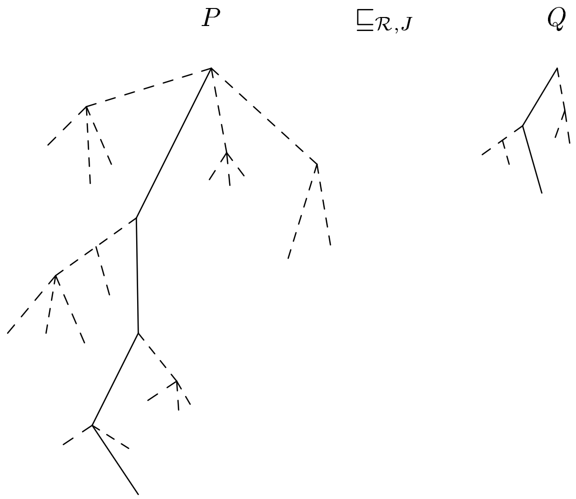
Program refinement in UNITY



T. E. J. Vos and **S. D. Swierstra**
Informatica Instituut, Utrecht University
e-mail: {tanja, doaitse}@cs.uu.nl

UU-CS-2001-41

December 23, 2001

Contents

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, Hoa72, 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 it 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 $\overset{d}{=}$ 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 $\mathsf{bitotal}.R.A.B$), 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 $\prec$ 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 forall 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 `Var` of program variables and a universe `Val` of values that these variables can take. Program states will be modelled as functions that are elements of `Var`$\rightarrow$`Val`, and the set of all program states will be denoted by `State`. A state-predicate is an element of `State`$\rightarrow$`bool`. We say that a state-predicate p is *confined by* a set of variables $V \subseteq$ `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 *CONF_DEF*

$$p\, \mathcal{C}\, V \overset{d}{=} \forall s, t : s =_V t : p.s = p.t$$

The confinement operator is monotonic in its second argument.

Theorem 3.2 $\mathcal{C}$ MONOTONICITY *CONF_MONO*

$$\forall f :: V \subseteq W \wedge (f\, \mathcal{C}\, V) \Rightarrow (f\, \mathcal{C}\, W)$$

3.2 Actions

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

All actions is 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 `ACTION`, and their semantics is defined by a recursive function, e.g. compile, of type `ACTION`$\rightarrow$(`State`$\rightarrow$`State`$\rightarrow$`Bool`). As a consequence, we are able to obtain and reason about various components of actions. For example, we assume that we have functions guard_of and assign_vars that given an action returns its guard and the set of variables it assigns to respectively. Examples of these functions:

guard_of(if $x > 0 \wedge y < 10$ then $x := x + 1 \,\|\, y := y - 1) = x > 0 \wedge y < 10$
assign_vars(if $x > 0 \wedge y < 10$ then $x := x + 1 \,\|\, y := y - 1) = \{x, y\}$

Moreover, we have functions is_assign and is_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 *ALWAYS_ENABLED*

$$\Box_{\mathsf{En}} A \overset{d}{=} \forall s :: (\exists t :: \mathsf{compile}.A.s.t)$$

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 *SKIP_DEF*

For any action A, $\mathsf{skip} \overset{d}{=} \mathsf{if\ false\ then}\ A$

A set of variables is V *ignored-by* an action A, denoted by $V \nleftarrow A$, if executing A's executable in any state does not change the values of these variables. Variables in V^{c} *may* however be written by A.

Definition 3.5 variables Ignored-by action *dIG_BY_DEF*

$$V \nleftarrow A \overset{d}{=} \forall s, t : \mathsf{compile}.A.s.t : s =_V t$$

A set of variables V is said to be *invisible-to* an action A, denoted by $V \not\twoheadrightarrow A$, 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 *dINVI_DEF*

$$V \not\twoheadrightarrow A \overset{d}{=} \forall s, t, s', t' : s =_{V^{\mathsf{c}}} s' \wedge t =_{V^{\mathsf{c}}} t' \wedge s' =_V t' \wedge \mathsf{compile}.A.s.t : \mathsf{compile}.A.s'.t'$$

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*

$$\mathsf{strengthen\_guard}.g.A \overset{d}{=} \mathsf{GUARD}.(g \wedge \mathsf{guard\_of}.A).(\mathsf{assign\_of}.A)$$

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

Definition 3.8 Augmenting an action *augment_DEF*

$$\mathsf{augment}.Ac.As \overset{d}{=} \mathsf{GUARD}.(\mathsf{guard\_of}.Ac).((\mathsf{assign\_of}.Ac) \parallel As)$$

When an action Ac is transformed by augmentation to yield $\mathsf{augment}.Ac.As$, 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 $\nleftarrow$ *streng_guard_PRESERVES_IG_BY*

$$\frac{V \nleftarrow A}{V \nleftarrow \mathsf{strengthen\_guard}.g.A}$$

Theorem 3.10 preservation of $\not\twoheadrightarrow$ *streng_guard_PRESERVES_INVI*

$$\frac{V \not\twoheadrightarrow A \wedge g\, \mathcal{C}\, V^{\mathsf{c}}}{V \not\twoheadrightarrow \mathsf{strengthen\_guard}.g.A}$$

Theorem 3.11 *streng_augment_COMMUTE*

$$\mathsf{strengthen\_guard}.g.(\mathsf{augment}.Ac.As) = \mathsf{augment}.(\mathsf{strengthen\_guard}.g.Ac).As$$

Theorem 3.12 preservation of $\nleftarrow$ *augment_PRESERVES_IG_BY*

$$\frac{V \nleftarrow Ac \wedge V \nleftarrow As \wedge \mathsf{is\_assign}.As}{V \nleftarrow \mathsf{augment}.Ac.As}$$

3.3 Programs

UNITY programs P are modelled by a quadruple $(\mathbf{a}P, \mathbf{ini}P, \mathbf{r}P, \mathbf{w}P)$; $\mathbf{a}P$, is the set of actions separated by the symbol $[]$; $\mathbf{ini}P$ is the initial condition of the program; $\mathbf{r}P$ is the set of read variables; and $\mathbf{w}P$ 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 *dUNITY*

$$\mathsf{Unity}.P \overset{d}{=} (\mathbf{a}P \neq \emptyset) \ \wedge \ (\mathbf{w}P \subseteq \mathbf{r}P) \ \wedge \ (\forall A : A \in \mathbf{a}P : (\mathbf{w}P)^{\mathsf{c}} \nleftarrow A) \ \wedge \ (\forall A : A \in \mathbf{a}P : (\mathbf{r}P)^{\mathsf{c}} \not\twoheadrightarrow A)$$

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 *HOAe_DEF*

$$\{p\}\ A\ \{q\} \ \overset{d}{=} \ \forall s, t : p.s \wedge \mathsf{compile}.A.s.t : q.t$$

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) *UNLESSe*

$${}_P\!\vdash p \ \mathsf{unless}\ q \ \overset{d}{=} \ \forall A : A \in \mathbf{a}P : \{p \wedge \neg q\}\ A\ \{p \vee q\}$$

Definition 3.16 Stable Predicate *STABLEe*

$${}_P\!\vdash \circlearrowright p \ \overset{d}{=} \ {}_P\!\vdash p \ \mathsf{unless\ false}$$

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

Theorem 3.17 Anti-Reflexivity *UNLESS_ANTI_REFL*

$${}_P\!\vdash \ p \ \mathsf{unless}\ \neg p$$

One-step progress properties are specified by:

Definition 3.18 Ensures (Progress Property) *ENSURESe*

$${}_P\!\vdash p \ \mathsf{ensures}\ q \ \overset{d}{=} \ ({}_P\!\vdash p \ \mathsf{unless}\ q) \ \wedge \ (\exists A : A \in \mathbf{a}P : \{p \wedge \neg q\}\ A\ \{q\})$$

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

Definition 3.19 Reach Operator *REACHe*
$(\lambda p, q.\ J\ {}_P\!\vdash p \rightarrowtail q)$ is defined as the smallest relation $\rightarrow$ satisfying:

Lifting $$\frac{p\,\mathcal{C}\,\mathbf{w}P \ \wedge\ q\,\mathcal{C}\,\mathbf{w}P \ \wedge\ ({}_{P}{\vdash}\circlearrowleft J) \ \wedge\ ({}_{P}{\vdash}\ J \wedge p \ \mathsf{ensures}\ q)}{p \rightarrow q}$$

Transitivity $$\frac{p \rightarrow q \wedge q \rightarrow r}{p \rightarrow r}$$

Disjunctivity $$\frac{\forall i : W.i : p_i \rightarrow q}{(\exists i : W.i : p_i) \rightarrow q}$$

where $W \in \alpha{\rightarrow}\mathtt{Val}$ characterises a non-empty set.

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

The $\rightsquigarrow$-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 $\rightarrowtail$:

Definition 3.20 CONVERGENCE *CONe*

$$J\ {}_{P}{\vdash}\ p \rightsquigarrow q \ \triangleq\ q\,\mathcal{C}\,\mathbf{w}P \ \wedge\ (\exists q' :: (J\ {}_{P}{\vdash}\ p \rightarrowtail q' \wedge q) \ \wedge\ ({}_{P}{\vdash}\circlearrowleft (J \wedge q' \wedge q)))$$

Again some properties taken from [Pra95] are listed in Appendix B. Most properties are analogous to those of $\rightarrowtail$. There is, however, one property that is satisfied by $\rightsquigarrow$ but not by $\rightarrowtail$ nor $\mapsto$, 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[1]. More specifically, a program P is refined by another program P' (denoted by $P \leq P'$ or $P \sqsubseteq P'$) 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 $t \in \mathit{programs} \rightarrow \mathit{programs}$ and added to the refinement calculus framework by proving theorems of the form:

$$\forall P \in \mathit{programs} :: \frac{\text{Verification Conditions hold for } P}{P \leq t.P}$$

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].

---

[1] 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 *mspecs*) to define a notion of *program* refinement in UNITY. An *mspec* incorporates both program text and a set of program properties. More specifically, an *mspec* consists of a *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 *mspec*); an *initially* section that contains a predicate that specifies the allowed initial values of the variables; an *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 *property* section containing a set of program properties (expressed in a modified[2] 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 assign section is empty, an *mspec* is a standard UNITY specification, and if the properties section is empty an *mspec* is a standard UNITY program. An *mspec* is called implementable when all properties in the property section can be proved to hold for the actions in the 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 *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 *mspec*s which is based on a refinement mapping [Lam83, LS84, AL88, Lam91, Lam96] $\mathcal{M}$ from the state space of the refinement to the state space of the original. It is denoted by $(G\ \mathit{refines}\ F)_{\mathcal{M}}$, and informally means:

- all initial conditions of G are mapped by $\mathcal{M}$ to the initial conditions on F
- if a state change from y_0 to y_1 is permitted by the assignments in the assign section of G, then either a state change from $\mathcal{M}.y_0$ to $\mathcal{M}.y_1$ is permitted by the the assignments in the assign section of F, or $\mathcal{M}.y_0$ equals $\mathcal{M}.y_1$.
- 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 *mspec* can be considered to hold in a refinement. To give an indication of what these theorems look like, the $\mapsto$ preservation theorem is copied below: [San90, page 13]:

$$\dfrac{\begin{array}{c} {}_F\vdash p \mapsto q \ \wedge\ (G\ \mathit{refines}\ F)_{\mathcal{M}} \\ \forall i\ :\ (\ {}_F\vdash r_i\ \mathsf{ensures}\ q_i \text{ is used in the proof of } {}_F\vdash p \mapsto q)\ :\ {}_G\vdash r_i \circ \mathcal{M}\ \ \mathsf{ensures}\ \ q_i \circ \mathcal{M} \end{array}}{{}_G\vdash p \circ \mathcal{M}\ \mapsto\ q \circ \mathcal{M}}$$

Similar theorems are given for preservation of unless , 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 *mspec*s, starting with an *mspec* containing a high level of abstraction, and ending up with an *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 511]:

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

- ${}_F\vdash p\ \mathsf{unless}\ q \ \Rightarrow\ {}_G\vdash p\ \mathsf{unless}\ q$
- ${}_F\vdash p \mapsto q \ \Rightarrow\ {}_G\vdash p \mapsto q$

Similarly, G is a *fixed-point preserving* refinement of F iff

- ${}_F\vdash \mathsf{true} \mapsto \mathsf{FP}_F \ \Rightarrow\ {}_G\vdash \mathsf{true} \mapsto \mathsf{FP}_G$
- $(\mathsf{FP}_G \wedge \mathsf{SI}_G)\ \Rightarrow\ (\mathsf{FP}_F \wedge \mathsf{SI}_F)$

where 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; SI_P denotes the strongest invariant of a program P, i.e. it denotes the set of states reachable from the initial state.

---

[2] The modified version was defined to eliminate the need of the substitution axiom [San91]

[3] The definitions of unless , ensures , and $\mapsto$ 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 $s :: A \textit{ if } p$ be a statement. Let statement $t :: A \textit{ if } p \wedge q$ be obtained by strengthening the guard of statement s. Then, program $F \,[\!]\, t$ is a property and a fixed-point preserving refinement of the program $F \,[\!]\, s$ if the following two conditions hold.

