 10$
\[\Leftrightarrow (\Rightarrow \text{CONJUNCTION (B.1132)}, \Rightarrow \text{SUBSTITUTION (B.231)})\]
true $\gamma \vdash x = 10 \land y = 10$ \iff $\gamma \vdash \neg \neg x = 10$ 

We continue with the first conjunct, the proof of the second conjunct is similar.
true $\gamma \vdash x = 10 \land y = 10$
\[\Leftrightarrow (\Rightarrow \text{BOUNDED PROGRESS (B.1232)}, \text{and} \ \gamma \vdash \Box x = 10)\]
true $\gamma \vdash x = 10 \land (10 - x = k) \Rightarrow (x = 10 \land (10 - x < k)) \lor x = 10$
\[\Leftrightarrow (\Rightarrow \text{CASE DISTINCTION (B.632)}, x = 10 \lor x \neq 10, \Rightarrow \text{REFLEXIVITY (B.432)}, \ \gamma \vdash \Box x = 10, \text{and} \ \Rightarrow \text{SUBSTITUTION (B.231)})\]
true $\gamma \vdash x < 10 \land (10 - x = k) \Rightarrow x \leq 10 \land (10 - x < k)$
\[\Leftrightarrow (\Rightarrow \text{INTRODUCTION (B.332)}, \ \gamma \vdash \Box x < 10 \land (10 - x < k))\]
true $\gamma \vdash x < 10 \land (10 - x = k)$ ensures $x \leq 10 \land (10 - x < k)$

Figure 3: Proof of true $\gamma \vdash x = 0 \land y = 0 \Rightarrow x = 10 \land y = 10$

to decompose the proof obligation $x = 0 \land y = 0 \Rightarrow x = 10 \land y = 10$ into the simpler proof obligations $x = 0 \Rightarrow x = 10$ and $y = 0 \Rightarrow y = 10$. For program $Q$ this is an inefficacious proof strategy because $x$ and $y$ cannot be increased independently. Another reason is that, because of $Q$’s restricted freedom to increase $x$ and $y$ (i.e., determinism), additional case distinctions on whether it is $x$’s turn or not have to be made in order to be able to prove that progress can indeed be made.

Although this is just a simple example, it suggest that the total proof effort can be significantly reduced if we have a refinement relation supporting non-determinism reducing refinement. Since then, instead of laboriously proving properties directly for a more deterministic program $Q$, we can reduce the proof-complexity by proving these properties for the least deterministic variant $P$ of $Q$, and conclude that these properties also hold for $Q$.

### 7.2 The formal definition of our refinement relation

We start by defining the refinement relation between two actions. Suppose we have two actions $A_t, A_r \in \text{ACTION}$, a state-predicate $J$, and a set of variables $V$, we say that $A_t$ is refined by $A_r$, or $A_r$ refines $A_t$, with respect to $V$ and $J$ (denoted by $A_t \sqsubseteq_{V,J} A_r$), when:

- the conjunction of $J$ with the guard of $A_r$ is stronger then the guard of $A_t$.
- the results of $A_t$ and $A_r$, both executed in the same state $s$ where $J.s$ holds, on the variables in $V$ are the same.

**Definition 7.1** ACTION REFINE

Let $A_t$ and $A_r$ be two actions from the universe ACTION, $J$ be a state predicate, and $V$ be a set of variables, then action refinement is defined as follows:

\[
A_t \sqsubseteq_{V,J} A_r \iff \forall s : \text{guardof}_A.r.s \land J.s \Rightarrow \text{guardof}_A.t.s \\
\land \forall s, t, t' : (\text{compile}_A.t.s.t \land \text{compile}_A.r.s.t'.\land \text{guardof}_A.r.s \land J.s) \Rightarrow t =_V t'
\]

The fact that action refinement is reflexive and transitive is captured by the following theorems.

**Theorem 7.2** ACTION REFINE REFLEXIVITY

For all $A \in \text{ACTION}$, state-predicates $J$, and sets of variables $V$:\n
\[A \sqsubseteq_{V,J} A\]

**Theorem 7.3** ACTION REFINE TRANSITIVITY

For all $A_1, A_2, A_3 \in \text{ACTION}$, state-predicates $J_2$ and $J_3$, and sets of variables $V_1, V_2$, and $V_3$:

\[
J_3 \Rightarrow J_2 \land V_3 \subseteq V_1 \land V_3 \subseteq V_2 \land A_1 \sqsubseteq_{V_1,J_2} A_2 \land A_2 \sqsubseteq_{V_2,J_3} A_3
\]

\[A_1 \sqsubseteq_{V_1,J_3} A_3\]
Take $J = \neg x_{\text{turn}} \Rightarrow y = x - 1 \land (x_{\text{turn}} \Rightarrow y = x)$, and prove that $\varnothing \vdash J$.

$$J \vdash x = 0 \land y = 0 \Rightarrow x = 10 \land y = 10$$
$$\Leftarrow (\neg \text{Substitution (B.21)})$$

$$J \vdash x \leq 10 \land y \leq 10 \Rightarrow x = 10 \land y = 10$$
$$\Leftarrow (\rightarrow \text{Bounded Progress (B.12)}, \text{ and } \varnothing \vdash x = 10 \land y = 10)$$

$$J \vdash x \leq 10 \land y \leq 10 \land (20 - x - y = k) \Rightarrow (x \leq 10 \land y \leq 10 \land (20 - x - y < k)) \lor (x = 10 \land y = 10)$$
$$\Leftarrow (\rightarrow \text{Case Distinction (B.6.2)}, y = 10 \lor y \neq 10, \rightarrow \text{Introduction (B.3.3)}, \text{ and }$$

$$\rightarrow \text{Substitution (B.2.1)}$$

$$J \vdash x \leq 10 \land y \leq 10 \land (2 - x - y = k) \Rightarrow (x \leq 10 \land y \leq 10 \land (2 - x - y < k))$$
$$\Leftarrow (\rightarrow \text{Case Distinction (B.6.2)}, x = 10 \lor x \neq 10, \rightarrow \text{Substitution (B.2.1)})$$

$$J \vdash x = 10 \land y = 10 \land (20 - x - y = k) \Rightarrow (x \leq 10 \land y \leq 10 \land (20 - x - y < k))$$

The first conjunct can be proved by $\rightarrow \text{Introduction (B.3.3)}$, since $(J \land x = 10 \land y \leq 10) \implies \neg x_{\text{turn}}$, and thus

\[ \varnothing \vdash J \land x = 10 \land y < 10 \land (20 - x - y = k) \text{ ensures } x \leq 10 \land y \leq 10 \land (20 - x - y < k) \]

We continue with the second conjunct as follows:

$$J \vdash x < 10 \land y \leq 10 \land (20 - x - y = k) \Rightarrow x \leq 10 \land y \leq 10 \land (20 - x - y < k)$$
$$\Leftarrow (\rightarrow \text{Case Distinction (B.6.2)})$$

$$J \vdash x < 10 \land y < 10 \land (20 - x - y = k) \land x_{\text{turn}} \Rightarrow x \leq 10 \land y \leq 10 \land (20 - x - y < k)$$

\[ \Leftarrow (\rightarrow \text{Introduction (B.3.3)}) \text{ on both conjuncts} \]

$$\varnothing \vdash J \land x < 10 \land y \leq 10 \land (20 - x - y = k) \land x_{\text{turn}} \text{ ensures } x \leq 10 \land y \leq 10 \land (20 - x - y < k)$$

\[ \Leftarrow (\rightarrow \text{Introduction (B.3.3)}) \text{ on both conjuncts} \]

$$\varnothing \vdash J \land x < 10 \land y < 10 \land (20 - x - y = k) \land \neg x_{\text{turn}} \text{ ensures } x \leq 10 \land y \leq 10 \land (20 - x - y < k)$$

$$\varnothing \vdash J \land x < 10 \land y < 10 \land (20 - x - y = k) \land \neg x_{\text{turn}} \text{ ensures } x \leq 10 \land y \leq 10 \land (20 - x - y < k)$$

\[ \Box \]

Figure 4: Proof of $J \vdash x = 0 \land y = 0 \Rightarrow x = 10 \land y = 10$

Next, we define our relation of program refinement. $P$ is refined by $Q$, or $Q$ refines $P$, with respect to some relation $R$ and state-predicate $J$, (denoted by $P \sqsubseteq_{R,J} Q$), if we can decompose the actions of program $Q$ into $aQ_1$ and $aQ_2$, such that

- $R$ is a bitotal relation on the two sets of actions $aP$ and $aQ_1$, i.e. for every action $A_P$ in $aP$ there exists at least one action in $aQ_1$, to which $aP$ is related by $R$, and similarly for every action $A_Q$ in $aQ_1$ there exists at least one action in $aP$ to which $A_Q$ is related by $R$.
- for all actions $A_P$ of $aP$ and $A_Q$ of $aQ_1$ that are related to each other by $R$ (i.e. $A_P \mathcal{R} A_Q$ holds), we can prove that $A_Q$ refines $A_P$ with respect to the write variables of $P$ and state-predicate $J$.
- the actions of $Q$ that are in $aQ_2$ refine skip with respect to the write variables of $P$ and $J$.

For those readers that are geared to pictures, in Figure 5 a depiction of program refinement is given. The formal definition of program refinement now reads:

**Definition 7.4 Program Refinement**

Let $P$ and $Q$ be two UNITY programs, $R$ be a relation, and $J$ be a state predicate, then program refinement is defined as follows:

$$P \sqsubseteq_{R,J} Q = \exists aQ_1, aQ_2 :: aQ = aQ_1 \cup aQ_2 \land \text{bitotal } R.aP.aQ_1$$

$$\land$$

$$\forall A_P : A_P \in aP \land A_P \mathcal{R} A_Q : A_P \sqsubseteq_{wP,J} A_Q$$

$$\land$$

$$\forall A_Q : A_Q \in aQ_2 : \text{skip} \sqsubseteq_{wP,J} A_Q$$

Note that $P \sqsubseteq_{R,J} Q$ does not say anything about $Q$ inheriting properties or correctness from $P$. Nor does it say anything about the explicit program transformations that were (or could have been) applied to $P$ in order to obtain $Q$. Moreover note that, opposed to superposition refinement, $P \sqsubseteq_{R,J} Q$, does not necessarily imply that $wP \sqsubseteq wQ$. Consider the two programs $P$ and $Q$ in Figure 6. Suppose $z$ and
Figure 5: Program refinement in a picture.

\[
\begin{align*}
\text{prog } & P \\
\text{read } & \{x, y, z\} \\
\text{write } & \{x, y, z\} \\
\text{init } & b = \text{true} \\
\text{assign } & x := x + 1 \ aP_1 \\
& y := y + 1 \ aP_2 \\
& z := z \ aP_3
\end{align*}
\]

\[
\begin{align*}
\text{prog } & Q \\
\text{read } & \{x, y, w\} \\
\text{write } & \{x, y, w\} \\
\text{init } & b = \text{true} \\
\text{assign } & \text{if } x \leq 15 \text{ then } x := x + 1 \ aQ_1 \\
& \text{if } y \leq 20 \text{ then } y := y + 1 \ aQ_2 \\
& w := w + 1 \ aQ_3
\end{align*}
\]

Figure 6: Q refines P

\( w \) are different variables, then it can easily be seen that for any state-predicate \( J \), and relation \( \mathcal{R} \) defined by \( \mathcal{R} = \{(aP_i, aQ_i) \mid i = 1, 2\} \), it holds that \( P \sqsubseteq_{\mathcal{R}, J} Q \). However, since \( z \) and \( w \) are different variables, \( wP \subseteq wQ \) does not hold.

The following theorems state that program refinement is reflexive and under certain conditions also transitive.

**Theorem 7.5** Program refinement Reflexivity

For all programs \( P \), and state-predicate \( J \):

\( P \sqsubseteq_{\mathcal{R}, J} P \)

**Theorem 7.6** Program refinement Transitivity

For all programs \( P_1, P_2, P_3 \), and state-predicates \( J_2, J_3 \):

\[
J_3 \Rightarrow J_2 \land wP_1 \subseteq wP_2 \land P_1 \sqsubseteq_{\mathcal{R}_1, J_2} P_2 \land P_2 \sqsubseteq_{\mathcal{R}_2, J_3} P_3
\]

\[
P_1 \sqsubseteq_{\mathcal{R}_1 \circ \mathcal{R}_2, J_3} P_3
\]

Reflexivity, and transitivity are necessary properties of a refinement relation, in order to make the latter suitable for the step-wise derivation of programs [Bac88]. However, our definition of refinement is not purely transitive in the sense that additional requirements on the component programs are demanded in the premises of Theorem 7.6 stating transitivity of \( \sqsubseteq \). Suppose we want to derive program \( P_{n+1} \) from \( P_1 \) \((n > 1)\) by the following sequence of refinements:

\[
P_1 \sqsubseteq_{\mathcal{R}_1, J_2} P_2 \sqsubseteq_{\mathcal{R}_2, J_3} P_3 \sqsubseteq_{\mathcal{R}_3, J_4} P_4 \ldots \sqsubseteq_{\mathcal{R}_{n-1}, J_n} P_n \sqsubseteq_{\mathcal{R}_n, J_{n+1}} P_{n+1}
\]

in order to conclude that

\[
P_1 \sqsubseteq_{\mathcal{R}_1 \circ \ldots \circ \mathcal{R}_{n+1}} P_{n+1}
\]

we have to prove that:
• the write variables of the program \( P_i \) in intermediate step \( P_i \subseteq R_{i \cdot 0 \ldots i \cdot 1}, J_i+1 \) \( P_{i+1} \) are included or equal to the write variables of program \( P_{i+1} \).