- ${}_F\!\vdash p \mapsto q$
- There exists a non-increasing function g from the program variables to a well-founded set such that ${}_F\!\vdash (g = k \;\wedge\; q)$ unless $(\neg p \;\vee\; g < k)$, for all k

In [Sin93] similar theorems are proved for program transformations like data refinement and atomicity refinement, and applied to 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 [ZGK90, Jon90, Kor91, Udi95, UK96, Din97, GKSU98].

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 `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 *RU_superpose_DEF*

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

$$\mathsf{RU\_S}.P.A.iA \quad = \quad P \;[\!]\; (\{A\}, iA, (\mathsf{assign\_vars}.A), (\mathsf{assign\_vars}.A))$$

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 $\mathsf{ensures}$ *RU_Superpose_PRESERVES_UNLESS* *RU_Superpose_PRESERVES_ENSURES*

$$\frac{p\,\mathcal{C}\,\mathbf{w}P \;\wedge\; q\,\mathcal{C}\,\mathbf{w}P \;\wedge\; \mathbf{w}P \not\leftarrow A}{\begin{array}{l} {}_P\!\vdash p \;\; \mathsf{unless} \;\; q \;\;\Rightarrow\;\; {}_{\mathsf{RU\_S}.P.A.iA}\!\vdash p \;\; \mathsf{unless} \;\; q \\ {}_P\!\vdash p \;\; \mathsf{ensures} \;\; q \;\;\Rightarrow\;\; {}_{\mathsf{RU\_S}.P.A.iA}\!\vdash p \;\; \mathsf{ensures} \;\; q \end{array}}$$

Theorem 6.3 PRESERVATION OF $\rightarrowtail$

RU_Superpose_PRESERVES_REACH
RU_Superpose_PRESERVES_CON

$$\frac{J\ \mathcal{C}\ \mathbf{w}P\ \wedge\ \mathbf{w}P \not\leftarrow A}{\begin{array}{l} J\ {}_{P}\vdash p \rightarrowtail q\ \Rightarrow\ J\ {}_{\mathsf{RU\_S}.P.A.iA}\vdash p \rightarrowtail q \\ J\ {}_{P}\vdash p \leadsto q\ \Rightarrow\ J\ {}_{\mathsf{RU\_S}.P.A.iA}\vdash p \leadsto q \end{array}}$$

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

AUG_superpose_DEF

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

$$\begin{array}{lcl} \mathsf{AUG\_S}.P.ACs.As.iA & = & (\{Ac \mid Ac \in \mathbf{a}P \wedge Ac \notin ACs\} \\ & & \quad \cup \\ & & \{\mathsf{augment}.Ac.As \mid Ac \in \mathbf{a}P \wedge Ac \in ACs\}, \\ & & \mathbf{ini}P\ \wedge\ iA, \\ & & \mathbf{r}P\ \cup\ (\mathsf{assign\_vars}.As), \\ & & \mathbf{w}P\ \cup\ (\mathsf{assign\_vars}.As)) \end{array}$$

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

AUG_Superpose_PRESERVES_UNLESS
AUG_Superpose_PRESERVES_ENSURES

$$\frac{p\ \mathcal{C}\ \mathbf{w}P\ \wedge\ q\ \mathcal{C}\ \mathbf{w}P\ \wedge\ \mathbf{w}P \not\leftarrow As\ \wedge\ \mathsf{is\_assign}.As}{\begin{array}{l} {}_{P}\vdash p\ \mathsf{unless}\ q\ \Rightarrow\ {}_{\mathsf{AUG\_S}.P.ACs.As.iA}\vdash p\ \mathsf{unless}\ q \\ {}_{P}\vdash p\ \mathsf{ensures}\ q\ \Rightarrow\ {}_{\mathsf{AUG\_S}.P.ACs.As.iA}\vdash p\ \mathsf{ensures}\ q \end{array}}$$

Theorem 6.6 PRESERVATION OF $\rightarrowtail$

AUG_Superpose_PRESERVES_REACH
AUG_Superpose_PRESERVES_CON

$$\frac{J\ \mathcal{C}\ \mathbf{w}P\ \wedge\ \mathbf{w}P \not\leftarrow A\ \wedge\ \mathsf{is\_assign}.As}{\begin{array}{l} J\ {}_{P}\vdash p \rightarrowtail q\ \Rightarrow\ J\ {}_{\mathsf{AUG\_S}.P.ACs.As.iA}\vdash p \rightarrowtail q \\ J\ {}_{P}\vdash p \leadsto q\ \Rightarrow\ J\ {}_{\mathsf{AUG\_S}.P.ACs.As.iA}\vdash p \leadsto q \end{array}}$$

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 $\mathsf{true}\ {}_P\!\vdash\ x = 0 \land y = 0 \leadsto x = 10 \land y = 10$ (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) $J \vdash x = 0 \land y = 0 \leadsto x = 10 \land y = 10$, and can be constructed by introducing a variable x_turn of type bool – the value of which indicates that it is x's turn – and transforming P as follows:

```
prog    $Q$
read    $\{x, y, x\_turn\}$
write   $\{x, y, x\_turn\}$
init    $x = 0 \land y = 0 \land x\_turn = \mathsf{true}$
assign  if $x < 10\ \land\ x\_turn$ then $x := x + 1 \parallel x\_turn := \neg x\_turn$        $Q_x$
[]      if $y < 10\ \land\ \neg x\_turn$ then $y := y + 1 \parallel x\_turn := \neg x\_turn$       $Q_y$
```

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 $x\_turn := \neg x\_turn$ to yield the program $\mathsf{AUG\_S}.P.\{P_x\}.(x\_turn := \neg x\_turn).(x\_turn = \mathsf{true})$, then we cannot subsequently augment action P_y (of $\mathsf{AUG\_S}.P.\{P_x\}.(x\_turn := \mathsf{false}).(x\_turn = \mathsf{true})$) with the assignment $x\_turn := \neg x\_turn$ and prove that the properties are preserved, since the write variables of $\mathsf{AUG\_S}.P.\{P_x\}.(x\_turn := \neg x\_turn).(x\_turn = \mathsf{true})$ (i.e. $\mathbf{w}P \cup \{x\_turn\}$) are *not* ignored by the assignment $x\_turn := \neg x\_turn$. 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 $x = 0 \land y = 0 \leadsto x = 10 \land y = 10$ 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

$\mathsf{true}\ {}_P\!\vdash\ x = 0 \wedge y = 0 \leadsto x = 10 \wedge y = 10$
$\Leftarrow$ ($\leadsto$ CONJUNCTION (B.11_{32}), $\leadsto$ SUBSTITUTION (B.2_{31}))
$\mathsf{true}\ {}_P\!\vdash\ x \leq 10 \leadsto x = 10\ \wedge\ \mathsf{true}\ {}_P\!\vdash\ y \leq 10 \leadsto y = 10$
We continue with the first conjunct, the proof of the second conjunct is similar.
$\mathsf{true}\ {}_P\!\vdash\ x \leq 10 \leadsto x = 10$
$\Leftarrow$ ($\leadsto$ BOUNDED PROGRESS (B.12_{32}), and ${}_P\!\vdash\!\circlearrowleft x = 10$)
$\mathsf{true}\ {}_P\!\vdash\ x \leq 10 \wedge (10 - x = k) \leadsto (x \leq 10 \wedge (10 - x < k)) \vee x = 10$
$\Leftarrow$ ($\leadsto$ CASE DISTINCTION (B.6_{32}) $x = 10 \vee x \neq 10$, $\leadsto$ REFLEXIVITY (B.4_{32}), ${}_P\!\vdash\!\circlearrowleft x = 10$,
and $\leadsto$ SUBSTITUTION (B.2_{31}))
$\mathsf{true}\ {}_P\!\vdash\ x < 10 \wedge (10 - x = k) \leadsto x \leq 10 \wedge (10 - x < k)$
$\Leftarrow$ ($\leadsto$ INTRODUCTION (B.3_{32}), and ${}_P\!\vdash\!\circlearrowleft x \leq 10 \wedge (10 - x < k)$)
${}_P\!\vdash\ x < 10 \wedge (10 - x = k)\ \mathsf{ensures}\ x \leq 10 \wedge (10 - x < k)$

Figure 3: Proof of $\mathsf{true}\ {}_P\!\vdash\ x = 0 \wedge y = 0 \leadsto x = 10 \wedge y = 10$

◄

to decompose the proof obligation $x = 0 \wedge y = 0 \leadsto x = 10 \wedge y = 10$ into the simpler proof obligations $x = 0 \leadsto x = 10$ and $y = 0 \leadsto y = 10$. 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 $A_l, A_r \in$ `ACTION`, a state-predicate J, and a set of variables V, we say that A_l is refined by A_r, or A_r refines A_l, with respect to V and J (denoted by $A_l \sqsubseteq_{V,J} A_r$), when:

- the conjunction of J with the guard of A_r is stronger then the guard of A_l.
- the results of A_l 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 REFINEMENT *A_ref_DEF*
Let A_l 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:

$$\begin{array}{rcl} A_l \sqsubseteq_{V,J} A_r & = & \forall s\ ::\ \mathsf{guard\_of}.A_r.s \wedge J.s \Rightarrow \mathsf{guard\_of}.A_l.s \\ & & \wedge \\ & & \forall s, t, t'\ ::\ (\mathsf{compile}.A_l.s.t \wedge \mathsf{compile}.A_r.s.t' \wedge \mathsf{guard\_of}.A_r.s \wedge J.s) \Rightarrow t =_V t' \end{array}$$

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

Theorem 7.2 ACTION REFINEMENT REFLEXIVITY *A_ref_REFL*
For all $A \in$ `ACTION`, state-predicates J, and sets of variables V:

$$A \sqsubseteq_{V,J} A$$

Theorem 7.3 ACTION REFINEMENT TRANSITIVITY *A_ref_TRANS1*
For all $A_1, A_2, A_3 \in$ `ACTION`, state-predicates J_2 and J_3, and sets of variables V_1, V_2 and V_3:

$$\frac{J_3 \Rightarrow J_2\ \wedge\ V_3 \subseteq V_1\ \wedge\ V_3 \subseteq V_2\ \wedge\ A_1 \sqsubseteq_{V_1,J_2} A_2\ \wedge\ A_2 \sqsubseteq_{V_2,J_3} A_3}{A_1 \sqsubseteq_{V_3,J_3} A_3}$$

Take $J = (\neg x\_turn \Rightarrow y = x - 1) \wedge (x\_turn \Rightarrow y = x)$, and prove that ${}_Q\vdash\circlearrowleft J$.

$J\ {}_Q\vdash\ x = 0 \wedge y = 0 \rightsquigarrow x = 10 \wedge y = 10$
$\Leftarrow$ ($\rightsquigarrow$ SUBSTITUTION (B.2_{31}))
$J\ {}_Q\vdash\ x \leq 10 \wedge y \leq 10 \rightsquigarrow x = 10 \wedge y = 10$
$\Leftarrow$ ($\rightsquigarrow$ BOUNDED PROGRESS (B.12_{32}), and ${}_Q\vdash\circlearrowleft x = 10 \wedge y = 10$)
$J\ {}_Q\vdash\ x \leq 10 \wedge y \leq 10 \wedge (20 - x - y = k) \rightsquigarrow (x \leq 10 \wedge y \leq 10 \wedge (20 - x - y < k)) \vee (x = 10 \wedge y = 10)$
$\Leftarrow$ ($\rightsquigarrow$ CASE DISTINCTION (B.6_{32}) $y = 10 \vee y \neq 10$, $\rightsquigarrow$ INTRODUCTION (B.3_{32}), and
$\rightsquigarrow$ SUBSTITUTION (B.2_{31}) and $(J \wedge x \leq 10 \wedge y \leq 10 \wedge y = 10) \Rightarrow (x = 10 \wedge y = 10)$)
$J\ {}_Q\vdash\ x \leq 10 \wedge y < 10 \wedge (20 - x - y = k) \rightsquigarrow (x \leq 10 \wedge y \leq 10 \wedge (20 - x - y < k)) \vee (x = 10 \wedge y = 10)$
$\Leftarrow$ ($\rightsquigarrow$ CASE DISTINCTION (B.6_{32}) $x = 10 \vee x \neq 10$, and $\rightsquigarrow$ SUBSTITUTION (B.2_{31}))
$J\ {}_Q\vdash\ x = 10 \wedge y < 10 \wedge (20 - x - y = k) \rightsquigarrow (x \leq 10 \wedge y \leq 10 \wedge (20 - x - y < k))$
$\wedge$
$J\ {}_Q\vdash\ x < 10 \wedge y < 10 \wedge (20 - x - y = k) \rightsquigarrow (x \leq 10 \wedge y \leq 10 \wedge (20 - x - y < k))$
The first conjunct can be proved by $\rightsquigarrow$ INTRODUCTION (B.3_{32}), since $(J \wedge x = 10 \wedge y < 10)$ implies $\neg x\_turn$, and thus
${}_Q\vdash J \wedge x = 10 \wedge y < 10 \wedge (20 - x - y = k)$ ensures $x \leq 10 \wedge y \leq 10 \wedge (20 - x - y < k)$
We continue with the second conjunct as follows:
$J\ {}_Q\vdash\ x < 10 \wedge y < 10 \wedge (20 - x - y = k) \rightsquigarrow x \leq 10 \wedge y \leq 10 \wedge (20 - x - y < k)$
$\Leftarrow$ ($\rightsquigarrow$ CASE DISTINCTION (B.6_{32}) $(x\_turn \vee \neg(x\_turn))$)
$J\ {}_Q\vdash\ x < 10 \wedge y < 10 \wedge (20 - x - y = k) \wedge x\_turn \rightsquigarrow x \leq 10 \wedge y \leq 10 \wedge (20 - x - y < k)$
$\wedge$
$J\ {}_Q\vdash\ x < 10 \wedge y < 10 \wedge (20 - x - y = k) \wedge \neg(x\_turn) \rightsquigarrow x \leq 10 \wedge y \leq 10 \wedge (20 - x - y < k)$
$\Leftarrow$ ($\rightsquigarrow$ INTRODUCTION (B.3_{32}) on both conjuncts)
${}_Q\vdash\ J \wedge x < 10 \wedge y < 10 \wedge (20 - x - y = k) \wedge x\_turn$ ensures $x \leq 10 \wedge y \leq 10 \wedge (20 - x - y < k)$
$\wedge$
${}_Q\vdash\ J \wedge x < 10 \wedge y < 10 \wedge (20 - x - y = k) \wedge \neg x\_turn$ ensures $x \leq 10 \wedge y \leq 10 \wedge (20 - x - y < k)$

Figure 4: Proof of $J\ {}_Q\vdash\ x = 0 \wedge y = 0 \rightsquigarrow x = 10 \wedge y = 10$

Next, we define our relation of program refinement. P is refined by Q, or Q refines P, with respect to some relation $\mathcal{R}$ and state-predicate J, (denoted by $P \sqsubseteq_{\mathcal{R},J} Q$), if we can decompose the actions of program Q into $\mathbf{a}Q_1$ and $\mathbf{a}Q_2$, such that

- $\mathcal{R}$ is a bitotal relation on the two sets of actions $\mathbf{a}P$ and $\mathbf{a}Q_1$, i.e. for every action A_P in $\mathbf{a}P$ there exists at least one action in $\mathbf{a}Q_1$ to which $\mathbf{a}P$ is related by $\mathcal{R}$, and similarly for every action A_Q in $\mathbf{a}Q_1$ there exists at least one action in $\mathbf{a}P$ to which A_Q is related by $\mathcal{R}$.
- for all actions A_P of $\mathbf{a}P$ and A_Q of $\mathbf{a}Q_1$ that are related to each other by $\mathcal{R}$ (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 $\mathbf{a}Q_2$ 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 *P_ref_DEF*
Let P and Q be two UNITY programs, $\mathcal{R}$ be a relation, and J be a state predicate, then program refinement is defined as follows:

$$\begin{array}{lll} P \sqsubseteq_{\mathcal{R},J} Q & = \exists \mathbf{a}Q_1, \mathbf{a}Q_2 :: & \mathbf{a}Q = \mathbf{a}Q_1 \cup \mathbf{a}Q_2\ \wedge\ \mathsf{bitotal}.\mathcal{R}.\mathbf{a}P.\mathbf{a}Q_1 \\ & & \wedge \\ & & \forall A_P\ A_Q\ :\ A_P \in \mathbf{a}P\ \wedge\ A_P\ \mathcal{R}\ A_Q\ :\ A_P \sqsubseteq_{\mathbf{w}P,J} A_Q \\ & & \wedge \\ & & \forall A_Q\ :\ A_Q \in \mathbf{a}Q_2\ :\ \mathsf{skip} \sqsubseteq_{\mathbf{w}P,J} A_Q \end{array}$$

Note that $P \sqsubseteq_{\mathcal{R},J} Q$ 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, $P \sqsubseteq_{\mathcal{R},J} Q$, does not necessarily imply that $\mathbf{w}P \subseteq \mathbf{w}Q$. Consider the two programs P and Q in Figure 6. Suppose z and

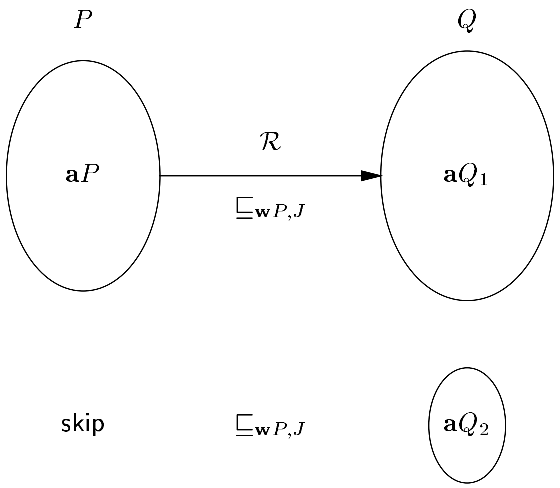


Figure 5: Program refinement in a picture.

```
prog    $P$
read    $\{x, y, z\}$
write   $\{x, y, z\}$
init    $b = \mathsf{true}$
assign  $x := x + 1$   $aP_1$
[]      $y := y + 1$   $aP_2$
[]      $z := z$       $aP_3$
```

```
prog    $Q$
read    $\{x, y, w\}$
write   $\{x, y, w\}$
init    $b = \mathsf{true}$
assign  if $x \leq 15$ then $x := x + 1$   $aQ_1$
[]      if $y \leq 20$ then $y := y + 1$   $aQ_2$
[]      $w := w + 1$                       $aQ_3$
```

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, $\mathbf{w}P \subseteq \mathbf{w}Q$ does not hold.

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

Theorem 7.5 Program refinement Reflexivity `P_ref_REFL`

For all programs P, and state-predicate J:

$$P \sqsubseteq_{=,J} P$$

Theorem 7.6 Program refinement Transitivity `P_ref_TRANS`

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

$$\frac{J_3 \Rightarrow J_2 \ \wedge\ \mathbf{w}P_1 \subseteq \mathbf{w}P_2 \ \wedge\ P_1 \sqsubseteq_{\mathcal{R}_1,J_2} P_2 \ \wedge\ P_2 \sqsubseteq_{\mathcal{R}_2,J_3} P_3}{P_1 \sqsubseteq_{\mathcal{R}_1 \circ \mathcal{R}_2, J_3} P_3}$$

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:

$$P_1 \sqsubseteq_{\mathcal{R}_1,J_2} P_2 \sqsubseteq_{\mathcal{R}_2,J_3} P_3 \sqsubseteq_{\mathcal{R}_3,J_4} P_4 \ldots \sqsubseteq_{\mathcal{R}_n,J_{n+1}} P_{n+1}$$

in order to conclude that

$$P_1 \sqsubseteq_{\mathcal{R}_1 \circ \ldots \circ \mathcal{R}_n, J_{n+1}} P_{n+1}$$

we have to prove that:

- the write variables of the program P_i in intermediate step $P_i \sqsubseteq_{\mathcal{R}_1 \circ \ldots \circ \mathcal{R}_i, J_{i+1}} P_{i+1}$ 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 $\circlearrowleft J$, 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 `P_ref_AND_SUPERPOSE_WRITE_PRESERVES_UNLESSe`

$$\frac{\begin{array}{c} P \sqsubseteq_{\mathcal{R},J} Q \wedge \mathsf{Unity}.P \wedge \mathsf{Unity}.Q \wedge ({}_Q\vdash \circlearrowleft J_Q) \wedge (J_Q \Rightarrow J) \\ \exists W :: (\mathbf{w}Q = \mathbf{w}P \cup W) \wedge (p\ \mathcal{C}\ W^{\mathsf{c}}) \wedge (q\ \mathcal{C}\ W^{\mathsf{c}}) \end{array}}{{}_P\vdash p\ \mathsf{unless}\ q \Rightarrow\ {}_Q\vdash (J_Q \wedge p)\ \mathsf{unless}\ q}$$

Theorem 7.12 $\circlearrowleft$ PRESERVATION `P_ref_AND_SUPERPOSE_WRITE_PRESERVES_STABLEe`

$$\frac{\begin{array}{c} P \sqsubseteq_{\mathcal{R},J} Q \wedge \mathsf{Unity}.P \wedge \mathsf{Unity}.Q \wedge ({}_Q\vdash \circlearrowleft J_Q) \wedge (J_Q \Rightarrow J) \\ \exists W :: (\mathbf{w}Q = \mathbf{w}P \cup W) \wedge (p\ \mathcal{C}\ W^{\mathsf{c}}) \end{array}}{{}_P\vdash \circlearrowleft p \Rightarrow\ {}_Q\vdash \circlearrowleft (J_Q \wedge p)}$$

The conditions $(p\ \mathcal{C}\ W^{\mathsf{c}})$ and $(q\ \mathcal{C}\ W^{\mathsf{c}})$, 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 then stating that p and q do not name any superposed variable.

Preservation of one-step progress properties (i.e. $\mathsf{ensures}$) 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 $\mathcal{R}$ and J, then the progress properties $p \rightarrowtail q$ and $p \leadsto q$ under the stability of predicate J_P in program P, are preserved under the stability of predicate $J_P \wedge J_Q$ in program Q, provided that the following verification conditions hold:

- $(J_P \wedge J_Q)$ is stable in Q.
- $(J_P \wedge J_Q)$ 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 $\mathcal{R}$ 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 $\mathcal{R}$, 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 Q, *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 $\prec$ be a well-founded relation over some set A, and $M \in \mathtt{State}{\to}A$.

Theorem 7.7

P_ref_SUPERPOSE_AND_WF_FUNC_PRESERVES_REACHe_GEN
P_ref_SUPERPOSE_AND_WF_FUNC_PRESERVES_CONe_GEN

$$\begin{array}{c}
P \sqsubseteq_{R,J} Q \land \mathsf{Unity}.P \land \mathsf{Unity}.Q \land (\ {}_{Q}\!\vdash\circlearrowright J_P \land J_Q) \land (J_P \land J_Q \Rightarrow J) \\
\exists W :: (\mathbf{w}Q = \mathbf{w}P \cup W) \land (J_P\ \mathcal{C}\ W^{\mathsf{c}}) \land (\mathbf{w}P \subseteq W^{\mathsf{c}}) \\
\forall A_Q : A_Q \in \mathbf{a}Q \land (\exists A_P :: (A_P \in \mathbf{a}P) \land (A_P\ \mathcal{R}\ A_Q)) : (\mathsf{guard\_of}.A_Q\ \mathcal{C}\ \mathbf{w}Q) \\
\forall A_P : A_P \in \mathbf{a}P : (J_P \land J_Q)\ {}_{Q}\!\vdash \mathsf{guard\_of}.A_P \rightarrowtail (\exists A_Q :: (A_P\ \mathcal{R}\ A_Q) \land \mathsf{guard\_of}.A_Q) \\
\exists M :: (M\ \mathcal{C}\ \mathbf{w}Q) \land (\forall k : k \in A :\ {}_{Q}\!\vdash (J_P \land J_Q \land M = k)\ \mathsf{unless}\ (M \prec k)) \\
\quad \land \forall k\ A_P A_Q : k \in A \land A_P \in \mathbf{a}P \land A_P\ \mathcal{R}\ A_Q : \\
\qquad {}_{Q}\!\vdash (J_P \land J_Q \land \mathsf{guard\_of}.A_Q \land M = k)\ \mathsf{unless}\ (\neg(\mathsf{guard\_of}.A_P) \lor M \prec k) \\
\hline
((J_P\ {}_{P}\!\vdash p \rightarrowtail q) \Rightarrow (J_P \land J_Q\ {}_{Q}\!\vdash p \rightarrowtail q))\ \land\ ((J_P\ {}_{P}\!\vdash p \leadsto q) \Rightarrow (J_P \land J_Q\ {}_{Q}\!\vdash p \leadsto q))
\end{array}$$

Theorem 7.8

P_ref_SUPERPOSE_PRESERVES_REACHe_GEN
P_ref_SUPERPOSE_PRESERVES_CONe_GEN

$$\begin{array}{c}
P \sqsubseteq_{R,J} Q \land \mathsf{Unity}.P \land \mathsf{Unity}.Q \land (\ {}_{Q}\!\vdash\circlearrowright J_P \land J_Q) \land (J_P \land J_Q \Rightarrow J) \\
\exists W :: (\mathbf{w}Q = \mathbf{w}P \cup W) \land (J_P\ \mathcal{C}\ W^{\mathsf{c}}) \land (\mathbf{w}P \subseteq W^{\mathsf{c}}) \\
\forall A_Q : A_Q \in \mathbf{a}Q \land (\exists A_P :: (A_P \in \mathbf{a}P) \land (A_P\ \mathcal{R}\ A_Q)) : (\mathsf{guard\_of}.A_Q\ \mathcal{C}\ \mathbf{w}Q) \\
\forall A_P : A_P \in \mathbf{a}P : (J_P \land J_Q)\ {}_{Q}\!\vdash \mathsf{guard\_of}.A_P \rightarrowtail (\exists A_Q :: (A_P\ \mathcal{R}\ A_Q) \land \mathsf{guard\_of}.A_Q) \\
\forall A_P\ A_Q : A_P \in \mathbf{a}P \land A_P\ \mathcal{R}\ A_Q :\ {}_{Q}\!\vdash (J_P \land J_Q \land \mathsf{guard\_of}.A_Q)\ \mathsf{unless}\ \neg(\mathsf{guard\_of}.A_P) \\
\hline
((J_P\ {}_{P}\!\vdash p \rightarrowtail q) \Rightarrow (J_P \land J_Q\ {}_{Q}\!\vdash p \rightarrowtail q))\ \land\ ((J_P\ {}_{P}\!\vdash p \leadsto q) \Rightarrow (J_P \land J_Q\ {}_{Q}\!\vdash p \leadsto q))
\end{array}$$