• the predicate \( J_{i+1} \) (which shall usually correspond to the strongest invariant of the program \( P_{i+1} \)) must be stronger than the predicate \( J_i \) (thus the strongest invariant of program \( P_i \)).

Consideration of the fact that the underlying transformations of these intermediate refinement steps are superposition, guard strengthening and atomicity refinement (see Section 7.6), these requirements are very natural. Consequently, our definition of refinement is very suitable for stepwise derivation and verification of distributed programs in UNITY.

7.3 Property preservation

Safety properties \( p \) unless \( q \), and \( J \), where \( p, q \) and \( J \) do not name any superposed variable, are always preserved under refinement of two UNITY programs.

**Theorem 7.11** unless preservation

\[
\begin{align*}
P \subseteq R, J & \quad Q \land \text{Unity} \quad P \land \text{Unity}, Q \land (\varphi \vdash J_Q) \land (J_Q \Rightarrow J) \\

\exists W : (wQ = wP \cup W) \land (p \not\subseteq W^c) \land (q \not\subseteq W^c) \\

\vdash p \text{ unless } q \Rightarrow \varphi \vdash (J_Q \land p) \text{ unless } q
\end{align*}
\]

**Theorem 7.12** \( \circ \) preservation

\[
\begin{align*}
P \subseteq R, J & \quad Q \land \text{Unity} \quad P \land \text{Unity}, Q \land (\varphi \vdash J_Q) \land (J_Q \Rightarrow J) \\

\exists W : (wQ = wP \cup W) \land (p \not\subseteq W^c) \land (q \not\subseteq W^c) \\

\vdash p \circ q \Rightarrow \varphi \vdash (J_Q \land p)
\end{align*}
\]

The conditions \((p \not\subseteq W^c)\) and \((q \not\subseteq W^c)\), in the premises of the two theorems above, state that the values of state-predicates \( p \) and \( q \) do not depend on the values of the variables in \( W \). Note that when \( W \) is the set of variables that are superposed up on program \( P \), these conditions are weaker than stating that \( p \) and \( q \) do not name any superposed variable.

Preservation of one-step progress properties (i.e. ensures) cannot be proved under our definition of refinement. Fortunately, preservation of reach and convergence properties can be proved, and in most situations these are all that are required.

Figure 7 shows the theorems stating verification conditions under which general progress properties are preserved by refinements. Theorem 7.7 is a generalisation of the theorem given in [Sin93] mentioned earlier in Section 5.3. It states verification conditions for property preservation not only under strengthening the guard of one action in a program, but under multiple compositions of guard strengthening, superposition and atomicity refinements on various actions in the program. Informally this theorem states that when a UNITY program \( Q \) refines \( P \) with respect to relation \( R \) and \( J \), then the progress properties \( p \rightsquigarrow q \) and \( p \rightsquigarrow q \) under the stability of predicate \( J_P \) in program \( P \), are preserved under the stability of predicate \( J_P \land J_Q \) in program \( Q \), provided that the following verification conditions hold:

- \((J_P \land J_Q)\) is stable in \( Q \).
- \((J_P \land J_Q)\) implies \( J \).
- \( p \) nor \( q \) depend on the values of the variables in \( W \).
- the guards of those actions \( A_Q \) of \( Q \) that are related by \( R \) to one or more actions from \( P \) are confined by the write variables of \( Q \).
- for all actions \( A_P \) of program \( P \); if the guard of \( A_P \) holds in \( Q \), then eventually there will exists an action \( A_Q \) of \( Q \) that is related to \( A_P \) by \( R \), and the guard of which becomes true in \( Q \). Consequently, if \( A_P \) can make progress in \( P \), then eventually there exists at least one action of \( A_Q \) of \( Q \) that, when executed in \( P \), can make the same progress on the write variables of \( P \) as \( A_P \) does when executed in \( P \).

Note that this requirement is not enough to guarantee that \( A_Q \) indeed makes the same progress as \( A_P \), since between the point in time that the guard of \( A_Q \) becomes true, and the actual execution of \( A_Q \) it is possible that the guard of \( A_Q \) is prematurely falsified and no progress is made by \( A_Q \) whatsoever. The next (and last) verification condition states that this premature falsification of the guard of \( A_Q \) cannot happen infinitely and hence ensures that eventually \( A_Q \) will make the same progress as \( A_P \) on the write variables of program \( P \).
Let $<$ be a well-founded relation over some set $A$, and $M \in \text{State} \to A$.

**Theorem 7.7**

$$ P \subseteq_{r,J} Q \land \text{Unity}.P \land \text{Unity}.Q \land (\text{path} \land J_P \land J_Q) \land (J_P \land J_Q \Rightarrow J) $$

$$ \forall W : (wQ = wP \cup W) \land (J_P \subseteq W) \land (wP \subseteq W) $$

$$ \forall A_Q : A_Q \in aQ \land (\exists A_P : (A_P \in aP) \land (A_P \cap A_Q) : (\text{guard}.A_P \cap A_Q) \land \text{guard}.A_Q) $$

$$ \forall M : (M \subseteq W) \land (M \subseteq W) $$

$$ \forall k : k \in A : \text{path}(J_P \land J_Q \land M = k) $$

$$ (J_P \text{path} \Rightarrow q) \Rightarrow (J_P \land J_Q \text{path} \Rightarrow q) \land (J_P \land J_Q \text{path} \Rightarrow q) $$

**Theorem 7.8**

$$ P \subseteq_{r,J} Q \land \text{Unity}.P \land \text{Unity}.Q \land (\text{path} \land J_P \land J_Q) \land (J_P \land J_Q \Rightarrow J) $$

$$ \forall W : (wQ = wP \cup W) \land (J_P \subseteq W) \land (wP \subseteq W) $$

$$ \forall A_Q : A_Q \in aQ \land (\exists A_P : (A_P \in aP) \land (A_P \cap A_Q) : (\text{guard}.A_P \cap A_Q) \land \text{guard}.A_Q) $$

$$ \forall A_P : A_P \in aP : (J_P \land J_Q) \text{path} \Rightarrow (\exists A_Q : (A_Q \cap A_P) \land \text{guard}.A_Q) $$

$$ \forall M : (M \subseteq W) \land (M \subseteq W) $$

$$ \forall k : k \in A : \text{path}(J_P \land J_Q \land M = k) $$

$$ (J_P \text{path} \Rightarrow q) \Rightarrow (J_P \land J_Q \text{path} \Rightarrow q) \land (J_P \text{path} \Rightarrow q) $$

**Theorem 7.9**

$$ P \subseteq_{r,J} Q \land \text{Unity}.P \land \text{Unity}.Q \land (\text{path} \land J_P \land J_Q) \land (J_P \land J_Q \Rightarrow J) $$

$$ \forall W : (wQ = wP \cup W) \land (J_P \subseteq W) \land (wP \subseteq W) $$

$$ \forall A_Q : A_P \in aP \land A_P \cap A_Q : (J_P \land J_Q) \text{path} \Rightarrow \text{guard}.A_P $$

$$ \forall M : (M \subseteq W) \land (M \subseteq W) $$

$$ \forall k : k \in A : \text{path}(J_P \land J_Q \land M = k) $$

$$ (J_P \text{path} \Rightarrow q) \Rightarrow (J_P \land J_Q \text{path} \Rightarrow q) \land (J_P \text{path} \Rightarrow q) $$

**Theorem 7.10**

$$ P \subseteq_{r,J} Q \land \text{Unity}.P \land \text{Unity}.Q \land (\text{path} \land J_P \land J_Q) \land (J_P \land J_Q \Rightarrow J) $$

$$ \forall W : (wQ = wP \cup W) \land (J_P \subseteq W) \land (wP \subseteq W) $$

$$ \forall A_Q : A_P \in aP \land A_P \cap A_Q : (J_P \land J_Q) \text{path} \Rightarrow \text{guard}.A_P $$

$$ \forall A_Q : A_P \in aP \land A_P \cap A_Q : (J_P \land J_Q) \text{path} \Rightarrow \text{guard}.A_P $$

$$ (J_P \text{path} \Rightarrow q) \Rightarrow (J_P \land J_Q \text{path} \Rightarrow q) \land (J_P \text{path} \Rightarrow q) $$

**Figure 7:** Preservation of $\Rightarrow$ and $\Rightarrow$ properties.
• for all actions \( A_P \) of program \( P \) and those actions \( A_Q \) of \( Q \) that are related to \( A_P \) by \( R \), there exists a function \( M \) that is non-increasing with respect to some well-founded relation \( \prec \), such that: if the guard of \( A_Q \) is true and \( M \) equals some value \( k \) at any point during the execution of \( Q \), then either:
  • the guard of \( A_P \) always holds, the value of \( M \) always remains \( k \), and the guard of \( A_Q \) continues to hold forever, so both actions can make the same progress;
  • eventually \( M \) decreases or the guard of \( A_P \) becomes false, but at least until this happens, \( M \) remains \( k \) and the guard of \( A_Q \) continues to hold.
Consequently, if the guard of \( A_Q \) is prematurely falsified while the guard of \( A_P \) still holds, then we know that the value of \( M \) has decreased. By the previous verification condition we know that eventually the guard of \( A_Q \) will become true again, and hence given a chance to execute. Again, the guard of \( A_Q \) can be prematurely falsified, and we have the same process all over again. However, the well-foundedness of \( \prec \) guarantees that \( M \) cannot decrease infinitely, and hence that premature falsification of the guard of \( A_Q \) cannot happen infinitely.

Theorem 7.8 states a corollary of theorem 7.7. It can be proved by taking \( M \) to be a constant function. Theorem 7.9 and 7.10 state corollaries of 7.7 and 7.8 respectively. These can be proved by using the theorem stated below.

**Theorem 7.13**

\[
(\exists A :: \text{bitotal}. R. \mathsf{aP.A}) \\
\forall A_P. A_Q : A_P \in \mathsf{aP} \land A_P \mathrel{R} A_Q : J \Downarrow \text{guard}_P. A_P \rightarrow \text{guard}_P. A_Q \\
\forall A_P : A_P \in \mathsf{aP} : J \Downarrow \text{guard}_P. A_P \rightarrow (\exists A_Q :: (A_P \mathrel{R} A_Q) \land \text{guard}_P. A_Q)
\]

Note that the Theorems in Figure 7 state property preservation in refinements independently from the specific program transformations that were applied.

### 7.4 Guard strengthening and superposition refinement

Strengthening the guard of, or augmenting an assignment on an action \( A \) are action refinements of \( A \).

**Theorem 7.14**

\( \text{augment}\_A.\text{ref} \)

For all \( A, As \in \text{ACTION}, \text{state-predicates} \ J, \text{and} \ V \) a set of variables:

\[
\text{is assign}. As \land V \leftrightarrow As \land \text{WF action}. A \land \text{WF action}. As \\
A \subseteq V, J \text{ augment}. A. As \\
\]

**Theorem 7.15**

\( \text{strengthen guard}.\_A.\text{ref} \)

For all \( A \in \text{ACTION}, \text{state-predicates} \ g \) and \( J \), and \( V \) a set of variables:

\[
A \subseteq V, J \text{ strengthen guard}. g. A \\
\]

Consequently, restricted union superposition and augmentation superposition on a program \( P \) are program refinements of \( P \).

**Theorem 7.16**

\( \text{RU Superpose}.\_A.\text{ref} \)

For all programs \( P, A \in \text{ACTION}, \text{state-predicates} \ J \) and \( i.A \):

\[
w_P \leftarrow A \\
\overline{P \subseteq_{=, J} \text{RU_S.P.A}. i.A}
\]

**Theorem 7.17**

\( \text{AUG Superpose}.\_A.\text{ref} \)

For all programs \( P, As \in \text{ACTION}, \text{state-predicate} \ i.A, \text{and} \ ACs \subseteq \text{ACTION}:

\[
w_P \leftarrow A \land \text{is assign}. As \land \text{WF action}. As \\
\overline{\exists R :: P \subseteq_{R, J} \text{AUG_S.P.ACs}. As}. i.A
\]

The witness used to prove this theorem is\(^4\): \( \mathcal{R} = \mathcal{R} \mathcal{E} \mathcal{R} (\lambda A. (A \in ACs) \rightarrow \text{augment}. A. As \mid A) \).

\(^4\)Where the function \( \mathcal{R} \mathcal{E} f = (\lambda x, y. y = f.x) \), i.e. converts a function to a relation.

17
7.5 Non-determinism reducing refinement

Our definition of refinement in the previous section incorporates multiple compositions of guard strengthening and superposition program transformations, without having to specify these individual transformations explicitly. The requirement that

\[ \forall A_P \, A_Q : A_P \in \text{aP} \land A_P \not\in \mathcal{R} \, A_Q : A_P \sqsubseteq_{\text{w}, P, J} A_Q \]

takes care of (possibly multiple compositions of) guard strengthening and augmentation superpositions. The requirement

\[ \forall A_Q : A_Q \in \text{aQ}_2 : \text{skip} \sqsubseteq_{\text{w}, P, J} A_Q \]

takes care of (possibly multiple compositions of) restricted union superpositions. As a consequence, non-determinism reducing refinements like the one presented in Section 7.1, can be handled by our definition of refinement. Consider again programs \( P \) and \( Q \) from Figures 1 and 2 respectively. By taking \( \mathcal{R} = \{ (P_i, Q_i) \mid i \in \{x, y\} \} \), we can prove that for any \( J, P \sqsubseteq_{\mathcal{R}, J} Q \) holds. The proof of this is displayed below to give the interested reader an idea of the concepts involved; it may however be skipped.

**proof of:** \( P \sqsubseteq_{\mathcal{R}, J} Q \)

= (rewriting with Definition 7.4)

\[ \exists \text{aQ}_1, \text{aQ}_2 : \text{aQ} = \text{aQ}_1 \cup \text{aQ}_2 \land \text{bitalTot} \mathcal{R}. \text{aP}. \text{aQ}_1 \]

\[ \land \forall A_P \, A_Q : A_P \in \text{aP} \land A_P \not\in \mathcal{R} \, A_Q : A_P \sqsubseteq_{\text{w}, P, J} A_Q \]

\[ \land \forall A_Q : A_Q \in \text{aQ}_2 : \text{skip} \sqsubseteq_{\text{w}, P, J} A_Q \]

\[ \iff (\text{Reduce goal using witnesses aQ and } \emptyset \text{ respectively}) \]

\[ \text{aQ} = \text{aQ} \cup \emptyset \land \text{bitalTot} \mathcal{R}. \text{aP}. \text{aQ} \land (\forall A_P \, A_Q : A_P \in \text{aP} \land A_P \not\in \mathcal{R} \, A_Q : A_P \sqsubseteq_{\text{w}, P, J} A_Q) \land (\forall A_Q : A_Q \in \emptyset : \text{skip} \sqsubseteq_{\text{w}, P, J} A_Q) \]

\[ \iff (\mathcal{R} \text{ is a bitalTot properties of } \cup, \in, \text{ and } \emptyset) \]

\[ \forall A_P \, A_Q : A_P \in \text{aP} \land A_P \not\in \mathcal{R} \, A_Q : A_P \sqsubseteq_{\text{w}, P, J} A_Q \]

= (actions of programs \( P \) and \( Q \), definition of \( \mathcal{R} \))

\[ P_\mathcal{R} \sqsubseteq_{\text{w}, P, J} Q_\mathcal{R} \land P_\mathcal{R} \sqsubseteq_{\text{w}, P, J} Q_\mathcal{R} \]

= (We shall prove the one for \( P_\mathcal{R} \) the other is similar; Rewrite with Definition 7.1)

\[ \forall s : \text{guard} \mathcal{R}. Q, s \land J, s \Rightarrow \text{guard} \mathcal{R}. P, s \land J, s \]

\[ \land \forall s, t, t' : (\text{compile} P, s \land \text{compile} Q, s \land J, s) \Rightarrow t =_{\text{w}, P, J} t' \]

= (\text{guard} \mathcal{R}. P, s = (s, x \leq 10 \land s \cdot x \text{ turn}) \land \text{guard} \mathcal{R}. Q, s \land J, s \Rightarrow t =_{\text{w}, P, J} t' \]

Discharge the antecedents of this goal into the assumptions after rewriting with \( P_\mathcal{R} \) and \( Q_\mathcal{R} \).

\[ A_1 : \text{if } s \cdot x \leq 10 \text{ then } t \cdot x := s \cdot x + 1 \]

\[ A_2 : \text{if } s \cdot x \leq 10 \land s \cdot x \text{ turn} \text{ then } t \cdot x, t \cdot x \text{ turn} := s \cdot x + 1, \text{false} \]

\[ A_3 : s \cdot x \leq 10 \land s \cdot x \text{ turn} \land J, s \]

From these assumptions it is easy to deduce that \( t =_{\text{w}, P, J} t' \) which equals \( t =_{\text{w}, P, J} t' \).

**end of proof**

Proving that the property \( \text{true} \vdash x = 0 \land y = 0 \Rightarrow x = 10 \land y = 10 \) of program \( P \) is indeed preserved by its non-determinism reducing refinement \( Q \) can be established using Theorem 7.9. We already have that:

\[ A_1 : \text{true} \vdash x = 0 \land y = 0 \Rightarrow x = 10 \land y = 10 \]

\[ A_2 : \mathcal{R} = \{ (P_i, Q_i) \mid i \in \{x, y\} \} \]

\[ A_3 : J = (\neg x \cdot \text{ turn} \Rightarrow (y = x - 1)) \lor (x \cdot \text{ turn} \Rightarrow (x = y)) \]

\[ A_3 : \text{true} \vdash \square J \]

\[ A_3 : P \sqsubseteq_{\mathcal{R}, J} Q \]

Now Theorem 7.9, using witnesses \( W = \{x \cdot \text{ turn}\} \) and \( M = 20 - x - y \), and taking \( < \) to be \( < \) on numbers, leaves us with the following proof obligations:

\[ \bullet \vdash (J \land M = k) \land \text{unless } (M > k) \]

\[ \vdash (J \land y < 10 \land \neg (x \cdot \text{ turn}) \land M = k) \land \text{unless } (y < 10) \lor M < k \]

\[ \vdash (J \land x < 10 \land x \cdot \text{ turn} \land M = k) \land \text{unless } (x < 10) \lor M < k \]

\[ J \vdash x < 10 \Rightarrow x < 10 \land x \cdot \text{ turn} \]
- \( J, \vdash y < 10 \Rightarrow y < 10 \land \neg \varphi.\text{turn} \)

Proving these obligations is not hard, and left to the reader. This is a small example, and the proof-effort is not significantly reduced when we compare the proof obligations in the bullets above with the ones in Figure 4. However, we found that this example gives a good insight into the concepts that are involved when using non-determinism reducing refinements.

### 7.6 Atomicity refinement

Since our definition of refinements is based on a bitotal relation \( R \) which can relate one action in the original program to several actions in its refinement, our definition of refinement allows for some kind of atomicity relation. In the rest of this section we shall present how a simple guard simplification (taken from [Sin93]), that results in a finer grain of atomicity, can be handled within our framework of refinement.

Consider the two programs in Figure 8, where \( S \) is a finite set, and \( i \) does not occur free in \( A \). Evidently, programs \( P \) and \( Q \) keep executing action \( A \) until no element in \( S \) satisfies predicate \( g \). Let \( p = \text{if } (\exists i : i \in S : g(i)) \text{ then } A \) and \( q.i = \text{if } g(i) \text{ then } A \). It easy to prove that the relation \( R = \{ (p, q.i) \mid i \in S \} \) is bitotal on \( aP \) and \( aQ \), and consequently that for any \( J, P \subseteq R, J, Q \) to determine the conditions that need to be satisfied in order to conclude property preservation, Theorem 7.8.16 can be used to conclude:

\[
\forall i : i \in S : g(i) \subseteq wQ \quad \forall i : i \in S : q(i) \quad (J_P \vdash J_Q \land g(i)) \quad \text{unless } (\exists i : i \in S : g(i))
\]

\[
((J_P, p \Rightarrow q) \Rightarrow (J_P \land J_Q, q \Rightarrow p \Rightarrow q)) \land ((J_P, p \Rightarrow q) \Rightarrow (J_P \land J_Q, q \Rightarrow p \Rightarrow q))
\]

for the programs \( P \) and \( Q \) as displayed in Figure 8. These conditions coincide with the ones required in [Sin93].

### 8 Application of the theory

In this section we will show how the theory can be applied to prove the correctness of some relative complex distributed algorithms taken from [Vos00]. Before we start the proofs we will first explain the algorithms.

#### 8.1 The communication network

The communication networks are assumed to be connected centralised networks employing bi-directional asynchronous communication.

The networks are modelled by a triple \((P, \text{starter}, \text{neighs})\), where \( P \) is a finite set of processes; \text{starter} is a process in \( P \) that distinguishes itself from all other processes (called the \text{followers}), in that it can spontaneously start the execution of its local algorithm (e.g. because it is triggered by some internal event). The \text{followers} can only start execution of their local algorithm after they have received a first message from some neighbour; \text{neighs} is a function that given some process \( p \in P \), gives the set of neighbours of \( p \). In other words, for \( p \in P \), \text{neighs} \( p \) is the set of processes that are connected to \( p \) by a bi-directional communication link. Obviously, the function \text{neighs} should satisfy: \( \forall p \in P : \text{neighs} \ p \subseteq P \). We will only consider communication between distinct processes and not allow self-loops, thus \text{neighs} must also satisfy: \( \forall p \in P, q \in \text{neighs} \ p : p \neq q \). Since communication links are bi-directional it holds that: \( \forall p, q \in P : (q \in \text{neighs} \ p) = (p \in \text{neighs} \ q) \).

Such a network is connected if every pair of processes is connected by a path of communication links.

19
\[ \begin{align*}
\text{init} & \quad (\forall p \in P : (p = \text{starter}) \neq (\text{idle}. p)) \land (\text{father}. \text{starter} = \text{starter}) \land \text{init}_1 \\
\text{assign} & \\
\quad \text{if } \text{idle}. p \land \text{mit}. q.p & \\
\quad \quad \text{then receive } p.q.\langle \text{mes} \rangle \parallel \text{father}. p := q \parallel \text{idle}. p := \text{false} \quad \text{(idle)} \\
\quad \text{if } \lnot \text{idle}. p \land \text{mit}. q.p & \\
\quad \quad \text{then receive } p.q.\langle \text{mes} \rangle \quad \text{(col)} \\
\quad \text{if } \lnot \text{idle}. p \land \text{can}_\text{propagation}. p.q \land \text{propagating}_1.p & \\
\quad \quad \text{then send } p.q.\langle \text{mes} \rangle \quad \text{(prop)} \\
\quad \text{if } \text{finished}_\text{collecting} \text{and}_\text{propagating}. p & \land \lnot \text{reported}_\text{to}_\text{father}. p \\
\quad \quad \text{then send } p.(\text{father}. p).\langle \text{mes} \rangle \quad \text{(done)}
\end{align*} \]

Figure 9: The the local algorithm of process \( p \in P \) for \( \Pi \in \{\text{plum, echo}\} \).

For this paper it is sufficient to give an abstract model of \textit{asynchronous communication}, stating the functionality of the primitives (send and receive) and some additional operations (mit, nr\_sent\_to and nr\_rec\_from). send\( p.q.m \), implements that a process \( p \) sends message \( m \) to \( q \); receive\( p.q.f.v \), makes sure that if there is a message in transit from \( q \) to \( p \), process \( p \) receives a message from \( q \), and the value of the received message is assigned to variable \( v \) after function \( f \) has been applied to it; mit\( p.q \), the name of which is an acronym for message in transit, can be used to check for a message in transit from \( p \) to \( q \); \( p \text{ nr}_\text{sent}_\text{to} q \), enables processes to check how many messages they have already sent to a neighbour \( q \); similarly, \( p \text{ nr}_\text{rec}_\text{from} q \), to check the amount of messages received from \( q \).

### 8.2 Distributed Hylomorphisms

The class of distributed hylomorphisms from [Vos00] consists of 4 algorithms: PLUM, ECHO, TARRY and DFS. They are displayed in Figures 9 until 11 respectively. All four algorithms build a rooted spanning tree (using the father variable) in the connected network of processes and use this tree to let the required information (e.g. the values of which the sum has to be computed, or the feedback of the information that has to be propagated through the network) flow from the leaves to the root of the spanning tree. The similarities of the algorithms are captured by the characterisation of the following predicates:

\[ \begin{align*}
\text{rec}_\text{from}_\text{all}_\text{neighs} \ p & = \forall q \in \text{neighs}. p : p \text{ nr}_\text{rec}_\text{from} q = 1 \\
\text{sent}_\text{to}_\text{all}_\text{non}_\text{fathers} \ p & = \forall q \in \text{neighs}. p : (q \neq \text{father}. p) \Rightarrow (p \text{ nr}_\text{sent}_\text{to} q = 1) \\
\text{can}_\text{propagate} \ p.q & = (p \text{ nr}_\text{sent}_\text{to} q = 0) \land (q \neq \text{father}. p) \\
\text{finished}_\text{collecting}_\text{and}_\text{propagating} \ p & = \text{rec}_\text{from}_\text{all}_\text{neighs} \ p \land \text{sent}_\text{to}_\text{all}_\text{non}_\text{fathers} \ p \\
\text{reported}_\text{to}_\text{father} \ p & = (p \text{ nr}_\text{sent}_\text{to} (\text{father}. p) = 1) \\
\text{sent}_\text{to}_\text{all}_\text{neighs} \ p & = \forall q \in \text{neighs}. p : p \text{ nr}_\text{sent}_\text{to} q = 1
\end{align*} \]
prog TARRY

\textbf{init} \quad (\forall p \in \mathbb{P} : (p = \text{starter}) \neq (\text{idle}. p)) \land (\text{father}. \text{starter} = \text{starter}) \\
\land \quad (\forall p \in \mathbb{P} : (p = \text{starter}) \neq (\neg \text{le}_\text{rec}. p))

assign

\begin{align*}
\begin{array}{ll}
\lfloor q \in \text{neighs}. p & \text{if } \text{idle}. p \land \text{mit}. q.p \\
& \text{then } \text{receive}. p.q.(\text{mes}) \parallel \text{father}. p := q \parallel \text{idle}. p := \text{false} \\
& \quad \quad \quad \parallel \text{le}_\text{rec}. p := \text{true} \\
& \quad \quad \quad \parallel \text{end}
\end{array}
\end{align*}

\begin{align*}
\begin{array}{ll}
\lfloor q \in \text{neighs}. p & \text{if } \neg \text{idle}. p \land \text{mit}. q.p \land \begin{array}{l}
\text{collecting}_\text{TARRY}. p
\end{array} \\
& \text{then } \text{receive}. p.q.(\text{mes}) \parallel \text{le}_\text{rec}. p := \text{true} \\
& \quad \quad \quad \parallel \text{end}
\end{array}
\end{align*}

\begin{align*}
\begin{array}{ll}
\lfloor q \in \text{neighs}. p & \text{if } \neg \text{idle}. p \land \text{can}_\text{propagate}. p.q \land \begin{array}{l}
\text{propagating}_\text{TARRY}. p
\end{array} \\
& \text{then } \text{send}. p.q.(\text{mes}) \parallel \text{le}_\text{rec}. p := \text{false} \\
& \quad \quad \quad \parallel \text{end}
\end{array}
\end{align*}

\begin{align*}
\begin{array}{ll}
\text{end}
\end{array}
\end{align*}

\begin{align*}
\begin{array}{ll}
\text{if } \text{finished}_\text{collecting} \land \text{propagating}_\text{TARRY}. p \land \neg \text{reported}_\text{to}_\text{father}. p \\
& \text{then } \text{send}. p.(\text{father}. p).(\text{mes}) \parallel \text{le}_\text{rec}. p := \text{false}
\end{array}
\end{align*}

Figure 10: The local algorithm of process $p \in \mathbb{P}$ of the TARRY algorithm.

\begin{align*}
done.p = \text{rec}_\text{from}_\text{all}_\text{neighs}. p \land \text{sent}_\text{to}_\text{all}_\text{neighs}. p 
\end{align*}

The differences between the algorithms are in the communication protocols, i.e. when they are allowed to collect messages and propagate them.