Theorem 7.9

P_ref_SUPERPOSE_AND_WF_FUNC_PRESERVES_REACHe
P_ref_SUPERPOSE_AND_WF_FUNC_PRESERVES_CONe

$$\begin{array}{c}
P \sqsubseteq_{R,J} Q \land \mathsf{Unity}.P \land \mathsf{Unity}.Q \land (\ {}_{Q}\!\vdash\circlearrowright J_P \land J_Q) \land (J_P \land J_Q \Rightarrow J) \\
\exists W :: (\mathbf{w}Q = \mathbf{w}P \cup W) \land (J_P\ \mathcal{C}\ W^{\mathsf{c}}) \land (\mathbf{w}P \subseteq W^{\mathsf{c}}) \\
\forall A_P\ A_Q\ :\ A_P \in \mathbf{a}P\ \land\ A_P\ \mathcal{R}\ A_Q\ : (J_P \land J_Q)\ {}_{Q}\!\vdash \mathsf{guard\_of}.A_P \rightarrowtail \mathsf{guard\_of}.A_Q \\
\exists M :: (M\ \mathcal{C}\ \mathbf{w}Q) \land (\forall k : k \in A :\ {}_{Q}\!\vdash (J_P \land J_Q \land M = k)\ \mathsf{unless}\ (M \prec k)) \\
\quad \land \forall k\ A_P A_Q : k \in A \land A_P \in \mathbf{a}P \land A_P\ \mathcal{R}\ A_Q : \\
\qquad {}_{Q}\!\vdash (J_P \land J_Q \land \mathsf{guard\_of}.A_Q \land M = k)\ \mathsf{unless}\ (\neg(\mathsf{guard\_of}.A_P) \lor M \prec k) \\
\hline
((J_P\ {}_{P}\!\vdash p \rightarrowtail q) \Rightarrow (J_P \land J_Q\ {}_{Q}\!\vdash p \rightarrowtail q))\ \land\ ((J_P\ {}_{P}\!\vdash p \leadsto q) \Rightarrow (J_P \land J_Q\ {}_{Q}\!\vdash p \leadsto q))
\end{array}$$

Theorem 7.10

P_ref_AND_SUPERPOSE_WRITE_PRESERVES_REACHe
P_ref_AND_SUPERPOSE_WRITE_PRESERVES_CONe

$$\begin{array}{c}
P \sqsubseteq_{R,J} Q \land \mathsf{Unity}.P \land \mathsf{Unity}.Q \land (\ {}_{Q}\!\vdash\circlearrowright J_P \land J_Q) \land (J_P \land J_Q \Rightarrow J) \\
\exists W :: (\mathbf{w}Q = \mathbf{w}P \cup W) \land (J_P\ \mathcal{C}\ W^{\mathsf{c}}) \land (\mathbf{w}P \subseteq W^{\mathsf{c}}) \\
\forall A_P\ A_Q\ :\ A_P \in \mathbf{a}P\ \land\ A_P\ \mathcal{R}\ A_Q\ : (J_P \land J_Q)\ {}_{Q}\!\vdash \mathsf{guard\_of}.A_P \rightarrowtail \mathsf{guard\_of}.A_Q \\
\forall A_P\ A_Q\ :\ A_P \in \mathbf{a}P\ \land\ A_P\ \mathcal{R}\ A_Q\ :\ {}_{Q}\!\vdash (J_P \land J_Q \land \mathsf{guard\_of}.A_Q)\ \mathsf{unless}\ \neg(\mathsf{guard\_of}.A_P) \\
\hline
((J_P\ {}_{P}\!\vdash p \rightarrowtail q) \Rightarrow (J_P \land J_Q\ {}_{Q}\!\vdash p \rightarrowtail q))\ \land\ ((J_P\ {}_{P}\!\vdash p \leadsto q) \Rightarrow (J_P \land J_Q\ {}_{Q}\!\vdash p \leadsto q))
\end{array}$$

Figure 7: Preservation of $\rightarrowtail$ and $\leadsto$ properties.

- for all actions A_P of program P and those actions A_Q of Q that are related to A_P by $\mathcal{R}$, 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 *BITOTAL_IMP_GUARD_REACH_EXIST_GUARD*

$$\frac{\begin{array}{c}(\exists A :: \mathsf{bitotal}.\mathcal{R}.\mathbf{a}P.A)\\ \forall A_P\ A_Q\ :\ A_P \in \mathbf{a}P\ \wedge\ A_P\ \mathcal{R}\ A_Q\ : J\ {}_Q{\vdash}\ \mathsf{guard\_of}.A_P \rightarrowtail \mathsf{guard\_of}.A_Q\end{array}}{\forall A_P : A_P \in \mathbf{a}P : J\ {}_Q{\vdash}\ \mathsf{guard\_of}.A_P \rightarrowtail (\exists A_Q :: (A_P\ \mathcal{R}\ A_Q) \wedge \mathsf{guard\_of}.A_Q)}$$

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 *augment_A_ref*

For all $A, As \in$ `ACTION`, state-predicates J, and V a set of variables:

$$\frac{\mathsf{is\_assign}.As \wedge V \nleftarrow As \wedge \mathsf{WF\_action}.A \wedge \mathsf{WF\_action}.As}{A \sqsubseteq_{V,J} \mathsf{augment}.A.As}$$

Theorem 7.15 *strengthen_guard_A_ref*

For all $A \in$ `ACTION`, state-predicates g and J, and V a set of variables:

$$A \sqsubseteq_{V,J} \mathsf{strengthen\_guard}.g.A$$

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

Theorem 7.16 *RU_Superpose_P_ref*

For all programs P, $A \in$ `ACTION`, state-predicates J and iA:

$$\frac{\mathbf{w}P \nleftarrow A}{P \sqsubseteq_{=,J} \mathsf{RU\_S}.P.A.iA}$$

Theorem 7.17 *AUG_Superpose_P_ref*

For all programs P, $As \in$ `ACTION`, state-predicate iA, and $ACs \subseteq$ `ACTION`:

$$\frac{\mathbf{w}P \nleftarrow A\ \wedge\ \mathsf{is\_assign}.As\ \wedge\ \mathsf{WF\_action}.As}{\exists \mathcal{R}\ ::\ P \sqsubseteq_{\mathcal{R},J} \mathsf{AUG\_S}.P.ACs.As.iA}$$

The witness used to prove this theorem is[4]: $(\mathcal{R} = \mathsf{f2r}.(\lambda A.(A \in ACs) \rightarrow \mathsf{augment}.A.As \mid A))$.

---

[4] Where the function $\mathsf{f2r}.f = (\lambda x, y.y = f.x)$, i.e. converts a function to a relation.

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

$$\forall A_P\ A_Q\ :\ A_P \in \mathbf{a}P\ \wedge\ A_P\ \mathcal{R}\ A_Q\ :\ A_P \sqsubseteq_{\mathbf{w}P,J} A_Q$$

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

$$\forall A_Q\ :\ A_Q \in \mathbf{a}Q_2\ :\ \mathsf{skip} \sqsubseteq_{\mathbf{w}P,J} A_Q$$

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)

$\exists \mathbf{a}Q_1, \mathbf{a}Q_2 ::\quad \mathbf{a}Q = \mathbf{a}Q_1 \cup \mathbf{a}Q_2\ \wedge\ \mathsf{bitotal}.\mathcal{R}.\mathbf{a}P.\mathbf{a}Q_1$

$\wedge$

$\forall A_P\ A_Q\ :\ A_P \in \mathbf{a}P\ \wedge\ A_P\ \mathcal{R}\ A_Q\ :\ A_P \sqsubseteq_{\mathbf{w}P,J} A_Q$

$\wedge$

$\forall A_Q\ :\ A_Q \in \mathbf{a}Q_2\ :\ \mathsf{skip} \sqsubseteq_{\mathbf{w}P,J} A_Q$

$\Leftarrow$ (Reduce goal using witnesses $\mathbf{a}Q$ and $\emptyset$ respectively)

$\mathbf{a}Q = \mathbf{a}Q \cup \emptyset\ \wedge\ \mathsf{bitotal}.\mathcal{R}.\mathbf{a}P.\mathbf{a}Q$

$\wedge\ (\forall A_P\ A_Q\ :\ A_P \in \mathbf{a}P\ \wedge\ A_P\ \mathcal{R}\ A_Q\ :\ A_P \sqsubseteq_{\mathbf{w}P,J} A_Q)\ \wedge\ (\forall A_Q\ :\ A_Q \in \emptyset\ :\ \mathsf{skip} \sqsubseteq_{\mathbf{w}P,J} A_Q)$

$\Leftarrow$ ($\mathcal{R}$ is a bitotal; properties of $\cup$, $\in$, and $\emptyset$)

$\forall A_P\ A_Q\ :\ A_P \in \mathbf{a}P\ \wedge\ A_P\ \mathcal{R}\ A_Q\ :\ A_P \sqsubseteq_{\mathbf{w}P,J} A_Q$

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

$P_x \sqsubseteq_{\mathbf{w}P,J} Q_x\ \wedge\ P_y \sqsubseteq_{\mathbf{w}P,J} Q_y$

$=$ (We shall prove the one for P_x the other is similar; Rewrite with Definition 7.1)

$\forall s\ ::\ \mathsf{guard\_of}.Q_x.s \wedge J.s \Rightarrow \mathsf{guard\_of}.P_x.s$

$\wedge\ \forall s, t, t'\ ::\ (\mathsf{compile}.P_x.s.t \wedge \mathsf{compile}.Q_x.s.t' \wedge \mathsf{guard\_of}.Q_x.s \wedge J.s) \Rightarrow t =_{\mathbf{w}P} t'$

$=$ ($\mathsf{guard\_of}.Q_x.s = (s.x \leq 10 \wedge\ s.x\_turn)$, and $\mathsf{guard\_of}.P_x.s = s.x \leq 10$)

$\forall s, t, t'\ ::\ (\mathsf{compile}.P_x.s.t \wedge \mathsf{compile}.Q_x.s.t' \wedge s.x \leq 10 \wedge s.x\_turn \wedge J.s) \Rightarrow t =_{\mathbf{w}P} t'$

Discharge the antecedents of this goal into the assumptions after rewriting with P_x and Q_x

$\mathbf{A}_1$: $\mathsf{if}\ s.x \leq 10\ \mathsf{then}\ t.x := s.x + 1$

$\mathbf{A}_2$: $\mathsf{if}\ s.x \leq 10 \wedge s.x\_turn\ \mathsf{then}\ t.x, t.x\_turn := s.x + 1, \mathsf{false}$

$\mathbf{A}_3$: $s.x \leq 10\ \wedge\ s.x\_turn\ \wedge\ J.s$

From these assumptions it is easy to deduce that $t =_{\{x,y\}} t'$ which equals $t =_{\mathbf{w}P} t'$.

end of proof

Proving that the property $\mathsf{true}\ {}_P\!\vdash\ x = 0 \wedge y = 0 \rightsquigarrow x = 10 \wedge y = 10$ of program P is indeed preserved by its non-determinism reducing refinement Q can be established using Theorem 7.9. We already have that:

$\mathbf{A}_1$: $\mathsf{true}\ {}_P\!\vdash\ x = 0 \wedge y = 0 \rightsquigarrow x = 10 \wedge y = 10$

$\mathbf{A}_2$: $\mathcal{R} = \{(P_i, Q_i) \mid i \in \{x, y\}\}$

$\mathbf{A}_3$: $J = (\neg x\_turn \Rightarrow (y = x - 1)) \vee (x\_turn \Rightarrow (x = y))$

$\mathbf{A}_3$: ${}_Q\!\vdash\circlearrowright J$

$\mathbf{A}_3$: $P \sqsubseteq_{\mathcal{R},J} Q$

Now Theorem 7.9, using witnesses $W = \{x\_turn\}$ and $M = 20 - x - y$, and taking $\prec$ to be $<$ on numbers, leaves us with the following proof obligations:

- ${}_Q\!\vdash (J \wedge M = k)$ unless $(M < k)$
- ${}_Q\!\vdash (J \wedge y < 10 \wedge \neg(x\_turn) \wedge M = k)$ unless $(\neg(y < 10) \vee M < k)$
- ${}_Q\!\vdash (J \wedge x < 10 \wedge x\_turn \wedge M = k)$ unless $(\neg(x < 10) \vee M < k)$
- $J\ {}_Q\!\vdash x < 10 \rightarrowtail x < 10\ \wedge\ x\_turn$

```
prog    $P$
read    $\mathbf{r}P$
write   $\mathbf{w}P$
init    $\mathbf{ini}P$
assign  if $(\exists i\ : i \in S :\ g.i)$ then $A$
```

```
prog    $Q$
read    $\mathbf{r}P$
write   $\mathbf{w}P$
init    $\mathbf{ini}P$
assign  $[]_{i \in S}$ if $g.i$ then $A$
```