The PLUM algorithm allows a process to freely merge its propagating and collecting actions as long as it has not yet received messages from all its neighbours, and it has not yet sent to all its neighbours that are not its father. Consequently:

\begin{align*}
\text{propagating}^{\text{PLUM}}. p = \neg \text{sent}_\text{to}_\text{all}_\text{non}_\text{fathers}. p \\
\text{collecting}^{\text{PLUM}}. p = \neg \text{rec}_\text{from}_\text{all}_\text{neighs}. p
\end{align*}

In the ECHO algorithm, a non-idle process $p$ can only receive a message, after $p$ has sent messages to all its non-father-neighbours. So, the propagating activities must be completed before starting collecting from non-father-neighbours. Consequently:

\begin{align*}
\text{propagating}^{\text{ECHO}}. p = \neg \text{sent}_\text{to}_\text{all}_\text{non}_\text{fathers}. p \\
\text{collecting}^{\text{ECHO}}. p = \neg \text{rec}_\text{from}_\text{all}_\text{neighs}. p \land \neg \text{propagating}^{\text{ECHO}}. p
\end{align*}

In the TARRY algorithm, a non-idle process $p$ can only propagate to a neighbour if the last event of $p$ was a receive event; otherwise it has to wait until it receives something. So, the propagating and collecting activities alternate. From Figure 10 we can see that a boolean-typed variable $\text{le}_\text{rec}. p$ (i.e. last event was a receive) has been introduced for every process $p$. The assignments ($\text{le}_\text{rec}. p := \text{true}$) and ($\text{le}_\text{rec}. p := \text{false}$) in the then clauses of (COL) and (PROP) respectively, guarantee that the the value of $\text{le}_\text{rec}. p$
\textbf{prog DFS}

\textbf{init} \quad \forall p \in P : (p = \text{starter}) \neq (idle.p) \land (\text{father.starter} = \text{starter})
\land \forall p \in P : (p = \text{starter}) \neq (\neg \text{le_rec.p})

\textbf{assign}

\[
\begin{align*}
\quad & [q \in \text{neighs.p} \text{ if } \text{idle.p} \land \text{mit.q.p}] \\
& \quad \text{then receive.} p.q.(\text{mes}) \land (\text{father.p} = q) \land \text{idle.p} = \text{false} \\
& \quad \quad \quad \text{if } le\text{.rec.p} = \text{true} \land (lp\text{.rec.p} = q)
\end{align*}
\]

\[
\begin{align*}
\quad & [q \in \text{neighs.p} \text{ if } \neg \text{idle.p} \land \text{mit.q.p} \land \text{collectingDFS.p}] \\
& \quad \text{then receive.} p.q.(\text{mes}) \land \text{le.rec.p} = \text{true} \land (lp\text{.rec.p} = q)
\end{align*}
\]

\[
\begin{align*}
\quad & [q \in \text{neighs.p} \text{ if } \neg \text{idle.p} \land \text{can_propagate.p.q} \land \text{propagatingDFS.p} \land (q = lp\text{.rec.p})] \\
& \quad \text{then send.} p.q.(\text{mes}) \land \text{le.rec.p} = \text{false}
\end{align*}
\]

\[
\begin{align*}
\quad & [q \in \text{neighs.p} \text{ if } \neg \text{idle.p} \land \text{can_propagate.p.q} \land \text{propagatingDFS.p} \land (\neg (\text{can_propagate.p.(lp\text{.rec.p}))})] \\
& \quad \text{then send.} p.q.(\text{mes}) \land \text{le.rec.p} = \text{false}
\end{align*}
\]

\textbf{if} \quad \text{finished_collecting and propagating.p.q} \land \text{report_to_father.p}

\textbf{then} \quad \text{send.} p.(\text{father.p}).(\text{mes}) \land \text{le.rec.p} = \text{false}

\textbf{Figure 11: The local algorithm of process } p \text{ of the DFS algorithm.}

indicates whether the last event of } p \text{ was a receive event. Consequently, we characterise the } \text{collecting and propagating } \text{predicates as follows:}

\[
\text{propagating}_{\text{TARRY}}.p = \neg \text{sent_to_all_non_fathers.p} \land (\text{le.rec.p}) \quad (12)
\]

\[
\text{collecting}_{\text{TARRY}}.p = \neg \text{rec_from_all_neighbours.p} \land (\neg (\text{le.rec.p}) \quad (13)
\]

The characterisation of the \text{propagating} and \text{collecting} predicates for the DFS algorithm are identical to those of TARRY. The difference with TARRY is in the lesser freedom to choose a neighbour to send a message to in the propagating phase (see Figure 11). More specifically, for a non-idle process } p \text{ in its propagating phase (i.e. there are still non-father-neighbours to which } p \text{ has not yet sent) whose last event was receiving a message from some neighbour } q: \text{ if } p \text{ can propagate a message back to } q, \text{ i.e. } q \text{ is not } p \text{'s father, and } p \text{ has not yet sent to } q, \text{ then } p \text{ has to send a message back to this process } q, \text{ otherwise it can act like in TARRY, and just pick any non-father-neighbour to which it has not yet sent a message (i.e. to which it can propagate). In order to be able to formalise and check these conditions each process in the DFS algorithm, remembers the identity of the sender of its last incoming message in the variable } lp\text{.rec.p} \text{ (last process of which } p \text{ has received a message).}

\[
\text{propagating}_{\text{DFS}}.p = \text{propagating}_{\text{TARRY}}.p \quad (14)
\]

\[
\text{collecting}_{\text{DFS}}.p = \text{collecting}_{\text{TARRY}}.p \quad (15)
\]
8.3 A refinement ordering on the distributed hylomorphisms

The algorithms in Figure 9 until 11 are ordered by our refinement relation as is visualised with venn-diagrams in Figure 12(a). The bitotal relations, with respect to which the different refinements are proved, are listed in Figure 12(b). Their definitions are straightforward, in that they relate all IDLE, COL, PROP and DONE actions of the original program to the corresponding actions in the refinement. For the relation between TARRY and DFS this results in PROP$_{TARRY}.p.q$ being related to both PROP$_{LP.REC}.p.q$ and PROP$_{NOT\ LP.REC}.p.q$. Although tedious, proving the bitotality of these relations and subsequently verifying the refinement ordering depicted in Figure 12 is reasonably easy. The resulting refinement theorems are listed below.

**Theorem 8.1**

\[ \forall J : \text{PLUM} \subseteq \text{PLUM}_ECHO, J \text{ ECHO} \]

**Theorem 8.2**

\[ \forall J : \text{PLUM} \subseteq \text{PLUM}_TARRY, J \text{ TARRY} \]

**Theorem 8.3**

\[ \forall J : \text{TARRY} \subseteq \text{TARRY}_DFS, J \text{ DFS} \]

8.4 The correctness of PLUM

Since this example serves to illustrate our refinement relation we will just state the correctness of the PLUM algorithm, the whole proof, however, can be found in [VS01].

**Theorem 8.4**

\[ J_{\text{PLUM}} \text{PLUM}^+ \text{iniPLUM} \leadsto \forall p : p \in P : \text{done}.p \]

Where the invariant \( J_{\text{PLUM}} \) is defined below. The M.p.q variables model the communication channels between processes \( p \) and \( q \).
\[ J_{PLUM} = \]
\[
\forall p \in \mathcal{P}, q \in \text{neighs}. p : \neg \text{idle}. p \land q = \text{father}. p \Rightarrow \neg \text{idle}. q \]
\[
\land \forall p \in \mathcal{P}, q \in \text{neighs}. p : p \text{ nr sent to } q = 0 \lor p \text{ nr sent to } q = 1 \]
\[
\land \forall p \in \mathcal{P}, q \in \text{neighs}. p : \text{idle}. p \Rightarrow p \text{ nr rec from } q = 0 \]
\[
\land \forall p \in \mathcal{P}, q \in \text{neighs}. p : (q \text{ nr rec from } p < p \text{ nr sent to } q) = \text{mit}. p.q \]
\[
\land \text{father}. \text{starter} = \text{starter} \land \neg (\text{idle}. \text{starter}) \]
\[
\land \forall p \in \mathcal{P} : (p \neq \text{starter}) \land \neg (\text{idle}. p) \Rightarrow (\text{father}. p \in \text{neighs}. p) \]
\[
\land (\lambda s. \forall p \in \mathcal{P} : \neg s.(\text{idle}. p) \Rightarrow \exists k : \text{depth} . (s \circ \text{father}). \text{starter}. p.k) \]
\[
\land \forall p, q \in \mathcal{P} : \neg (\text{idle}. p) \land \neg \text{done}. p \land (q = \text{father}. p) \Rightarrow p \text{ nr sent to } q = 0 \]
\[
\land \forall p, q \in \mathcal{P} : q \text{ nr rec from } p \leq p \text{ nr sent to } q \]
\[
\land \forall p, q \in \mathcal{P} : M.p.q = [] \lor (\exists x : M.p.q = \{x\}) \]
\[
\land \forall p, q \in \mathcal{P} : \text{idle}. p \Rightarrow p \text{ nr sent to } q = 0 \]

8.5 Using refinements to derive the correctness of ECHO

This section shall describe how termination of the ECHO algorithm can be proved using our refinements framework and the already proved fact that:

\[ \forall J : \text{PLUM} \subseteq \mathcal{R}_{\text{PLUM} \cup \text{ECHO}}, J \text{ ECHO} \]

For ECHO, the UNITY specification reads:

**Theorem 8.6**

\[ J_{PLUM} \land J_{ECHO} \land \text{in} \text{ECHO} \Rightarrow \forall p : p \in \mathcal{P} : \text{done}. p \]

where invariant \( J_{ECHO} \) captures additional safety properties for ECHO (if any).

Using \( \circ \) Preservation (Theorem 7.1215), it is straightforward to derive that \( J_{PLUM} \) is also a stable predicate in ECHO.

**Theorem 8.7**

\[ \text{ECHO} \vdash \circ J_{PLUM} \]

For readability we introduce the notational convention that:

\[ \vdash \text{ and } \text{ECHO} \vdash \text{ now abbreviate } J_{PLUM} \land J_{ECHO} \land \text{ECHO} \]

Termination of ECHO will be proved using the property preserving Theorem 7.1016.

\[ \text{ECHO} \vdash \text{in} \text{ECHO} \Rightarrow \forall p : p \in \mathcal{P} : \text{done}. p \]

\[ \Leftarrow (\text{Theorem 7.1016, 8.423, 8.123}) \]

\[ \exists W : (w \text{ ECHO} = w \text{ PLUM} \cup W) \land (J_{PLUM} \text{ C} W^c) \land (w \text{ PLUM} \subseteq W^c) \]
\[
\land \forall A.P \ A.E : A.P \in a\text{PLUM} \land A.P \text{ R}_{\text{PLUM} \cup \text{ECHO}} A.E : \text{ECHO} \vdash \text{guard of } A.P \Rightarrow \text{guard of } A.E \]
\[
\land \forall A.P \ A.E : A.P \in a\text{PLUM} \land A.P \text{ R}_{\text{PLUM} \cup \text{ECHO}} A.E : \text{ECHO} \vdash (J_{PLUM} \land J_{ECHO} \land \text{guard of } A.E) \text{ unless } \neg (\text{guard of } A.E) \]

Since no variables are superimposed on PLUM in order to construct ECHO, the first conjunct can be proved by instantiation with \( \emptyset \). Subsequently, using:

- the characterisation of \( \text{R}_{\text{PLUM} \cup \text{ECHO}} \) (Figure 12)
- the fact that the guards of the IDLE.\text{ECHO}, PROP.\text{ECHO}, and DONE.\text{ECHO} actions are equal to those of \text{PLUM}

24
• anti-reflexivity of unless (Theorem 3.17₅)
• reflexivity of ⇒ (Theorem A.4₃₁)
• the implicit assumption stating stability of \((J_{\text{PLUM}} \land J_{\text{ECHO}})\)

the second and the third conjunct can, for arbitrary \(p \in \mathbb{P}\) and \(q \in \text{neighs}_p\), be reduced to:

\[
\begin{align*}
\text{ECHO} \vdash & \quad \text{guard} \text{-} \text{of} \cdot \text{COL} \text{PLUM} \cdot p \cdot q \Rightarrow \text{guard} \text{-} \text{of} \cdot \text{COL} \text{ECHO} \cdot p \cdot q \\
\text{ECHO} \vdash & \quad J_{\text{PLUM}} \land J_{\text{ECHO}} \land \text{guard} \text{-} \text{of} \cdot \text{COL} \text{ECHO} \cdot p \cdot q \text{ unless } \neg \text{guard} \text{-} \text{of} \cdot \text{COL} \cdot p \cdot q \\
& \quad \} \text{ reach – part }
\end{align*}
\]

The \textit{unless} – part is not hard to verify, in order to prove it, the current conjuncts from \(J_{\text{PLUM}}\) suffice, and hence no additional safety properties have to be added to \(J_{\text{ECHO}}\).