Figure 8: Q refines P

- $J\ {}_{Q}\!\vdash y < 10 \rightarrowtail y < 10\ \wedge\ \neg x\_turn$

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 $p = \textbf{if}\ (\exists i\ : i \in S :\ g.i)\ \textbf{then}\ A$ and $q.i = \textbf{if}\ g.i\ \textbf{then}\ A$. It easy to prove that the relation $\mathcal{R} = \{(p, q.i) \mid i \in S\}$ is bitotal on $\mathbf{a}P$ and $\mathbf{a}Q$, and consequently that for any J, $P \sqsubseteq_{\mathcal{R},J} Q$. To determine the conditions that need to be satisfied in order to conclude property preservation, Theorem 7.8_{16} can be used to conclude:

$$\frac{\begin{array}{c}\forall i : i \in S : g.i\ \mathcal{C}\ \mathbf{w}Q \\ \forall i : i \in S :\ {}_{Q}\!\vdash (J_P \wedge J_Q \wedge g.i)\ \mathsf{unless}\ \neg(\exists i : i \in S : g.i)\end{array}}{((J_P\ {}_{P}\!\vdash p \rightarrowtail q) \Rightarrow (J_P \wedge J_Q\ {}_{Q}\!\vdash p \rightarrowtail q))\ \wedge\ ((J_P\ {}_{P}\!\vdash p \leadsto q) \Rightarrow (J_P \wedge J_Q\ {}_{Q}\!\vdash p \leadsto q))}$$

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

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

```
prog PLUM and ECHO

init   $(\forall p \in \mathbb{P} : (p = starter) \neq (\mathsf{idle}.p)) \wedge (\mathsf{father}.starter = starter) \wedge init_{\Pi}$

assign

$[]_{q \in \mathsf{neighs}.p}$   if $\mathsf{idle}.p \wedge \mathsf{mit}.q.p$
                                                                          (IDLE)
                then $\mathsf{receive}.p.q.\langle \mathbf{mes} \rangle \parallel \mathsf{father}.p := q \parallel \mathsf{idle}.p := \mathsf{false}$
$[]$

$[]_{q \in \mathsf{neighs}.p}$   if $\neg\ \mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge collecting_{\Pi}.p$
                                                                          (COL)
                then $\mathsf{receive}.p.q.\langle \mathbf{mes} \rangle$
$[]$

$[]_{q \in \mathsf{neighs}.p}$   if $\neg\ \mathsf{idle}.p \wedge can\_propagate.p.q \wedge propagating_{\Pi}.p$
                                                                          (PROP)
                then $\mathsf{send}.p.q.\langle \mathbf{mes} \rangle$
$[]$

if $finished\_collecting\_and\_propagating.p \wedge \neg reported\_to\_father.p$
                                                                          (DONE)
then $\mathsf{send}.p.(\mathsf{father}.p).\langle \mathbf{mes} \rangle$
```

Figure 9: The the local algorithm of process $p \in \mathbb{P}$ for $\Pi \in \{$PLUM, ECHO$\}$.

For this paper it is sufficient to give an abstract model of *asynchronous communication*, stating the functionality of the primitives (send and receive) and some additional operations (mit, nr_sent_to and nr_rec_from). $\mathsf{send}.p.q.m$, implements that a process p sends message m to q; $\mathsf{receive}.p.q.f.v$, 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; $\mathsf{mit}.p.q$, 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 nr_sent_to q, enables processes to check how many messages they have already sent to a neighbour q; similarly, p nr_rec_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:

$$rec\_from\_all\_neighs.p = \forall q \in \mathsf{neighs}.p\ : p\ \mathsf{nr\_rec\_from}\ q = 1 \tag{1}$$

$$sent\_to\_all\_non\_fathers.p = \forall q \in \mathsf{neighs}.p : (q \neq \mathsf{father}.p) \Rightarrow (p\ \mathsf{nr\_sent\_to}\ q = 1) \tag{2}$$

$$can\_propagate.p.q = (p\ \mathsf{nr\_sent\_to}\ q = 0) \wedge (q \neq \mathsf{father}.p) \tag{3}$$

$$finished\_collecting\_and\_propagating.p = rec\_from\_all\_neighs.p\ \wedge\ sent\_to\_all\_non\_fathers.p \tag{4}$$

$$reported\_to\_father.p = (p\ \mathsf{nr\_sent\_to}\ (\mathsf{father}.p) = 1) \tag{5}$$

$$sent\_to\_all\_neighs.p = \forall q \in \mathsf{neighs}.p\ : p\ \mathsf{nr\_sent\_to}\ q = 1 \tag{6}$$

```
prog TARRY

init  $(\forall p \in \mathbb{P} : (p = starter) \neq (\mathsf{idle}.p)) \wedge (\mathsf{father}.starter = starter)$
      $\wedge \boxed{\forall p \in \mathbb{P} : (p = starter) \neq (\neg \mathsf{le\_rec}.p)}$

assign

$[]_{q \in \mathsf{neighs}.p}$ if $\mathsf{idle}.p \wedge \mathsf{mit}.q.p$
                                                                        (IDLE)
          then $\mathsf{receive}.p.q.\langle \mathbf{mes} \rangle \parallel \mathsf{father}.p := q \parallel \mathsf{idle}.p := \mathsf{false}$
               $\parallel \boxed{\mathsf{le\_rec}.p := \mathsf{true}}$
$[]$

$[]_{q \in \mathsf{neighs}.p}$ if $\neg\, \mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \boxed{collecting_{\text{TARRY}}.p}$
                                                                        (COL)
          then $\mathsf{receive}.p.q.\langle \mathbf{mes} \rangle \parallel \boxed{\mathsf{le\_rec}.p := \mathsf{true}}$
$[]$

$[]_{q \in \mathsf{neighs}.p}$ if $\neg\, \mathsf{idle}.p \wedge can\_propagate.p.q \wedge \boxed{propagating_{\text{TARRY}}.p}$
                                                                        (PROP)
          then $\mathsf{send}.p.q.\langle \mathbf{mes} \rangle \parallel \boxed{\mathsf{le\_rec}.p := \mathsf{false}}$
$[]$

if $finished\_collecting\_and\_propagating.p \wedge \neg reported\_to\_father.p$
                                                                        (DONE)
then $\mathsf{send}.p.(\mathsf{father}.p).\langle \mathbf{mes} \rangle \parallel \boxed{\mathsf{le\_rec}.p := \mathsf{false}}$
```

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

$$done.p = rec\_from\_all\_neighs.p \wedge sent\_to\_all\_neighs.p \tag{7}$$

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:

$$propagating_{\text{PLUM}}.p = \neg\, sent\_to\_all\_non\_fathers.p \tag{8}$$

$$collecting_{\text{PLUM}}.p = \neg\, rec\_from\_all\_neighs.p \tag{9}$$

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:

$$propagating_{\text{ECHO}}.p = \neg\, sent\_to\_all\_non\_fathers.p \tag{10}$$

$$collecting_{\text{ECHO}}.p = \neg\, rec\_from\_all\_neighs.p \;\wedge\; \neg propagating_{\text{ECHO}}.p \tag{11}$$

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 $\mathsf{le\_rec}.p$ (i.e. $\underline{l}$ast $\underline{e}$vent was a $\underline{rec}$eive) has been introduced for every process p. The assignments ($\mathsf{le\_rec}.p := \mathsf{true}$) and ($\mathsf{le\_rec}.p := \mathsf{false}$) in the **then** clauses of (COL) and (PROP) respectively, guarantee that the the value of $\mathsf{le\_rec}.p$

```
prog DFS

init $\forall p \in \mathbb{P} : (p = starter) \neq (\mathsf{idle}.p) \wedge (\mathsf{father}.starter = starter)$
     $\wedge\ \forall p \in \mathbb{P} : (p = starter) \neq (\neg\mathsf{le\_rec}.p)$

assign

$[]_{q \in \mathsf{neighs}.p}$ if $\mathsf{idle}.p \wedge \mathsf{mit}.q.p$                                  (IDLE)
        then $\mathsf{receive}.p.q.\langle\mathbf{mes}\rangle \,\|\, \mathsf{father}.p{:=}q \,\|\, \mathsf{idle}.p{:=}\mathsf{false}$
             $\|\, \mathsf{le\_rec}.p{:=}\mathsf{true} \,\|\, \boxed{\mathsf{lp\_rec}.p{:=}q}$
$[]$

$[]_{q \in \mathsf{neighs}.p}$ if $\neg\, \mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \boxed{collecting_{\mathrm{DFS}}.p}$                   (COL)
        then $\mathsf{receive}.p.q.\langle\mathbf{mes}\rangle \,\|\, \mathsf{le\_rec}.p{:=}\mathsf{true} \,\|\, \boxed{\mathsf{lp\_rec}.p{:=}q}$
$[]$

$[]_{q \in \mathsf{neighs}.p}$ if $\neg\, \mathsf{idle}.p \wedge can\_propagate.p.q \wedge \boxed{propagating_{\mathrm{DFS}}.p} \wedge \boxed{q = \mathsf{lp\_rec}.p}$
                                                                  (PROP_LP_REC)
        then $\mathsf{send}.p.q.\langle\mathbf{mes}\rangle \,\|\, \mathsf{le\_rec}.p{:=}\mathsf{false}$
$[]$

$[]_{q \in \mathsf{neighs}.p}$ if $\neg\, \mathsf{idle}.p \wedge can\_propagate.p.q \wedge \boxed{propagating_{\mathrm{DFS}}.p} \wedge \boxed{\neg(can\_propagate.p.(\mathsf{lp\_rec}.p))}$
                                                                  (PROP_NOT_LP_REC)
        then $\mathsf{send}.p.q.\langle\mathbf{mes}\rangle \,\|\, \mathsf{le\_rec}.p{:=}\mathsf{false}$
$[]$

if $finished\_collecting\_and\_propagating.p \wedge \neg reported\_to\_father.p$       (DONE)
then $\mathsf{send}.p.(\mathsf{father}.p).\langle\mathbf{mes}\rangle \,\|\, \mathsf{le\_rec}.p{:=}\mathsf{false}$
```

Figure 11: The local algorithm of process $p \in \mathbb{P}$ of the DFS algorithm.

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

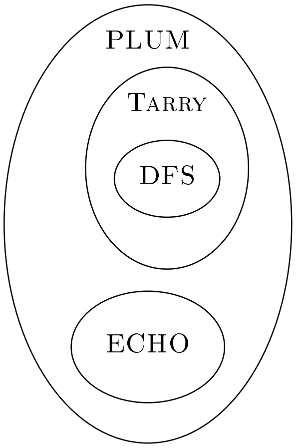
$$propagating_{\mathrm{TARRY}}.p = \neg\ sent\_to\_all\_non\_fathers.p\ \wedge\ (\mathsf{le\_rec}.p) \tag{12}$$

$$collecting_{\mathrm{TARRY}}.p = \neg\ rec\_from\_all\_neighs.p\ \wedge\ \neg(\mathsf{le\_rec}.p) \tag{13}$$

The characterisation of the *propagating* and *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 in its propagating phase (i.e. there are still non-father-neighbours to which p has not yet sent) whose last event was receiving a message from some neighbour q: *if* p can propagate a message back to q, i.e. q is not p's father, and p has not yet sent to q, *then* p has to send a message back to this process q, *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 $\mathsf{lp\_rec}.p$ ($\underline{\text{l}}$ast $\underline{\text{p}}$rocess of which p has $\underline{\text{rec}}$eived a message).

$$propagating_{\mathrm{DFS}}.p = propagating_{\mathrm{TARRY}}.p \tag{14}$$

$$collecting_{\mathrm{DFS}}.p = collecting_{\mathrm{TARRY}}.p \tag{15}$$



$\forall A \in \{\text{IDLE, COL, PROP, DONE}\},\ p \in \mathbb{P},\ q \in \mathsf{neighs}.p$

$$\begin{array}{l} \mathcal{R}\_\text{PLUM\_ECHO}.(A_{\text{PLUM}}.p.q).(A_{\text{ECHO}}.p.q) \\ \mathcal{R}\_\text{PLUM\_TARRY}.(A_{\text{PLUM}}.p.q).(A_{\text{TARRY}}.p.q) \\ \mathcal{R}\_\text{TARRY\_DFS}.(A_{\text{TARRY}}.p.q).(A_{\text{DFS}}.p.q) \end{array}$$

(a) (b)

Figure 12: (a) refinement relation on PLUM, ECHO, TARRY, and DFS. (b) bitotal relations

◄

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 $\text{PROP}_{\text{TARRY}}.p.q$ being related to both $\text{PROP\_LP\_REC}.p.q$ and $\text{PROP\_NOT\_LP\_REC}.p.q$. 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 *PLUM_refines_ECHO*

$$\forall J :: \text{PLUM} \sqsubseteq_{\mathcal{R}\_\text{PLUM\_ECHO},\ J} \text{ECHO}$$

Theorem 8.2 *PLUM_refines_Tarry*

$$\forall J :: \text{PLUM} \sqsubseteq_{\mathcal{R}\_\text{PLUM\_TARRY},\ J} \text{TARRY}$$

Theorem 8.3 *Tarry_refines_DFS*

$$\forall J :: \text{TARRY} \sqsubseteq_{\mathcal{R}\_\text{TARRY\_DFS},\ J} \text{DFS}$$

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 *HYLO_PLUM*

$$J_{\text{PLUM}}\ {}_{\text{PLUM}}\vdash \mathbf{ini}\text{PLUM} \rightsquigarrow \forall p : p \in \mathbb{P} : \mathit{done}.p$$

Where the invariant J_{PLUM} is defined below. The $\mathsf{M}.p.q$ variables model the communication channels between processes p and q.