Rewriting \textit{reach} – part the with the guards of the \textit{COL} actions from \textit{PLUM} and \textit{ECHO}, the correctness of the \textit{ECHO} algorithm comes down to proving that for an appropriate \(J_{\text{ECHO}}\) and arbitrary \(p \in \mathbb{P}\) and \(q \in \text{neighs}_p\):

\[
\begin{align*}
\text{ECHO} \vdash & \quad \neg \text{idle} \cdot p \land \text{mit} \cdot q \cdot p \land \neg \text{rec} \text{-} \text{from} \text{all} \text{neighs}_p \\
\text{ECHO} \vdash & \quad \neg \text{idle} \cdot p \land \text{mit} \cdot q \cdot p \land \neg \text{rec} \text{-} \text{from} \text{all} \text{neighs}_p \land \text{sent} \text{to} \text{all} \text{non} \text{fathers}_p \\
& \quad \} \text{ except – part }
\end{align*}
\]

The proof of this \textit{reach} – part can be found in [VS01], where it turns out that, again, \(J_{\text{PLUM}}\) is enough and hence \(J_{\text{ECHO}}\) can be substituted for \textit{true} – meaning that the safety properties of \textit{PLUM} and \textit{ECHO} are the same. Although the proof of the \textit{reach} – part is not trivial, it is considerably less complicated and laborious than proving 8.6 from scratch without using our refinement framework.

### 8.6 Using refinements to prove the correctness of \textit{TARRY}

This section shall describe how termination of the \textit{TARRY} algorithm is proved using our refinements framework, and the already proven fact that:

\[
\forall J : \text{PLUM} \sqsubseteq J \cdot \text{TARRY}, J \cdot \text{TARRY}
\]

The UNITY specification reads:

**Theorem 8.8**

\[
J_{\text{PLUM}} \land J_{\text{TARRY}} \Downarrow \text{TARRY} \vdash \text{init} \Downarrow \forall p : p \in \mathbb{P} : \text{done} \cdot p
\]

where invariant \(J_{\text{TARRY}}\) captures additional safety properties for \textit{TARRY}.

Using \(\circ\) Preservation (Theorem 7.12₅₅), it is straightforward to derive that \(J_{\text{PLUM}}\) is a stable predicate in \textit{TARRY}.

**Theorem 8.9**

\[
\text{TARRY} \vdash \circ J_{\text{PLUM}}
\]

For readability we introduce the notational convention that:

\[
\Downarrow \quad \text{and} \quad \Downarrow_{\text{TARRY}} \Downarrow \text{now abbreviate } J_{\text{PLUM}} \land J_{\text{TARRY}} \Downarrow
\]

Termination of \textit{TARRY} is proved using property preserving Theorem 7.9₁₆. The reason for using this theorem is that Theorem 7.10₁₆ – which is easier and hence preferable – cannot be used since its application results in the following, not provable, proof obligation:

\[
\text{TARRY} \Downarrow J_{\text{PLUM}} \land J_{\text{TARRY}} \land \text{guard} \cdot (\text{PROP} \cdot p \cdot q) \text{ unless } \neg \text{guard} \cdot (\text{PROP} \cdot p \cdot q)
\]

The reason why this cannot be proved is because, during the execution of \textit{TARRY}, it is possible that the guard of \(\text{PROP} \cdot p \cdot q\) is falsified while the guard of \(\text{PROP} \cdot p \cdot q\) still holds. Consequently, we cannot prove the \textit{unless} – property from above. What we need is a function which is non-increasing with respect to some well-founded relation, and which decreases when a message is sent. Since then, we can ensure
that this kind of premature falsification of the guard of \( \text{PROP}_{\text{TARRY}} \cdot p \cdot q \), while the guard of \( \text{PROP}_{\text{PLUM}} \cdot p \cdot q \) still holds, cannot happen infinitely often. So, since the least complicated property preservation theorem \((7.10)_{16}\) cannot be used to derive termination of \( \text{TARRY} \), we move on to the second least complicated one, i.e. \( 7.9_{16} \):

\[
\text{TARRY} \vdash \text{init}_{\text{TARRY}} \rightsquigarrow \forall p : p \in \mathbb{P} : \text{done}_p
\]

\((\text{Theorem } 7.9_{16}, 8.4_{23}, 8.2_{23})\) For some well-founded relation \( \prec \):

\[
\exists W :: (w_{\text{TARRY}} = w_{\text{PLUM}} \cup W) \land (J_{\text{PLUM}} \subseteq W) \land (w_{\text{PLUM}} \subseteq W)
\]

\(\land\)

\[
\forall A_P A_T : A_P \in a_{\text{PLUM}} \land A_P R_{\text{PLUM_{TARRY}}} A_T :
\]

\[
\text{TARRY} \vdash \text{guard}_A P \Rightarrow \text{guard}_A T
\]

\(\land\)

\[
\exists M :: (M \subseteq w_{\text{TARRY}})
\]

\(\land\)

\[
\forall k :: \text{TARRY} \vdash (J_{\text{PLUM}} \land J_{\text{TARRY}} \land M = k) \text{ unless } (M \prec k)
\]

\(\land\)

\[
\forall k A_P A_T : A_P \in a_{\text{PLUM}} \land A_P R_{\text{PLUM_{TARRY}}} A_T :
\]

\[
\text{TARRY} \vdash (J_{\text{PLUM}} \land J_{\text{TARRY}} \land \text{guard}_A T \land M = k)
\]

\(\text{unless } (\neg (\text{guard}_A P) \lor M \prec k)\)

Since, \( \text{le}_{\text{rec}} \cdot p \) variables are superimposed on \( \text{PLUM} \) in order to obtain \( \text{TARRY} \), the first conjunct is instantiated with the set \( \{ \text{le}_{\text{rec}} \cdot p : p \in \mathbb{P} \} \). Proving that \( J_{\text{PLUM}} \) is confined by the complement of this set is tedious but straightforward, since the variables \( \text{le}_{\text{rec}} \) do not appear in it.

Verification of the \textbf{unless-part} involves the construction of a function over the variables of \( \text{TARRY} \), that is non-increasing with respect to some well-founded relation \( \prec \). From the discussion above, we can deduce that we need a function that decreases when a message is sent. However, it turns out [V801] that the verification of the \textbf{reach-part} involves an application of \( \Rightarrow \text{Bounded Progress} \) \((A.10)_{11}\) that needs a function that decreases not only when a message is sent, but also when a message is received. Consequently, we shall continue with the construction of a function over the variables of \( \text{TARRY} \), that is non-increasing with respect to some well-founded relation \( \prec \), and that decreases when a message is sent as well as received. Obviously, this function can then be used for both purposes.

**Construction of a non-increasing function**

Constructing a non-increasing function that decreases when a message is sent, and when a message is received is not complicated. Observe the following:

- the sending of a message is always accompanied by incrementing \( \text{nr}_{\text{sent}} \cdot \text{to} \)
- similarly, receiving a message is always accompanied by incrementing \( \text{nr}_{\text{rec}} \cdot \text{from} \)
- from \( J_{\text{PLUM}} \) it follows that at most one message is sent over each directed communication link
- consequently, at most one message is received over each directed communication link
- consequently, the total amount of messages sent and received has an upper-bound, that equals twice the cardinality of the set of directed communication links

From these observations a non-increasing function is constructed as follows. First, we define the upper-bound on the total amount of messages sent and received.

**Definition 8.10**

\[
\text{MAX MAIL} = 2 \times \text{the amount of directed communication links in the network } (\mathbb{P}, \text{starter}, \text{neighs})
\]

Next, we define the total amount of messages that are sent, and respectively received, in the whole network of processes:
Definition 8.11 TOTAL NUMBER OF MESSAGES SENT IN THE NETWORK

\[
\text{TOTAL}_{\text{NR_SENT}.s} = \sum_{p \in P} \sum_{q \in \text{neigh}.p} s.(p \text{ nr_sent to } q)
\]

Definition 8.12 TOTAL NUMBER OF MESSAGES RECEIVED IN THE NETWORK

\[
\text{TOTAL}_{\text{NR_REC}.s} = \sum_{p \in P} \sum_{q \in \text{neigh}.p} s.(p \text{ nr_rec from } q)
\]

Now, we define our non-increasing function as follows:

Definition 8.13 NON-INCREASING FUNCTION OVER THE VARIABLES OF TARRY

\[
Y.s = \text{MAX_MAIL} - (\text{TOTAL}_{\text{NR_SENT}.s} + \text{TOTAL}_{\text{NR_REC}.s})
\]

The value of \(Y\) only depends on write variables of TARRY, and so it is easy to verify that:

Theorem 8.14

\[
Y \subseteq \text{wTARRY}
\]

The following lemma states that whenever a message is sent or received – because the guard of one of TARRY’s actions is enabled – the value of \(Y\) decreases.

Lemma 8.15

For all processes \(p \in P\), \(q \in \text{neigh}.p\), and actions \(A \in \{\text{IDLE}_{\text{TARRY}}, \text{COL}_{\text{TARRY}}, \text{PROP}_{\text{TARRY}}, \text{DONE}_{\text{TARRY}}\}\):

\[
\forall k :: \frac{J_{\text{PLUM}.s} \land A.p.q.s.t \land \text{guard_of}(A.p.q), s \land (Y.s = k)}{Y.t < k}
\]

Using this lemma, it is straightforward to prove that, during the execution of TARRY, \(Y\) is non-increasing with respect to the well-founded relation \(<\) on numerals.

Theorem 8.16

For arbitrary characterisations of \(J_{\text{TARRY}}\):

\[
\forall k :: \text{TARRY} \vdash (J_{\text{PLUM}} \land J_{\text{TARRY}} \land Y = k) \text{ unless } (Y < k)
\]

Verification of the unless-part

Return to page 26 for the unless-part. Instantiating this proof obligation with \(Y\), and rewriting with Theorems 8.1427 and 8.1627 results in the following proof obligation:

\[
\forall k A_P A_T : A_P \in a_{\text{PLUM}} \land A_P \text{ R}_{\text{PLUM,TARRY}} A_T : \\
\text{TARRY} \vdash (J_{\text{PLUM}} \land J_{\text{TARRY}} \land \text{guard_of}.A_T \land Y = k) \text{ unless } \neg(\text{guard_of}.A_P) \lor Y < k
\]

Proving this is straightforward using \(\text{R}_{\text{PLUM,TARRY}}\) from Figure 12, and Lemma 8.1527.

Verification of the reach-part

We shall now continue with the reach-part:

\[
\forall A_P A_T : A_P \in a_{\text{PLUM}} \land A_P \text{ R}_{\text{PLUM,TARRY}} A_T : \text{TARRY} \vdash \text{guard_of}.A_P \Rightarrow \text{guard_of}.A_T
\]

Subsequently, using:

- the characterisation of \(\text{R}_{\text{PLUM,TARRY}}\) (Figure 12)
- the fact that the guards of \(\text{IDLE}_{\text{TARRY}}\) and \(\text{DONE}_{\text{TARRY}}\) actions are equal to those of \(\text{PLUM}\)
- reflexivity of \(\Rightarrow\) (Theorem A.431)
- the implicit assumption stating stability of \((J_{\text{PLUM}} \land J_{\text{TARRY}})\)

27
we reduce the reach-part for arbitrary \( p \in \mathbb{P} \) and \( q \in \text{neighs}_p \), as follows:

\[
\begin{align*}
\text{TARRY} \vdash \text{guard}(\text{COL}_\text{PLUM}.p.q) & \Rightarrow \text{guard}(\text{COL}_\text{TARRY}.p.q) \quad \{ \text{reach - COL - part} \} \\
\text{TARRY} \vdash \text{guard}(\text{PROP}_\text{PLUM}.p.q) & \Rightarrow \text{guard}(\text{PROP}_\text{TARRY}.p.q) \quad \{ \text{reach - PROP - part} \}
\end{align*}
\]

Subsequently, rewriting with the characterisations of the guards we can reduce the verification of the TARRY's correctness to the following two proof obligations.

\[
\begin{align*}
\text{TARRY} \vdash \text{guard}(\text{COL}_\text{PLUM}.p.q) & \Rightarrow \text{guard}(\text{COL}_\text{PLUM}.p.q) \land \neg \text{rec}_p \\
\text{TARRY} \vdash \text{guard}(\text{PROP}_\text{PLUM}.p.q) & \Rightarrow \text{guard}(\text{PROP}_\text{PLUM}.p.q) \land \text{rec}_p
\end{align*}
\]

Again, their proofs can be found in [VS01]. This time \( J\text{\_PLUM} \) does not suffice, because, evidently, we need to capture the additional safety behaviour about the alternating sending and receiving activities \( \text{rec} \) in \( J\text{\_TARRY} \). Although not trivial, these proofs and the construction of \( J\text{\_TARRY} \) are considerably less complicated and laborious than proving 8.6 from scratch without using our refinement framework. More specific, the remaining efforts are a subset of all verification efforts that had to be done when proving TARRY's correctness from scratch!