Definition 8.5 PLUM'S INVARIANT *Invariant_DEF*

$J_{\text{PLUM}} =$

$$
\begin{array}{ll}
\quad \forall p \in \mathbb{P}, q \in \textsf{neighs}.p : \neg\textsf{idle}.p \wedge q = \textsf{father}.p \Rightarrow \neg\textsf{idle}.q & J^{1}_{\text{PLUM}} \\
\wedge\ \forall p \in \mathbb{P}, q \in \textsf{neighs}.p : p \ \textsf{nr\_sent\_to}\ q = 0 \ \vee\ p \ \textsf{nr\_sent\_to}\ q = 1 & J^{2}_{\text{PLUM}} \\
\wedge\ \forall p \in \mathbb{P}, q \in \textsf{neighs}.p : \textsf{idle}.p \Rightarrow p \ \textsf{nr\_rec\_from}\ q = 0 & J^{3}_{\text{PLUM}} \\
\wedge\ \forall p \in \mathbb{P}, q \in \textsf{neighs}.p : (q \ \textsf{nr\_rec\_from}\ p < p \ \textsf{nr\_sent\_to}\ q) = \textsf{mit}.p.q & J^{4}_{\text{PLUM}} \\
\wedge\ \textsf{father}.\mathit{starter} = \mathit{starter} \ \wedge\ \neg(\textsf{idle}.\mathit{starter}) & J^{6}_{\text{PLUM}} \\
\wedge\ \forall p \in \mathbb{P} :\ (p \neq \mathit{starter}) \wedge \neg(\textsf{idle}.p) \Rightarrow (\textsf{father}.p \in \textsf{neighs}.p) & J^{7}_{\text{PLUM}} \\
\wedge\ (\lambda s.\ \forall p \in \mathbb{P} :\ \neg s.(\textsf{idle}.p) \Rightarrow \exists k :\ \textsf{depth}.(s \circ \textsf{father}).\mathit{starter}.p.k) & J^{8}_{\text{PLUM}} \\
\wedge\ \forall p, q \in \mathbb{P} : \neg(\textsf{idle}.p) \wedge \neg \mathit{done}.p \wedge (q = \textsf{father}.p) \Rightarrow p \ \textsf{nr\_sent\_to}\ q = 0 & J^{9}_{\text{PLUM}} \\
\wedge\ \forall p, q \in \mathbb{P} : q \ \textsf{nr\_rec\_from}\ p \leq p \ \textsf{nr\_sent\_to}\ q & J^{10}_{\text{PLUM}} \\
\wedge\ \forall p, q \in \mathbb{P} : \textsf{M}.p.q = [] \vee (\exists x : \textsf{M}.p.q = [x]) & J^{11}_{\text{PLUM}} \\
\wedge\ \forall p, q \in \mathbb{P} : \textsf{idle}.p \Rightarrow p \ \textsf{nr\_sent\_to}\ q = 0 & J^{12}_{\text{PLUM}}
\end{array}
$$

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:

$$\forall J :: \text{PLUM} \sqsubseteq_{\mathcal{R}\_\text{PLUM\_ECHO},\ J} \text{ECHO}$$

For ECHO, the UNITY specification reads:

Theorem 8.6 *HYLO_ECHO*

$$J_{\text{PLUM}} \wedge J_{\text{ECHO}} \ {}_{\text{ECHO}}\!\vdash \textbf{ini}\text{ECHO} \rightsquigarrow \forall p : p \in \mathbb{P} : \mathit{done}.p$$

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

Using $\circlearrowleft$ PRESERVATION (Theorem 7.12_{15}), it is straightforward to derive that J_{PLUM} is also a stable predicate in ECHO.

Theorem 8.7 *STABLEe_Invariant_in_ECHO*

$${}_{\text{ECHO}}\!\vdash \circlearrowleft J_{\text{PLUM}}$$

For readability we introduce the notational convention that:

$\vdash$ and ${}_{\text{ECHO}}\!\vdash$ now abbreviate $J_{\text{PLUM}} \wedge J_{\text{ECHO}} \ {}_{\text{ECHO}}\!\vdash$

Termination of ECHO will be proved using the property preserving Theorem 7.10_{16}.

$${}_{\text{ECHO}}\!\vdash \textbf{ini}\text{ECHO} \rightsquigarrow \forall p : p \in \mathbb{P} : \mathit{done}.p$$

$\Leftarrow$(Theorem 7.10_{16}, 8.4_{23}, 8.1_{23})

$$
\begin{array}{l}
\quad \exists W :: (\textbf{w}\text{ECHO} = \textbf{w}\text{PLUM} \cup W) \wedge (J_{\text{PLUM}}\ \mathcal{C}\ W^{\mathsf{c}}) \wedge (\textbf{w}\text{PLUM} \subseteq W^{\mathsf{c}}) \\
\wedge \\
\quad \forall A_P\ A_E\ :\ A_P \in \textbf{a}\text{PLUM} \ \wedge\ A_P\ \mathcal{R}\_\text{PLUM\_ECHO}\ A_E\ :\ {}_{\text{ECHO}}\!\vdash \textsf{guard\_of}.A_P \rightarrowtail \textsf{guard\_of}.A_E \\
\wedge \\
\quad \forall A_P\ A_E\ :\ A_P \in \textbf{a}\text{PLUM} \ \wedge\ A_P\ \mathcal{R}\_\text{PLUM\_ECHO}\ A_E\ : \\
\qquad {}_{\text{ECHO}}\!\vdash (J_{\text{PLUM}} \wedge J_{\text{ECHO}} \wedge \textsf{guard\_of}.A_E)\ \textsf{unless}\ \neg(\textsf{guard\_of}.A_P)
\end{array}
$$

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 $\mathcal{R}\_\text{PLUM\_ECHO}$ (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 PLUM

- anti-reflexivity of unless (Theorem 3.17_5)
- reflexivity of $\rightarrowtail$ (Theorem $\text{A}.4_{31}$)
- the implicit assumption stating stability of $(J_{\text{PLUM}} \wedge J_{\text{ECHO}})$

the second and the third conjunct can, for arbitrary $p \in \mathbb{P}$ and $q \in \mathsf{neighs}.p$, be reduced to:

$$\begin{array}{l} {}_{\text{ECHO}}\vdash \ \mathsf{guard\_of}.\text{COL}_{\text{PLUM}}.p.q \rightarrowtail \mathsf{guard\_of}.\text{COL}_{\text{ECHO}}.p.q \quad \}\ \mathbf{reach-part} \\ \wedge \\ {}_{\text{ECHO}}\vdash \ J_{\text{PLUM}} \wedge J_{\text{ECHO}} \wedge \mathsf{guard\_of}.\text{COL}_{\text{ECHO}}.p.q \ \mathsf{unless} \ \neg\mathsf{guard\_of}.\text{COL}.p.q \quad \}\ \mathbf{unless-part} \end{array}$$

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

Rewriting **reach-part** the with the guards of the COL actions from PLUM and ECHO, the correctness of the ECHO algorithm comes down to proving that for an appropiate J_{ECHO} and arbitrary $p \in \mathbb{P}$ and $q \in \mathsf{neighs}.p$:

$$\begin{array}{ll} {}_{\text{ECHO}}\vdash & \neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg \mathit{rec\_from\_all\_neighs.p} \\ & \rightarrowtail \\ & \neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg \mathit{rec\_from\_all\_neighs.p} \wedge \mathit{sent\_to\_all\_non\_fathers.p} \end{array}$$

The proof of this **reach-part** can be found in [VS01], where it turns out that, again, J_{PLUM} is enough and hence J_{ECHO} can be substituted for true – meaning that the safety properties of PLUM and ECHO are the same. Although the proof of the **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 TARRY

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

$$\forall J :: \text{PLUM} \sqsubseteq_{\mathcal{R}\text{\_PLUM\_TARRY},\ J} \text{TARRY}$$

The UNITY specification reads:

Theorem 8.8 `HYLO_Tarry`

$$J_{\text{PLUM}} \wedge J_{\text{TARRY}} \ {}_{\text{TARRY}}\vdash \mathbf{ini}\text{TARRY} \rightsquigarrow \forall p : p \in \mathbb{P} : \mathit{done.p}$$

where invariant J_{TARRY} captures additional safety properties for TARRY.

Using ↻ PRESERVATION (Theorem 7.12_{15}), it is straightforward to derive that J_{PLUM} is a stable predicate in TARRY.

Theorem 8.9 `STABLEe_Invariant_in_Tarry`

$${}_{\text{TARRY}}\vdash \circlearrowright J_{\text{PLUM}}$$

For readability we introduce the notational convention that:

$\vdash$ and ${}_{\text{TARRY}}\vdash$ now abbreviate $J_{\text{PLUM}} \wedge J_{\text{TARRY}} \ {}_{\text{TARRY}}\vdash$

Termination of TARRY is proved using property preserving Theorem 7.9_{16}. The reason for using this theorem is that Theorem 7.10_{16} – which is easier and hence preferable – cannot be used since its application results in the following, not provable, proof obligation:

$${}_{\text{TARRY}}\vdash \ J_{\text{PLUM}} \wedge J_{\text{TARRY}} \wedge \mathsf{guard\_of}.(\text{PROP}_{\text{TARRY}}.p.q) \ \mathsf{unless} \ \neg\mathsf{guard\_of}.(\text{PROP}_{\text{PLUM}}.p.q)$$

The reason why this cannot be proved is because, during the execution of TARRY, it is possible that the guard of $\text{PROP}_{\text{TARRY}}.p.q$ is falsified while the guard of $\text{PROP}_{\text{PLUM}}.p.q$ still holds. Consequently, we cannot prove the 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 $\textsc{prop}_{\textsc{tarry}}.p.q$, while the guard of $\textsc{prop}_{\textsc{plum}}.p.q$ still holds, cannot happen infinitely often. So, since the least complicated property preservation theorem (7.10_{16}) cannot be used to derive termination of TARRY, we move on to the second least complicated one, i.e. 7.9_{16}:

$${}_{\textsc{tarry}}\vdash \ \mathbf{ini}\textsc{tarry} \leadsto \forall p : p \in \mathbb{P} : done.p$$

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

$$\begin{array}{l}
\quad \exists W :: (\mathbf{w}\textsc{tarry} = \mathbf{w}\textsc{plum} \cup W) \wedge (J_{\textsc{plum}}\ \mathcal{C}\ W^{c}) \wedge (\mathbf{w}\textsc{plum} \subseteq W^{c}) \\
\wedge \\
\left.\begin{array}{l}
\quad \forall A_P\ A_T\ :\ A_P \in \mathbf{a}\textsc{plum}\ \wedge\ A_P\ \mathcal{R}_{\textsc{-plum-tarry}}\ A_T\ : \\
\qquad {}_{\textsc{tarry}}\vdash \mathsf{guard\_of}.A_P \rightarrowtail \mathsf{guard\_of}.A_T
\end{array}\right\}\ \mathbf{reach-part} \\
\wedge \\
\left.\begin{array}{l}
\quad \exists M :: (M\ \mathcal{C}\ \mathbf{w}\textsc{tarry}) \\
\qquad \wedge \\
\qquad \forall k ::\ {}_{\textsc{tarry}}\vdash (J_{\textsc{plum}} \wedge J_{\textsc{tarry}} \wedge M = k)\ \mathsf{unless}\ (M \prec k) \\
\qquad \wedge \\
\qquad \forall k\ A_P\ A_T : A_P \in \mathbf{a}\textsc{plum} \wedge A_P\ \mathcal{R}_{\textsc{-plum-tarry}}\ A_T : \\
\qquad\quad {}_{\textsc{tarry}}\vdash \quad (J_{\textsc{plum}} \wedge J_{\textsc{tarry}} \wedge \mathsf{guard\_of}.A_T \wedge M = k) \\
\qquad\qquad\qquad \mathsf{unless} \\
\qquad\qquad\qquad (\neg(\mathsf{guard\_of}.A_P) \vee M \prec k)
\end{array}\right\}\ \mathbf{unless-part}
\end{array}$$

Since, $\mathsf{le\_rec}.p$ variables are superimposed on PLUM in order to obtain TARRY, the first conjunct is instantiated with the set $\{\mathsf{le\_rec}.p \mid p \in \mathbb{P}\}$. Proving that $J_{\textsc{plum}}$ is confined by the complement of this set is tedious but straightforward, since the variables $\mathsf{le\_rec}$ do not appear in it.