### 8.7 Using refinements to prove the correctness of DFS

This section shall describe how termination of the DFS algorithm is proved using our refinements framework and the already proven fact that:

\[
\forall J :: \text{TARRY} \sqsubseteq_{R} \text{DFS}_J, J \text{ DFS}
\]

The UNITY specification reads:

**Theorem 8.17**

\[
J\text{\_PLUM} \land J\text{\_TARRY} \land J\text{\_DFS} \vdash \text{init}_\text{DFS} \leadsto \forall p : p \in \mathbb{P} : \text{done}_p
\]

where invariant \( J\text{\_DFS} \) captures additional safety properties for DFS (if any). Using \( \bowtie \text{Preservation} \) Theorem 7.12, it is straightforward to derive:

**Theorem 8.18**

\[
\text{DFS} \vdash \bowtie (J\text{\_PLUM} \land J\text{\_TARRY})
\]

Again, for readability we introduce the notational convention that:

\[
\vdash \text{ and } \text{DFS} \vdash \text{ now abbreviate } J\text{\_PLUM} \land J\text{\_TARRY} \land J\text{\_DFS} \vdash
\]

Termination of DFS is proved using property preserving Theorem 7.7. The reasons for using this theorem are twofold. First, since every \text{PROP} action in TARRY is bitotally related to two actions in DFS (namely \text{PROP\_LP\_REC} and \text{PROP\_NOT\_LP\_REC}), we need to be able to pick one of those DFS \text{PROP} actions when proving that the guards of TARRY’s \text{PROP} actions eventually implies the guards of related DFS’s \text{PROP} actions. Consequently, we cannot use preservation theorems 7.10 or 7.9. The second reason for using 7.7 is not because 7.8 cannot be used, but because it reduces proof effort. As we have seen during TARRY’s verification, Lemma 8.15 was very useful when proving \text{unless} and ensures properties that involved \( Y \). A similar lemma can easily be proved for the actions of DFS, and hence verification of \text{unless} and ensures properties involving \( Y \) in the context of DFS will be simple too.

**Lemma 8.19**

\[
\text{For all } p \in \mathbb{P}, q \in \text{neighs}_p, \text{ and actions } A \in \{ \text{IDLE}_\text{DFS}, \text{COL}_\text{DFS}, \text{PROP\_LP\_REC}, \text{PROP\_NOT\_LP\_REC}, \text{DONE}_\text{DFS} \}:
\]

\[
\forall k :: \frac{J\text{\_PLUM}.s \land A.p.q.s.t \land \text{guard}(A.p.q).s \land (Y.s = k)}{Y.t < k}
\]

28
In resumen, to reduce proof effort we have decided to use $7.16$, although a function that is non-increasing with respect to some well-founded relation is not needed in order to be able to prove that falsification of the guards of $\text{DFS}$’s PROP-actions go hand in hand with the falsification of the guards of $\text{TARRY}$’s PROP-actions.

As a result, the initial specification stating termination of $\text{DFS}$ is decomposed as follows:

$$\text{DFS} \vdash \text{initDFS} \iff \forall p \in \mathbb{P} : \text{done} p$$
$$\iff (\text{Theorem } 7.16, 8.25, 8.33)$$

For some well-founded relation $\prec$:

$$\exists W :: (w_{\text{DFS}} = w_{\text{TARRY}} \cup W) \land (w_{\text{DFS}} = w_{\text{TARRY}} \cup W) \land (w_{\text{TARRY}} \subseteq w_{\text{DFS}})$$

$$\land$$

$$\forall A_D : A_D \in a_{\text{DFS}} \land (\exists A_T : A_T \in a_{\text{TARRY}} \land (A_T R_{\text{TARRY}, \text{DFS}} A_D)) : (\text{guard}_{\text{DFS}}.A_D C w_{\text{DFS}})$$

$$\land$$

$$\forall A_T A_D : A_T \in a_{\text{TARRY}}$$

$$\text{DFS} \vdash \text{guard}_{\text{DFS}}.A_T$$

$$\implies$$

$$\left( \exists A_D :: (A_T R_{\text{TARRY}, \text{DFS}} A_D) \land \text{guard}_{\text{DFS}}.A_D \right)$$

$$\land$$

$$\exists M :: (M C w_{\text{DFS}})$$

$$\land$$

$$\forall k :: \text{DFS} \vdash (J_{\text{PLUM}} \land J_{\text{TARRY}} \land J_{\text{DFS}} \land M = k) \text{ unless } (M < k)$$

$$\land$$

$$\forall k A_T A_D : A_T \in a_{\text{TARRY}} \land (A_T R_{\text{TARRY}, \text{DFS}} A_D) :$$

$$\text{DFS} \vdash (J_{\text{PLUM}} \land J_{\text{TARRY}} \land J_{\text{DFS}} \land \text{guard}_{\text{DFS}}.A_D \land M = k)$$

$$\land$$

$$\text{unless } (\neg (\text{guard}_{\text{DFS}}.A_T) \lor M < k)$$

Since, $\text{lp_rec.p}$ variables are superimposed on $\text{TARRY}$ in order to obtain $\text{DFS}$, the first conjunct is instantiated with the set $\{\text{lp_rec.p} | p \in \mathbb{P}\}$. Proving that $J_{\text{PLUM}}$ and $J_{\text{TARRY}}$ are confined by the complement of this set is tedious but straightforward, since the variables $\text{le_rec}$ do not appear in it. Similarly, proving that the guards of the actions in $\text{DFS}$ are confined by $\text{DFS}$’s write variables (i.e. the second conjunct) is not complicated.

The **unless-part** is now easy to prove by instantiating with $Y$ (Definition 8.1327):

- proving that $Y$ is confined by the write variables of $\text{DFS}$ is easy using Theorem 8.1427 and monotonicity of $\text{confinement}$ 3.23
- proving that $Y$ is non-increasing in $\text{DFS}$, can be proved using **unless preservation** Theorem 7.1115 and Theorem 8.1627.
- proving that falsification of the guards of $\text{DFS}$’s actions go hand in hand with the falsification of the guards of related $\text{TARRY}$’s actions is easy using Lemma 8.1928.

For the **reach-part**, the IDLE, COL, and DONE cases can be proved using $\Rightarrow \text{ introduction}$ (A.351). As a consequence, we are left with the **PROP case**:

$$\text{DFS} \vdash \text{guard}_{\text{DFS}}.(\text{PROP}_{\text{TARRY}}.p.q) \iff (\exists A_D :: (\text{PROP}_{\text{TARRY}}.p.q R_{\text{TARRY}, \text{DFS}} A_D) \land \text{guard}_{\text{DFS}}.A_D)$$

This case states that: from a situation in which $\text{guard}_{\text{DFS}}.(\text{PROP}.p.q)$ holds, we will eventually reach a situation in which either the guard of action $\text{PROP}_{\text{TARRY}}.p.q$ or $\text{PROP}_{\text{NOT LPREC}}.p.q$ holds. The proof can be found in [VS01], where it turns out that, $J_{\text{TARRY}}$ is enough and hence $J_{\text{DFS}}$ can be substituted for true — meaning that the safety properties of $\text{TARRY}$ and $\text{DFS}$ are the same.

### 9 Conclusion

We have defined a refinement relation on programs that incorporates (possibly multiple compositions of) program transformations like guard strengthening, superposition, and atomicity refinement. Moreover, we have given theorems that state property preservation in refinements independently from the specific program transformations that were applied. Consequently, we have a general framework of refinements that, besides being suitable for the stepwise derivation of programs, is also efficient for the reduction of
proof-effort when proving the correctness of a class of by refinement related algorithms. To illustrate the reduction of proof-effort we refer to Figure 13. The intuition behind Figure 13 is that the use of refinements can shorten the the actual proof of a refinement (i.e. the solid line) since instead of proving the program from scratch we prove the simpler verification conditions of one of the theorems in Figure 7. Moreover, the amount of time spent on repairing and backtracking is reduced since having verified $P$'s correctness we have obtained a good feeling about the workings of the algorithms in this particular class, and hence will it be less likely that we proceed on wrong proof-strategies.

Figure 13: Reducing proof-effort and complexity.
A  Laws of $\Rightarrow$

Theorem A.1 $\Rightarrow$ Stable Background and Confinement

\[
P : \quad \frac{J \vdash p \Rightarrow q}{\circ J \land p.q \mathcal{C} wP}
\]

Theorem A.2 $\Rightarrow$ Substitution

\[
P, J : \quad \frac{p.s \mathcal{C} wP \land [J \land p \Rightarrow q] \land (q \Rightarrow r) \land J \land r \Rightarrow s}{p \Rightarrow s}
\]

Theorem A.3 $\Rightarrow$ Introduction

\[
P, J : \quad \frac{p.q \mathcal{C} wP \land (\circ J) \land ([J \land p \Rightarrow q] \lor (J \land p \text{ ensures } q))}{p \Rightarrow q}
\]

Theorem A.4 $\Rightarrow$ Reflexivity

\[
P, J : \quad \frac{p \mathcal{C} wP \land (\circ J)}{p \Rightarrow p}
\]

Theorem A.5 $\Rightarrow$ Transitivity

\[
P, J : \quad \frac{(p \Rightarrow q) \land (q \Rightarrow r)}{p \Rightarrow r}
\]

Theorem A.6 $\Rightarrow$ Case Distinction

\[
P, J : \quad \frac{(p \land \neg r \Rightarrow q) \land (p \land r \Rightarrow q)}{p \Rightarrow q}
\]

Theorem A.7 $\Rightarrow$ Cancellation

\[
P, J : \quad \frac{q \mathcal{C} wP \land (p \Rightarrow q \lor r) \land (r \Rightarrow s)}{p \Rightarrow q \lor s}
\]

Theorem A.8 $\Rightarrow$ Progress Safety Progress (PSP)

\[
P, J : \quad \frac{r.s \mathcal{C} wP \land (r \land J \text{ unless } s) \land (p \Rightarrow q)}{p \land r \Rightarrow (q \land r) \lor s}
\]

Theorem A.9 $\Rightarrow$ Disjunction

\[
P, J : \quad \frac{(\forall i : i \in W : p.i \Rightarrow q.i)}{(\exists i : i \in W : p.i) \Rightarrow (\exists i : i \in W : q.i)} \quad \text{if } W \neq \emptyset
\]

Theorem A.10 $\Rightarrow$ Bounded Progress

For a well-founded relation $\prec$ over some set $W$, and metric $M \in \text{State} \to W$:

\[
P, J : \quad \frac{q \mathcal{C} wP \land (\forall m \in W : p \land (M = m) \Rightarrow (p \land (M \prec m)) \lor q)}{p \Rightarrow q}
\]

B  Laws of $\rightsquigarrow$

Theorem B.1 Convergence Implies Progress

\[
P, J : \quad \frac{p \rightsquigarrow q}{p \Rightarrow q}
\]

Theorem B.2 $\rightsquigarrow$ Substitution

\[
P, J : \quad \frac{p.s \mathcal{C} wP \land [J \land p \Rightarrow q] \land (q \Rightarrow r) \land [J \land r \Rightarrow s]}{p \rightsquigarrow s}
\]
Theorem B.3 \[ \text{Introduction} \]
\[
\begin{align*}
P, J : & \quad p \, q \, c \, w \, p \land (\bigcup J) \land (\bigcup (J \land q)) \land (J \land p \Rightarrow q) \lor (p \land J \text{ ensures } q) \\
p & \leadsto q
\end{align*}
\]

Theorem B.4 \[ \text{Reflexivity} \]
\[
\begin{align*}
P, J : & \quad p \, c \, w \, p \land (\bigcup J) \land (\bigcup (J \land p)) \\
p & \leadsto p
\end{align*}
\]

Theorem B.5 \[ \text{Transitivity} \]
\[
\begin{align*}
P, J : & \quad (p \leadsto q) \land (q \leadsto r) \\
p & \leadsto r
\end{align*}
\]

Theorem B.6 \[ \text{Case Distinction} \]
\[
\begin{align*}
P, J : & \quad (p \land \neg r \leadsto \neg q) \land (p \land r \leadsto q) \\
p & \leadsto q
\end{align*}
\]

Theorem B.7 \[ \text{Accumulation} \]
\[
\begin{align*}
P, J : & \quad (p \leadsto q) \land (q \leadsto r) \\
p & \leadsto q \land r
\end{align*}
\]

Theorem B.8 \[ \text{Stable Strengthening} \]
\[
\begin{align*}
P : & \quad q \, c \, w \, p \land (\bigcup (J_1 \land J_2)) \land (J_1 \vdash p \leadsto q) \\
(J_1 \land J_2) & \vdash p \leadsto q
\end{align*}
\]

Theorem B.9 \[ \text{Stable Shift} \]
\[
\begin{align*}
P : & \quad p' \, c \, w \, p \land (\bigcup J) \land (J \land p' \vdash p \leadsto q) \\
J & \vdash p' \land p \leadsto q
\end{align*}
\]

Theorem B.10 \[ \text{Disjunction} \]
\[
\begin{align*}
P, J : & \quad (\forall i : i \in W : p \cdot i \leadsto q \cdot i) \\
(\exists i : i \in W : p \cdot i) & \leadsto (\exists i : i \in W : q \cdot i)
\end{align*}
\]

Theorem B.11 \[ \text{Conjunction} \]
\[
\begin{align*}
P, J : & \quad (\forall i : i \in W : p \cdot i \leadsto q \cdot i) \\
(\forall i : i \in W : p \cdot i) & \leadsto (\forall i : i \in W : q \cdot i)
\end{align*}
\]

Theorem B.12 \[ \text{Bounded Progress} \]
\[
\begin{align*}
P, J : & \quad (q \leadsto q) \land (\forall m \in A : p \land (M = m) \leadsto (p \land (M < m)) \lor q) \\
p & \leadsto q
\end{align*}
\]

Theorem B.13 \[ \text{Iteration} \]
\[
\begin{align*}
P, J, L : & \quad ((\forall x : x \in L : Q \cdot x) \land J) \land (\forall x : x \in L : Q \cdot x \, c \, w \, p) \\
L \subseteq W & \Rightarrow ((f \cdot L) \subseteq W \land (\forall x : x \in L : Q \cdot x) \leadsto (\forall x : x \in f \cdot L : Q \cdot x)) \\
\forall n L : L \subseteq W & \Rightarrow (\forall x : x \in L : Q \cdot x) \leadsto (\forall x : x \in \text{iterate}_n \, f \cdot L : Q \cdot x)
\end{align*}
\]

C \ \text{Proofs of the refinement theorems}

This appendix presents detailed proofs of the Theorems 7.11, 7.15, and 7.16 stating the conditions under which unless and \[ \leadsto \] properties are preserved under refinement. The other theorems in Section 7.3 are corollaries of these two theorems.
C.1 Preservation of unless

**Theorem 7.11**

\[
P \models_{\mathcal{R},J} Q \land \text{Unity}_P \land Q \land (\varphi \circ J_Q) \land (J_Q \Rightarrow J)
\]

\[
\exists W :: (wQ = wP \cup W) \land (p \subseteq W^c) \land (q \subseteq W^c)
\]

\[
\vdash p \text{ unless } q \Rightarrow \varphi \ (J_Q \land p) \text{ unless } q
\]

**proof of 7.11**

Assume the following:

A1: \( P \models_{\mathcal{R},J} Q \)
A2: \( \text{Unity}_P \land Q \land (\varphi \circ J_Q) \land (J_Q \Rightarrow J) \)
A3: \( wQ = wP \cup W \)
A4: \( (p \subseteq W^c) \land (q \subseteq W^c) \)
A5: \( \vdash p \text{ unless } q \Rightarrow \varphi \ (J_Q \land p) \text{ unless } q \)

From A5 and the definition of confinement (3.13) we can infer:

A7: \( \forall t, t' : (t =_{W^c} t') \Rightarrow (p.t = p.t' \land q.t = q.t') \)

From A6, the definitions of unless (3.15\(a\)) and the definition of Hoare triples (3.14\(a\)) we can infer:

A8: \( \forall \alpha P : \alpha P \in \alpha P : \forall s, t : \text{compile} \alpha P, s.t \land p, s \land \neg (q.s) \Rightarrow (p.t \lor q.t) \)

Now we have to prove the following:

\( \vdash (J_Q \land p) \text{ unless } q \)

\( = \) (Definitions of unless (3.15\(a\)) and Hoare triples (3.14\(a\)))

\( \forall \alpha Q \in \alpha Q : \forall s, t : \text{compile} \alpha Q, s.t \land J_Q, s \land p, s \land \neg (q.s) \Rightarrow ((J_Q \land p.t) \lor q.t) \)

Choose an arbitrary \( \alpha Q \), and assume for arbitrary states \( s \) and \( t \) that:

A\( 9 \): \( \alpha Q \in \alpha Q \)
A\( 10 \): \( \text{compile} \alpha Q, s.t \)
A\( 11 \): \( J_Q, s \land p, s \land \neg (q.s) \)

Now we have to prove that \( ((J_Q \land p.t) \lor q.t) \). From A\( 3 \), we know that \( \varphi \circ J_Q \), and consequently, using assumptions A\( 9 \), A\( 10 \), A\( 11 \) and the definition of \( \circ \) (3.16\(a\) and 3.15\(a\)) we can conclude that \( J_Q \land t \). Thus, we are left with the following proof obligation:

\( p.t \lor q.t \)

**Case** \( \neg (\text{guard of } \alpha Q, s) \)

In this case A\( 10 \) implies \( s = t \), and thus assumption A\( 10 \) establishes the validity of \( p.t \lor q.t \).

\( \Box (\neg (\text{guard of } \alpha Q, s)) \)

**Case** \( \text{guard of } \alpha Q, s \)
A\( 12 \): \( \text{guard of } \alpha Q, s \)

From A\( 1 \) it follows that:

A\( 13 \): \( \alpha Q = \alpha Q_1 \cup \alpha Q_2 \land \text{bitotal} \mathcal{R}. \alpha P \alpha Q_1 \)
$A_{14}$: $\forall A_p\ A_Q : A_P \in aP \land A_P \models A_Q : A_P \subseteq_{wP,J} A_Q$

$A_{15}$: $\forall A_Q : A_Q \in aQ_2 : \text{skip} \subseteq_{wP,J} A_Q$

**Case** $A_Q \in aQ_1$

$A_{16}$: $A_Q \in aQ_1$

From $A_{13}$, $A_{14}$ and $A_{16}$ we can conclude that there exists an action $A_P$, such that:

$A_{17}$: $A_P \in aP$

$A_{18}$: $A_P \models A_Q$

$A_{19}$: $A_P \subseteq_{wP,J} A_Q$

From $A_{17}$ and the always-enabledness of actions in the universe $\text{ACTION (3.34)}$ we know that there exists a state $t'$ such that

$A_{20}$: $\text{compile}.A_P \cdot s \cdot t'$

and consequently from $A_{19}$, $A_{20}$, the definition of action refinement (7.112), and $A_{10}$, $A_{11}$, $A_{12}$ and $A_3$ we can infer that:

$A_{21}$: $t =_{wP} t'$

Moreover, using $A_{2}$, $A_{10}$, $A_{20}$, and the definition of a well-formed $\text{UNITY}$ program (3.135), and the definition of ignored variables (3.54) we can conclude:

$A_{22}$: $s =_{wP} t'$

$A_{23}$: $s =_{wQ} t$

From $A_{21}$, $A_{22}$ and $A_{23}$ we can prove that $t =_{wP} t'$, which with $A_{7}$ gives:

$A_{24}$: $p \cdot t = p \cdot t' \land q \cdot t = q \cdot t'$

From assumptions $A_8$, $A_{11}$, $A_{17}$, $A_{20}$ we can conclude that $p \cdot t' \land q \cdot t'$, and thus $A_{24}$ establishes this case.

$\square A_Q \in aQ_1$

**Case** $A_Q \in aQ_2$

From $A_9$, $A_{15}$, the definition of action refinement (7.112), skip (3.44), $A_3$, $A_{10}$, $A_{11}$ and $A_{12}$ and we can conclude:

$A_{25}$: $s =_{wP} t$

Again, using $A_{2}$ and the definition of a well-formed $\text{UNITY}$ program (3.135), and the definition of ignored variables (3.54) we can conclude:

$A_{26}$: $s =_{wQ} t$

From this we can derive $s =_{wP} t$, which with $A_{11}$ gives the desired result.

$\square A_Q \in aQ_2$

$\square_{\text{guard} \cdot f} A_Q \cdot s$

**end of proof** 7.11
C.2 Preservation of $\rightsquigarrow$

**Theorem 7.7**  

Let $\prec$ be a well-founded relation over some set $A$, and $M \in \text{State}\rightarrow A$.

\[
\begin{align*}
P \subseteq_{R, J} Q & \land P \land Q \land (\text{guard}_w.A \land J \lor J) \land (J \land Q) \Rightarrow J) \\
\forall A_Q : A_Q \in aQ \land (\exists A_P : (A_P \in aP) \land (A_P \land R.A_Q)) : (\text{guard}_w.A \land wQ) \\
\forall A_P : A_P \in aP : (J \land Q) \Rightarrow (\exists A_Q : (A_P \land R.A_Q) \land \text{guard}_w.A.Q) \\
\exists M : (M \land wQ) \land (\forall k : k \in A : \text{guard}_w.A \land (J \land Q \land M = k) \Rightarrow M \land k) \\
\land \forall k : A_P.A_Q : k \in A \land A_P \in aP \land A_P \land R.A_Q : \\
\text{guard}_w.A \land (J \land Q \land \text{guard}_w.A.Q \land M = k) \Rightarrow M \land k) \\
((J \land Q \land p \Rightarrow q) \Rightarrow ((J \land Q \land q \Rightarrow p \Rightarrow q)) \land ((J \land Q \land p \Rightarrow q) \Rightarrow (J \land Q \land q \Rightarrow p \Rightarrow q)))
\end{align*}
\]

**proof of 7.7 ($\rightsquigarrow$-part)**

Assume the following for a well-founded relation $\prec$:

\[A_1: P \subseteq_{R, J} Q\]
\[A_2: \text{Unity} \land \text{Unity} \land Q\]
\[A_3: \text{guard}_w.A \land (J \land Q) \land (J \land Q) \Rightarrow J)\]
\[A_4: wQ = wP \cup W \land J \land T \land W \land wP \subseteq W\]
\[A_5: \forall A_Q : A_Q \in aQ \land (\exists A_P : (A_P \in aP) \land (A_P \land R.A_Q)) : (\text{guard}_w.A \land wQ)\]
\[A_6: \forall A_P : A_P \in aP : (J \land Q) \Rightarrow (\exists A_Q : (A_P \land R.A_Q) \land \text{guard}_w.A.Q)\]
\[A_7: M \land wQ\]
\[A_8: \forall k : k \in A : \text{guard}_w.A \land (J \land Q \land M = k) \Rightarrow M \land k)\]
\[A_9: \forall k : A_P.A_Q : k \in A \land A_P \in aP \land A_P \land R.A_Q : \\
\text{guard}_w.A \land (J \land Q \land \text{guard}_w.A.Q \land M = k) \Rightarrow M \land k)\]

We have to prove that:

\[J \land Q \land p \Rightarrow q \Rightarrow (J \land Q \land q \Rightarrow p \Rightarrow q)\]

For this we use the following theorem directly taken from [Pra95]; it states an induction principle for the $\Rightarrow$ operator that corresponds to the latter’s definition (3.19b):

**Theorem C.1 $\Rightarrow$ INDUCTION**

For transitive and disjunctive $R$:

\[
\begin{align*}
P, J : \quad (\forall p, q : p.C(wP) \land q.C(wP) \land (\exists p \land p \text{ ensures } q)) \Rightarrow R.p.q \\
(p \Rightarrow q) \Rightarrow R.p.q
\end{align*}
\]

take $R = (\lambda p, q. J \land Q \land p \Rightarrow q)$. Since we already have $\Rightarrow$-TRANSITIVITY, and $\Rightarrow$-DISJUNCTION, we are left with the following proof-obligation:

\[
\forall p, q : (p.C(wP \land q.C(wP) \land (J \land p \text{ ensures } q)) \Rightarrow (J \land Q \land q \Rightarrow p \Rightarrow q)
\]

Choose arbitrary $p$ and $q$, and assume:

\[A_{10}: p.C(wP) \land q.C(wP)\]
\[A_{11}: p \Rightarrow (J \land Q)\]
\[A_{12}: p \Rightarrow J \land p \text{ ensures } q\]

Theorem 3.2, stating confinement monotonicity, together with $A_4$ and $A_{10}$ gives:

\[A_{13}: p.C(wQ \land q.C(wQ)\]

35
Rewriting $A_{12}$ with the definition of ensures (3.185), we know that there exists an action $A_P$, such that:

$A_{14}$: $p \vdash J_P \land p$ unless $q$

$A_{15}$: $A_P \in aP$

$A_{16}$: $\forall s \ t :: (J_P \land p \land s \land \neg (q \land \text{compile} \cdot A_P \cdot s \cdot t) \Rightarrow q \land t$

Now we have to prove that:

$J_P \land J_Q \vdash p \Rightarrow q$

$\Leftarrow (\Rightarrow \text{ Bounded Progress} (A.10_{\text{iii}}), A_7, \prec \text{ is well-founded})$

$\forall k :: J_P \land J_Q \vdash p \land M=k \Rightarrow (p \land M \prec k) \lor q$

$\Leftarrow (\Rightarrow \text{ Case Distinction} (A.6_{\text{iii}}))$

$\forall k :: J_P \land J_Q \vdash p \land M=k \land \neg (\text{guard of } A_P) \lor (p \land M \prec k) \lor q$

$\forall k :: J_P \land J_Q \vdash p \land M=k \land \text{guard of } A_P \lor (p \land M \prec k) \lor q$

Before we prove these two conjuncts we first prove the following lemma.

**Lemma 1**: $\forall s :: (J_P \land p \land s \land \neg (\text{guard of } A_P \cdot s)) \Rightarrow q \land s$

Choose an arbitrary state $s$, and assume that:

$J_P \land p \land s \land \neg (\text{guard of } A_P \cdot s)$

Since the guard of $A_P$ is false, compile $A_P \cdot s \cdot t = (s = t)$, instantiating $A_{16}$ with state $s$ and rewriting with these assumptions gives us:

$\forall t : (\neg (q \land s = t) \Rightarrow q \land t$

which equals $q \land s$.

\square

**false-guard-$A_P$-part**

Theorem $\Rightarrow \text{ Introduction} (A.3_{\text{iii}}, \text{ implication-part})$, assumptions $A_3, A_7$ and $A_{13}$, and **Lemma 1** establish this case.