Verification of the **unless-part** involves the construction of a function over the variables of 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 [VS01] that the verification of the **reach-part** involves an application of $\rightarrowtail$ BOUNDED PROGRESS ($A.10_{31}$) 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 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 $\mathsf{nr\_sent\_to}$
- similarly, receiving a message is always accompanied by incrementing $\mathsf{nr\_rec\_from}$
- from $J_{\textsc{plum}}$ 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 *MAX_MAIL*

$\mathsf{MAX\_MAIL} = 2\ \times$ the amount of directed communication links in the network ($\mathbb{P}$, *starter*, 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 *TOTAL_NR_SENT*

$$\mathsf{TOTAL\_NR\_SENT}.s = \sum_{p \in \mathbb{P}} \sum_{q \in \mathsf{neighs}.p} s.(p\ \mathsf{nr\_sent\_to}\ q)$$

Definition 8.12 TOTAL NUMBER OF MESSAGES RECEIVED IN THE NETWORK *TOTAL_NR_REC*

$$\mathsf{TOTAL\_NR\_REC}.s = \sum_{p \in \mathbb{P}} \sum_{q \in \mathsf{neighs}.p} s.(p\ \mathsf{nr\_rec\_from}\ q)$$

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

Definition 8.13 NON-INCREASING FUNCTION OVER THE VARIABLES OF TARRY *Y_DEF*

$$Y.s = \mathsf{MAX\_MAIL} - (\mathsf{TOTAL\_NR\_SENT}.s + \mathsf{TOTAL\_NR\_REC}.s)$$

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

Theorem 8.14 *CONF_Y_Write_Vars_Tarry*

$$Y\ \mathcal{C}\ \mathbf{w}\mathrm{TARRY}$$

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 *A_DECR_Y*

For all processes $p \in \mathbb{P}$, $q \in \mathsf{neighs}.p$, and actions $A \in \{\mathrm{IDLE}_{\mathrm{TARRY}}, \mathrm{COL}_{\mathrm{TARRY}}, \mathrm{PROP}_{\mathrm{TARRY}}, \mathrm{DONE}_{\mathrm{TARRY}}\}$:

$$\forall k :: \frac{J_{\mathrm{PLUM}}.s \wedge A.p.q.s.t \wedge \mathsf{guard\_of}.(A.p.q).s \wedge (Y.s = k)}{Y.t < k}$$

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 *DECREASING_DECR_FUNCTION*

For arbitrary characterisations of J_{TARRY}:

$$\forall k :: {}_{\mathrm{TARRY}}\vdash (J_{\mathrm{PLUM}} \wedge J_{\mathrm{TARRY}} \wedge Y = k)\ \mathsf{unless}\ (Y < k)$$

Verification of the unless-part

Return to page 26 for the **unless-part**. Instantiating this proof obligation with Y, and rewriting with Theorems 8.14_{27} and 8.16_{27} results in the following proof obligation:

$$\begin{aligned}&\forall k\ A_P\ A_T : A_P \in \mathbf{a}\mathrm{PLUM} \wedge A_P\ \mathcal{R}_{\mathrm{-PLUM-TARRY}}\ A_T : \\ &\quad {}_{\mathrm{TARRY}}\vdash (J_{\mathrm{PLUM}} \wedge J_{\mathrm{TARRY}} \wedge \mathsf{guard\_of}.A_T \wedge Y = k)\ \mathsf{unless}\ (\neg(\mathsf{guard\_of}.A_P) \vee Y < k)\end{aligned}$$

Proving this is straightforward using $\mathcal{R}_{\mathrm{-PLUM-TARRY}}$ from Figure 12, and Lemma 8.15_{27}.

Verification of the reach-part

We shall now continue with the **reach-part**:

$$\forall A_P\ A_T\ :\ A_P \in \mathbf{a}\mathrm{PLUM}\ \wedge\ A_P\ \mathcal{R}_{\mathrm{-PLUM-TARRY}}\ A_T\ :\ {}_{\mathrm{TARRY}}\vdash \mathsf{guard\_of}.A_P \rightarrowtail \mathsf{guard\_of}.A_T$$

Subsequently, using:

- the characterisation of $\mathcal{R}_{\mathrm{-PLUM-TARRY}}$ (Figure 12)
- the fact that the guards of the $\mathrm{IDLE}_{\mathrm{TARRY}}$, and $\mathrm{DONE}_{\mathrm{TARRY}}$ actions are equal to those of PLUM
- reflexivity of $\rightarrowtail$ (Theorem $\mathrm{A}.4_{31}$)
- the implicit assumption stating stability of $(J_{\mathrm{PLUM}} \wedge J_{\mathrm{TARRY}})$

we reduce the **reach-part** for arbitrary $p \in \mathbb{P}$ and $q \in \mathsf{neighs}.p$, as follows:

$$\begin{array}{l} {}_{\textsc{tarry}}\vdash \mathsf{guard\_of}.(\textsc{col}_{\textsc{plum}}.p.q) \rightarrowtail \mathsf{guard\_of}.(\textsc{col}_{\textsc{tarry}}.p.q) \ \big\}\,\textbf{reach} - \textsc{col} - \textbf{part} \\ \wedge \\ {}_{\textsc{tarry}}\vdash \mathsf{guard\_of}.(\textsc{prop}_{\textsc{plum}}.p.q) \rightarrowtail \mathsf{guard\_of}.(\textsc{prop}_{\textsc{tarry}}.p.q) \ \big\}\,\textbf{reach} - \textsc{prop} - \textbf{part} \end{array}$$

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

$$\begin{array}{l} {}_{\textsc{tarry}}\vdash \mathsf{guard\_of}.(\textsc{col}_{\textsc{plum}}.p.q) \rightarrowtail \mathsf{guard\_of}.(\textsc{col}_{\textsc{plum}}.p.q) \wedge \neg \mathsf{le\_rec}.p \\ \wedge \\ {}_{\textsc{tarry}}\vdash \mathsf{guard\_of}.(\textsc{prop}_{\textsc{plum}}.p.q) \rightarrowtail \mathsf{guard\_of}.(\textsc{prop}_{\textsc{plum}}.p.q) \wedge \mathsf{le\_rec}.p \end{array}$$

Again, their proofs can be found in [VS01]. This time $J_{\textsc{plum}}$ does not suffice, because, evidently, we need to capture the additional safety behaviour about the alternating sending and receiving activities –by means of le_rec– in $J_{\textsc{tarry}}$. Although not trivial, these proofs and the construction of $J_{\textsc{tarry}}$ 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:

$$\forall J :: \text{Tarry} \sqsubseteq_{\mathcal{R}\_\textsc{Tarry\_dfs},\ J} \textsc{dfs}$$

The UNITY specification reads:

Theorem 8.17 *HYLO_DFS*

$$J_{\textsc{plum}} \wedge J_{\textsc{Tarry}} \wedge J_{\textsc{dfs}} \ {}_{\textsc{dfs}}\vdash \textbf{ini}\textsc{dfs} \rightsquigarrow \forall p : p \in \mathbb{P} : \mathit{done}.p$$

where invariant $J_{\textsc{dfs}}$ captures additional safety properties for DFS (if any). Using $\circlearrowright$PRESERVATION Theorem 7.12_{15}, it is straightforward to derive:

Theorem 8.18 *STABLEe_Invariant_in_DFS*

$${}_{\textsc{dfs}}\vdash \circlearrowright (J_{\textsc{plum}} \wedge J_{\textsc{Tarry}})$$

Again, for readability we introduce the notational convention that:

$\vdash$ and ${}_{\textsc{dfs}}\vdash$ now abbreviate $J_{\textsc{plum}} \wedge J_{\textsc{Tarry}} \wedge J_{\textsc{dfs}} \ {}_{\textsc{dfs}}\vdash$

Termination of DFS is proved using property preserving Theorem 7.7_{16}. The reasons for using this Theorem are twofold. First, since every PROP action in TARRY is bitotally related to two actions in DFS (namely PROP_LP_REC and PROP_NOT_LP_REC), we need to be able to pick one of those DFS PROP-actions when proving that the guards of TARRY's PROP-actions eventually implies the guards of related DFS's PROP-actions. Consequently, we cannot use preservation theorems 7.10_{16} or 7.9_{16}. The second reason for using 7.7_{16} is *not* because 7.8_{16} cannot be used, but because it reduces proof effort. As we have seen during TARRY's verification, Lemma 8.15_{27} was very useful when proving unless and ensures properties that involved Y. A similar lemma can easily be proved for the actions of DFS, and hence verification of unless and ensures properties involving Y in the context of DFS will be simple too.

Lemma 8.19 *A_DECR_Y*

For all $p \in \mathbb{P}$, $q \in \mathsf{neighs}.p$, and actions $A \in \{\textsc{idle}_{\textsc{dfs}}, \textsc{col}_{\textsc{dfs}}, \textsc{prop\_lp\_rec}, \textsc{prop\_not\_lp\_rec}, \textsc{done}_{\textsc{dfs}}\}$:

$$\forall k :: \frac{J_{\textsc{plum}}.s \wedge A.p.q.s.t \wedge \mathsf{guard\_of}.(A.p.q).s \wedge (Y.s = k)}{Y.t < k}$$

In resumen, to reduce proof effort we have decided to use 7.7_{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 $\textsc{dfs}$'s $\textsc{prop}$-actions go hand in hand with the falsification of the guards of $\textsc{Tarry}$'s $\textsc{prop}$-actions.

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

$$_{\textsc{dfs}}\vdash\ \mathbf{ini}\textsc{dfs} \leadsto \forall p : p \in \mathbb{P} : \mathit{done}.p$$

$\Leftarrow$(Theorem 7.7_{16}, 8.8_{25}, 8.3_{23})
For some well-founded relation $\prec$:

$$
\begin{array}{l}
\quad \exists W :: (\mathbf{w}\textsc{dfs} = \mathbf{w}\textsc{Tarry} \cup W) \wedge ((J_{\textsc{plum}} \wedge J_{\textsc{Tarry}})\ \mathcal{C}\ W^{\mathsf{c}}) \wedge (\mathbf{w}\textsc{Tarry} \subseteq W^{\mathsf{c}})\\
\wedge\\
\quad \forall A_D : A_D \in \mathbf{a}\textsc{dfs} \wedge (\exists A_T :: A_T \in \mathbf{a}\textsc{Tarry} \wedge (A_T\ \mathcal{R}_{\textsc{\_Tarry\_dfs}}\ A_D)) : (\mathsf{guard\_of}.A_D\ \mathcal{C}\ \mathbf{w}\textsc{dfs})\\
\wedge\\
\quad \left.\begin{array}{l}
\forall A_T\ A_D\ :\ A_T \in \mathbf{a}\textsc{Tarry}\\
\quad _{\textsc{dfs}}\vdash\ \ \mathsf{guard\_of}.A_T\\
\qquad\quad \rightarrowtail\\
\qquad\quad (\exists A_D :: (A_T\ \mathcal{R}_{\textsc{\_Tarry\_dfs}}\ A_D) \wedge \mathsf{guard\_of}.A_D)
\end{array}\right\}\ \textbf{reach} - \textbf{part}\\
\wedge\\
\quad \left.\begin{array}{l}
\exists M :: (M\ \mathcal{C}\ \mathbf{w}\textsc{dfs})\\
\quad \wedge\\
\quad \forall k ::\ _{\textsc{dfs}}\vdash\ (J_{\textsc{plum}} \wedge J_{\textsc{Tarry}} \wedge J_{\textsc{dfs}} \wedge M = k)\ \mathsf{unless}\ (M \prec k)\\
\quad \wedge\\
\quad \forall k\ A_T\ A_D : A_T \in \mathbf{a}\textsc{Tarry} \wedge A_T\ \mathcal{R}_{\textsc{\_Tarry\_dfs}}\ A_D :\\
\qquad _{\textsc{dfs}}\vdash\ \ (J_{\textsc{plum}} \wedge J_{\textsc{Tarry}} \wedge J_{\textsc{dfs}} \wedge \mathsf{guard\_of}.A_D \wedge M = k)\\
\qquad\qquad \mathsf{unless}\\
\qquad\qquad (\neg(\mathsf{guard\_of}.A_T) \vee M \prec k)
\end{array}\right\}\textbf{unless} - \textbf{part}
\end{array}
$$

Since, $\mathsf{lp\_rec}.p$ variables are superimposed on $\textsc{Tarry}$ in order to obtain $\textsc{dfs}$, the first conjunct is instantiated with the set $\{\mathsf{lp\_rec}.p \mid p \in \mathbb{P}\}$. Proving that $J_{\textsc{plum}}$ and $J_{\textsc{Tarry}}$ are confined by the complement of this set is tedious but straightforward, since the variables $\mathsf{le\_rec}$ do not appear in it. Similarly, proving that the guards of the actions in $\textsc{dfs}$ are confined by $\textsc{dfs}$'s write variables (i.e. the second conjunct) is not complicated.
The **unless-part** is now easy to prove by instantiating with Y (Definition 8.13_{27}):

- proving that Y is confined by the write variables of $\textsc{dfs}$ is easy using Theorem 8.14_{27} and monotonicity of confinement 3.2_3
- proving that Y is non-increasing in $\textsc{dfs}$, can be proved using unless $\textsc{preservation}$ Theorem 7.11_{15} and Theorem 8.16_{27}.
- proving that falsification of the guards of $\textsc{dfs}$'s actions go hand in hand with the falsification of the guards of related $\textsc{Tarry}$'s actions is easy using Lemma 8.19_{28}.