\square

**true-guard-$A_P$-part**

For arbitrary $k$ we have to prove that:

$J_P \land J_Q \vdash p \land M=k \land \text{guard of } A_P \Rightarrow (p \land M \prec k) \lor q$

$\Leftarrow (\Rightarrow \text{ Cancellation} (A.7_{\text{iii}}, A_7, A_{13})$

$J_P \land J_Q \vdash p \land M=k \land \text{guard of } A_P$

$\Rightarrow (p \land M \prec k) \lor q \lor (p \land M=k \land (\exists A_Q :: A_P \land A_Q \land \text{guard of } A_Q))$

$\Rightarrow (p \land M \prec k) \lor q \Rightarrow (p \land M=k \land (\exists A_Q :: A_P \land A_Q \land \text{guard of } A_Q) \Rightarrow (p \land M \prec k) \lor q$

$\Rightarrow (C_1 \lor C_2)$

36
Before we continue with the proofs of these conjunct, we shall first prove another lemma.

**Lemma 2**: \( \phi \vdash (J_P \land J_Q) \land p \) unless \( q \)

(1) \( \phi \vdash J_Q \land (J_P \land p) \) unless \( q \)

(2) \( \phi \) (preservation of unless (7.11), \( A_1, A_2, A_3, A_4, A_14 \))

(3) \( J_P \land p \) \( \Lambda C \land q \Lambda C \land \Lambda C \)

(4) \( \phi \) (confinement of binary operators on first conjunct, and \( A_4 \))

(5) \( p \Lambda C \land q \Lambda C \land \Lambda C \)

(6) \( \phi \) (confinement monotonicity (3.12) on both conjuncts)

Assumption \( A_{10} \) and \( A_4 \) establishes this case.

\[ \square \text{lemma2} \]

**Proof of \( C_1 \)**

\[ J_P \land J_Q \phi \vdash p \land \Lambda M=k \land \text{guard of } A_P \]

\[ \rightarrow (p \land \Lambda M<k) \lor q \lor (p \land M=k \land (\exists A_Q :: (A_P \land A_Q) \land \text{guard of } A_Q)) \]

\[ \phi \vdash \phi \] (PSP (A.831), \( A_{13} \), lemma 2)

\[ J_P \land J_Q \phi \vdash M=k \land \text{guard of } A_P \rightarrow (\exists A_Q :: (A_P \land A_Q) \land \text{guard of } A_Q) \]

\[ \phi \vdash \phi \] (PSP (A.831), \( A_7 \))

Assumptions \( A_6, A_{15} \) and \( A_8 \) establish this.

\[ \square \text{C}_1 \]

**Proof of \( C_2 \)**

\[ J_P \land J_Q \phi \vdash p \land \Lambda M=k \land (\exists A_Q :: (A_P \land A_Q) \land \text{guard of } A_Q) \rightarrow (p \land \Lambda M<k) \lor q \] \( \square \text{C}_2 \)

\[ \phi \vdash \phi \] (Substitution (A.231), \( A_5, A_7, \) and \( A_{13} \))

\[ J_P \land J_Q \phi \vdash \exists A_Q :: (A_P \land A_Q) \land p \land \Lambda M=k \land \text{guard of } A_Q \rightarrow \exists A_Q :: (A_P \land A_Q) \land ((p \land \Lambda M<k) \lor q) \]

\[ \phi \vdash \phi \] (Disjunction (A.931), \( A_5, A_7, \) and \( A_{13} \))

Assume:

\( A_{18} : A_P \land A_Q \)

We are left with the proof obligations (definition of ensures (3.185))

\[ \phi \vdash J_P \land J_Q \land p \land \Lambda M=k \land \text{guard of } A_Q \) unless \( (p \land \Lambda M<k) \lor q \) \] \( \square \text{unless-part} \)

\[ \exists A_Q : A_Q \in \text{a}Q : \{ J_P \land J_Q \land p \land \Lambda M=k \land \text{guard of } A_Q \land \neg((p \land \Lambda M<k) \lor q) \} \land \{ (p \land \Lambda M<k) \lor q \} \] \( \square \text{exists-part} \)
proof of the unless-part
Assume for arbitrary actions $a$, and states $s$ and $t$:

$A_{19}$: $a \in aQ$
$A_{20}$: compile.$a.s.t$
$A_{21}$: $J_P.s \land J_Q.s \land p.s \land (M.s = k) \land \text{guard}_Q.A_Q.s$
$A_{22}$: $\neg(M.s < k) \land \neg(q.s)$

We have to prove that:

$J_P.t \land J_Q.t \land p.t \land (M.t = k) \land \text{guard}_Q.A.Q.t$
\lor

$p.t \land (M.t < k)$
\lor

$q.t$

From lemma 2 and assumptions $A_{19}$, $A_{20}$, $A_{21}$ and $A_{22}$ we know that:

$A_{23}$: $(J_P.t \land J_Q.t \land p.t) \lor q.t$

If $q.t$ holds, then the proof has been established. So assume:

$A_{24}$: $\neg(q.t)$

Then assumptions $A_{23}$ and $A_{24}$ leave us with the proof obligation:

$((M.t = k) \land \text{guard}_Q.A.Q.t) \lor (M.t < k)$

From $A_0$, $A_{19}$, $A_{20}$, $A_{21}$, $A_{22}$ and the definition of unless (3.15) we can deduce:

$A_{26}$: $\text{guard}_Q.A_.p.s \Rightarrow (\text{guard}_Q.A.Q.t \land (M.t = k)) \lor \neg(\text{guard}_Q.A.P.t) \lor (M.t < k)$

From $A_1$, $A_3$, $A_{18}$, $A_{21}$, and the definition of action refinement (7.12), we can conclude $\text{guard}_Q.A_.p.s$, and hence assumption $A_{26}$ gives:

$A_{27}$: $(\text{guard}_Q.A.Q.t \land (M.t = k)) \lor \neg(\text{guard}_Q.A.P.t) \lor (M.t < k)$

Suppose $\text{guard}_Q.A_.p.t$ holds, then $A_{27}$ establishes the proof. To reach a contradiction, we assume that:

$A_{28}$: $\neg(\text{guard}_Q.A_.p.t)$

Now lemma 1, $A_{23}$, $A_{23}$, $A_{24}$ imply $q.t$ which obviously contradicts $A_{24}$.
\square

unless-part

proof of the exists-part:
The action that does the trick is $A_Q$ (introduced in $A_{18}$). From $A_1$ we know that $R$ is bitotal, and hence using $A_{15}$, $A_{18}$, and the definition of a bitotal relation we can infer that $A_Q$ is indeed an action in $aQ$. Assume for arbitrary states $s$ and $t$ that:

$A_{29}$: $J_P.s \land J_Q.s \land p.s \land (M.s = k) \land \text{guard}_Q.A.Q.s$
$A_{30}$: $\neg(M.s < k) \land \neg(q.s)$
$A_{31}$: compile.$A.Q.s.t$

38
We are left with the proof obligation:

\[(p.t \land (M.t < k)) \lor q.t\]

From \(A_{15}\) and the always-enabledness of actions in the universe \(\text{ACTION} (3.34)\) we know that there exists a state \(t'\) such that:

\(A_{32} : \text{compile}. \overline{P}. s.t'\)

and consequently from \(A_1, A_{18}\), the definition of action refinement (7.12), and assumptions \(A_3, A_{29}, A_{31}, A_{32}\) we can infer that:

\(A_{33} : t =_{wP} t'\)

From assumption \(A_{16}, A_{29}\), and \(A_{32}\) we can conclude that:

\(A_{34} : q.t'\)

Finally from \(A_{33}, A_{34}\), and \(A_{10}\) we can conclude \(q.t\).

\(\square \text{exists-part}\)

\(\square c_2\)

\(\square \text{true-guard-A}_r\text{-part}\)

end of proof of Theorem 7.7 (\(\implies\)-part)

References


42
Index

Y (non-increasing function over the variables of TARRY), 27
□a (always enabledness), 3
\(\rightarrow\) (convergence operator), 6
\(\rightarrow\) (ignored-by operator), 6
\(\rightarrow\) (invisible-to operator), 6
\(\mathcal{R}_{\text{PLUM ECHO}}, \mathcal{R}_{\text{PLUM TARRY}}, \mathcal{R}_{\text{TARRY DFS}}, \mathcal{R}_{\text{TARRY DFS}}
\(\subseteq_{V,J}\) (action refinement), 12
\(\subseteq_{R,J}\) (program refinement), 13
action refinement \((\subseteq_{V,J})\)
definition, 12
properties
  reflexivity, 12
  transitivity, 13
action systems, 7
actions
  augmented, 4, 10
  guard strengthening, 4
  refinement of
    definition of \((\subseteq_{V,J})\), 12
    properties of \((\subseteq_{V,J})\), 12–13
atomicity refinement, 6, 19
AUG\_S (augmentation superposition operator), 10
definition, 10
properties
  preservation of \(\sim\), 10
  preservation of ensures, 10
  preservation of \(\rightarrow\), 10
  preservation of unless, 10
augment, 4
augmentation superposition, 10, 18
definition, 10
properties
  preservation of \(\sim\), 10
  preservation of ensures, 10
  preservation of \(\rightarrow\), 10
  preservation of unless, 10
augmented action, 4
Back’s refinement calculus, 7
bitotal relation, 13
convergence \((\sim)\)
definition, 6
properties
  accumulation, 32
  bounded progress, 32
  case distinction, 32
  conjunction, 32
  disjunction, 32
  introduction, 32
  iteration, 32
  reflexivity, 32
  stable shift, 32
  stable strengthening, 32
  substitution, 32
  transitivity, 32
data refinement, 6
distributed hylomorphisms
  refinement ordering on, 23
distributed system, 7
guard strengthening, 7, 17
  of actions, 4
ignored-by operator \((\sim)\), 4
invisible-to operator \((\rightarrow)\), 4
\textit{MAX MAIL}, 26
mixed specification, 8, see also mspec
mspec, 8
non-determinism reducing refinement, 11, 18
parallel system, 7
program
  refinement, 3
    definition of \((\subseteq_{R,J})\), 13
    properties of \((\subseteq_{R,J})\), 14–16
  stepwise development, 3
  transformation, 6, see also refinement
program refinement \((\subseteq_{R,J})\)
definition, 13
properties
  preservation of \(\sim\), 16
  preservation of \(\rightarrow\), 16
  preservation of \(\subseteq\), 15
  preservation of unless, 15
  reflexivity, 14
  transitivity, 14
property refinement, 6
\(\mathcal{R}_{\text{PLUM ECHO}}, \mathcal{R}_{\text{PLUM TARRY}}, \mathcal{R}_{\text{TARRY DFS}}, \mathcal{R}_{\text{TARRY DFS}}
reach operator \((\rightarrow)\)
properties
  bounded progress, 31
  cancellation, 31
  case distinction, 31
  confinement, 31
  disjunction, 31
  introduction, 31
  PSP, 31
reflexivity, 31
stable background, 31
substitution, 31
transitivity, 31
well-founded induction, 31
refinement
calculus, 7
meaning of the word, 6
of actions
definition of \( (\sqsubseteq_v, j) \), 12
properties of \( (\sqsubseteq_v, j) \), 12–13
of programs
atomicity, 6, 19
data, 6
definition of \( (\sqsubseteq_{\mathcal{P}}, j) \), 13
guard strengthening, 17
properties of \( (\sqsubseteq_{\mathcal{P}}, j) \), 14–16
reducing non-determinism, 11, 18
strengthening guards, 7
superposition, 7, 17
of properties, 6
of specifications, 6, see also property refinement, refinement of properties
refinement mapping, 8
reification, 6, see also property refinement, refinement of properties
relation
bitotal, 13
restricted union superposition, 9, 18
definition, 9
properties
preservation of \( \sim \), 10
preservation of ensures, 10
preservation of \( \rightarrow \), 10
preservation of unless, 10
RU\_S (restricted union superposition operator), 9
definition, 9
properties
preservation of \( \sim \), 10
preservation of ensures, 10
preservation of \( \rightarrow \), 10
preservation of unless, 10
Sanders
mixed specification (\texttt{mspec}), 8
self-stabilisation, 6
Singh
refinements
fixed-point preserving, 8
property preserving, 8
total correctness preserving, 8
solution strategy, 6
specification refinement, 6, see also property refinement, refinement of properties
stepwise development of programs, 3

\texttt{strenghen\_guard}, 4
\texttt{strenghen\_guard (of an action)}
definition, 4
properties
preservation of \( \sim \), 4
preservation of \( \rightarrow \), 4
strengthening guards, 4, 7, 17
superposition refinement, 7, 17
 augmentation, 10, 18
definition, 10
properties, 10
restricted union, 9, 17
definition, 9
properties, 10

\texttt{TARRY}
as a UNITY program, 21
termination
UNITY specification
defs, 28
Echo, 24
plum, 23
TARRY, 25
\texttt{TOTAL\_NR\_REC}, 27
\texttt{TOTAL\_NR\_SENT}, 27
transformational programming, 3
transforming actions, 4
 augmentation, 4
strengthening guards, 4

\texttt{UNITY}
specification, 6
Unity (well-formedness of UNIT\texttt{Y} programs), 5
UNIT\texttt{Y} program
refinement
restricted union superposition, 9
augmentation superposition, 10
well-formedness of a, 5

variable
 ignored, 4
 invisible, 4
 verification condition, 7, 9, 10, 15
well-formedness
 of a UNIT\texttt{Y} program (Unity), 5
well-founded
 induction, 31
\texttt{Y}' (non-increasing function)
definition, 27
properties, 27