For the **reach-part**, the $\textsc{idle}$, $\textsc{col}$, and $\textsc{done}$ cases can be proved using $\rightarrowtail$ $\textsc{Introduction}$ ($\mathrm{A}.3_{31}$). As a consequence, we are left with the $\textsc{prop}$ case:

$$_{\textsc{dfs}}\vdash\ \mathsf{guard\_of}.(\textsc{prop}_{\textsc{Tarry}}.p.q) \rightarrowtail (\exists A_D :: (\textsc{prop}_{\textsc{Tarry}}.p.q\ \mathcal{R}_{\textsc{\_Tarry\_dfs}}\ A_D) \wedge \mathsf{guard\_of}.A_D)$$

This case states that: from a situation in which $\mathsf{guard\_of}.(\textsc{prop}.p.q)$ holds, we will eventually reach a situation in which either the guard of action $\textsc{prop\_lp\_rec}.p.q$ or $\textsc{prop\_not\_lp\_rec}.p.q$ holds. The proof can be found in [VS01], where it turns out that, $J_{\textsc{Tarry}}$ is enough and hence $J_{\textsc{dfs}}$ can be substituted for true – meaning that the safety properties of $\textsc{tarry}$ and $\textsc{dfs}$ 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

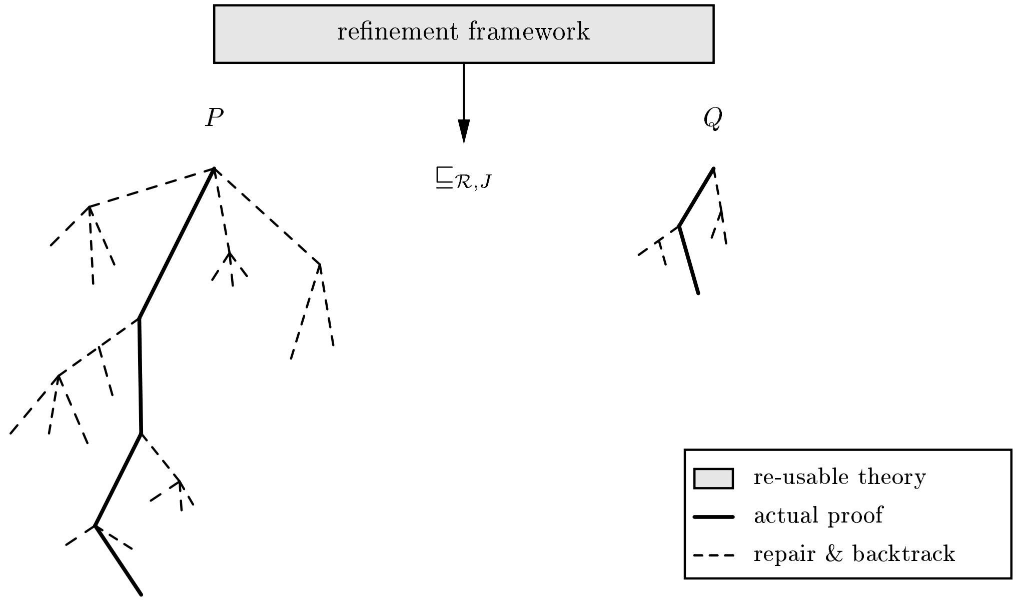


Figure 13: Reducing proof-effort and complexity.

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.

A Laws of $\rightarrowtail$

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

REACHe_IMP_STABLE
REACHe_IMP_CONF

$$P:\ \frac{J \vdash p \rightarrowtail q}{\circlearrowleft J \ \wedge\ p,q\,\mathcal{C}\,\mathbf{w}P}$$

Theorem A.2 $\rightarrowtail$ Substitution

REACHe_SUBST

$$P,J:\ \frac{p,s\,\mathcal{C}\,\mathbf{w}P \ \wedge\ [J \wedge p \Rightarrow q] \ \wedge\ (q \rightarrowtail r) \ \wedge\ [J \wedge r \Rightarrow s]}{p \rightarrowtail s}$$

Theorem A.3 $\rightarrowtail$ Introduction

REACHe_ENS_LIFT,REACHe_IMP_LIFT

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

Theorem A.4 $\rightarrowtail$ Reflexivity

REACHe_REFL

$$P,J:\ \frac{p\,\mathcal{C}\,\mathbf{w}P \ \wedge\ (\circlearrowleft J)}{p \rightarrowtail p}$$

Theorem A.5 $\rightarrowtail$ Transitivity

REACHe_TRANS

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

Theorem A.6 $\rightarrowtail$ Case distinction

REACHe_DISJ_CASES

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

Theorem A.7 $\rightarrowtail$ Cancellation

REACHe_CANCEL

$$P,J:\ \frac{q\,\mathcal{C}\,\mathbf{w}P \ \wedge\ (p \rightarrowtail q \vee r) \ \wedge\ (r \rightarrowtail s)}{p \rightarrowtail q \vee s}$$

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

REACHe_PSP

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

Theorem A.9 $\rightarrowtail$ Disjunction

REACHe_GEN_DISJe

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

Theorem A.10 $\rightarrowtail$ Bounded Progress

REACHe_WF_INDUCT

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

$$P,J:\ \frac{q\,\mathcal{C}\,\mathbf{w}P \ \wedge\ (\forall m \in W : p \wedge (M = m) \rightarrowtail (p \wedge (M \prec m)) \vee q)}{p \rightarrowtail q}$$

B Laws of $\rightsquigarrow$

Theorem B.1 Convergence Implies Progress

CONe_IMP_REACHe

$$P,J:\ \frac{p \rightsquigarrow q}{p \rightarrowtail q}$$

Theorem B.2 $\rightsquigarrow$ Substitution

CONe_SUBST

$$P,J:\ \frac{p,s\,\mathcal{C}\,\mathbf{w}P \wedge [J \wedge p \Rightarrow q] \wedge\ (q \rightsquigarrow r) \ \wedge [J \wedge r \Rightarrow s]}{p \rightsquigarrow s}$$

Theorem B.3 $\rightsquigarrow$ Introduction *CONe_ENSURES_LIFT, CONe_IMP_LIFT*

$$P, J : \ \frac{p, q\ \mathcal{C}\ \mathbf{w}P \ \wedge\ (\circlearrowright J) \ \wedge\ (\circlearrowright (J \wedge q)) \ \wedge\ ([J \wedge p \Rightarrow q] \ \vee\ (p \wedge J \ \mathsf{ensures}\ q))}{p \rightsquigarrow q}$$

Theorem B.4 $\rightsquigarrow$ Reflexivity *CONe_REFL*

$$P, J : \ \frac{p\ \mathcal{C}\ \mathbf{w}P \ \wedge\ (\circlearrowright J) \ \wedge\ (\circlearrowright (J \wedge p))}{p \rightsquigarrow p}$$

Theorem B.5 $\rightsquigarrow$ Transitivity *CONe_TRANS*

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

Theorem B.6 $\rightsquigarrow$ Case distinction *CONe_DISJ_CASES*

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

Theorem B.7 Accumulation *CON_SPIRAL*

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

Theorem B.8 $\rightsquigarrow$ Stable Strengthening *CONe_STAB_MONO_GEN*

$$P : \ \frac{q\ \mathcal{C}\ \mathbf{w}P \ \wedge\ (\circlearrowright (J_1 \wedge J_2)) \ \wedge\ J_1 \vdash p \rightsquigarrow q}{(J_1 \wedge J_2) \vdash p \rightsquigarrow q}$$

Theorem B.9 $\rightsquigarrow$ Stable Shift *CONe_STABLE_SHIFT*

$$P : \ \frac{p'\ \mathcal{C}\ \mathbf{w}P \wedge (\circlearrowright J) \ \wedge\ (J \wedge p' \vdash p \rightsquigarrow q)}{J \vdash p' \wedge p \rightsquigarrow q}$$

Theorem B.10 $\rightsquigarrow$ Disjunction *CONe_GEN_DISJ*

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

Theorem B.11 $\rightsquigarrow$ Conjunction *CONe_CONJ*
For all *non-empty* and *finite* sets W:

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

Theorem B.12 $\rightsquigarrow$ Bounded Progress *CONe_WF_INDUCT*
For a well-founded relation $\prec$ over some set A, and metric $M \in \texttt{State} \rightarrow A$:

$$P, J : \ \frac{(q \rightsquigarrow q) \ \wedge\ (\forall m \in A : p \wedge (M = m) \rightsquigarrow (p \wedge (M \prec m)) \vee q)}{p \rightsquigarrow q}$$

Theorem B.13 $\rightsquigarrow$ Iteration *Iterate_thm_CONe*
For arbitrary sets W,

$$P, J, L : \ \frac{\begin{array}{c}(\circlearrowright ((\forall x : x \in L : Q.x) \wedge J)) \wedge (\forall x : x \in L : Q.x\ \mathcal{C}\ \mathbf{w}P) \\ L \subseteq W \Rightarrow ((f.L) \subseteq W \ \wedge\ (\forall x : x \in L : Q.x) \rightsquigarrow (\forall x : x \in f.L : Q.x))\end{array}}{\forall n\ L : L \subseteq W \Rightarrow (\forall x : x \in L : Q.x) \rightsquigarrow (\forall x : x \in \mathsf{iterate}.n.f.L : Q.x)}$$

C Proofs of the refinement theorems

This appendix presents detailed proofs of the Theorems 7.11_{15} and 7.7_{16} stating the conditions under which unless and $\rightarrowtail$ 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_ref_AND_SUPERPOSE_WRITE_PRESERVES_UNLESSe*

$$\frac{\begin{array}{c} P \sqsubseteq_{\mathcal{R},J} Q \wedge \mathsf{Unity}.P \wedge \mathsf{Unity}.Q \wedge ({}_{Q}\!\vdash \circlearrowleft J_Q) \wedge (J_Q \Rightarrow J) \\ \exists W :: (\mathbf{w}Q = \mathbf{w}P \cup W) \wedge (p\ \mathcal{C}\ W^{\mathsf{c}}) \wedge (q\ \mathcal{C}\ W^{\mathsf{c}}) \end{array}}{{}_{P}\!\vdash p\ \mathsf{unless}\ q \Rightarrow {}_{Q}\!\vdash (J_Q \wedge p)\ \mathsf{unless}\ q}$$

proof of 7.11

Assume the following:

$\mathbf{A}_1$: $P \sqsubseteq_{\mathcal{R},J} Q$
$\mathbf{A}_2$: $\mathsf{Unity}.P \wedge \mathsf{Unity}.Q$
$\mathbf{A}_3$: $({}_{Q}\!\vdash \circlearrowleft J_Q) \wedge (J_Q \Rightarrow J)$
$\mathbf{A}_4$: $\mathbf{w}Q = \mathbf{w}P \cup W$
$\mathbf{A}_5$: $(p\ \mathcal{C}\ W^{\mathsf{c}}) \wedge (q\ \mathcal{C}\ W^{\mathsf{c}})$
$\mathbf{A}_6$: ${}_{P}\!\vdash p\ \mathsf{unless}\ q$

From $\mathbf{A}_5$ and the definition of confinement (3.1_3) we can infer:

$\mathbf{A}_7$: $\forall t, t' : (t =_{W^{\mathsf{c}}} t') \Rightarrow (p.t = p.t' \wedge q.t = q.t')$

From $\mathbf{A}_6$, the definitions of unless (3.15_5), and the definition of Hoare triples (3.14_5) we can infer:

$\mathbf{A}_8$: $\forall A_P : A_P \in \mathbf{a}P : \forall s, t : \mathsf{compile}.A_P.s.t \wedge p.s \wedge \neg(q.s) \Rightarrow (p.t \vee q.t)$

Now we have to prove the following:

${}_{Q}\!\vdash (J_Q \wedge p)\ \mathsf{unless}\ q$

= (Definitions of unless (3.15_5) and Hoare triples (3.14_5))

$\forall A_Q \in \mathbf{a}Q : \forall s, t : \mathsf{compile}.A_Q.s.t \wedge J_Q.s \wedge p.s \wedge \neg(q.s) \Rightarrow ((J_Q.t \wedge p.t) \vee q.t)$

Choose an arbitrary A_Q, and assume for arbitrary states s and t that:

$\mathbf{A}_9$: $A_Q \in \mathbf{a}Q$
$\mathbf{A}_{10}$: $\mathsf{compile}.A_Q.s.t$
$\mathbf{A}_{11}$: $J_Q.s \wedge p.s \wedge \neg(q.s)$

Now we have to prove that $((J_Q.t \wedge p.t) \vee q.t)$. From $\mathbf{A}_3$, we know that ${}_{Q}\!\vdash \circlearrowleft J_Q$, and consequently, using assumptions $\mathbf{A}_9$, $\mathbf{A}_{10}$, $\mathbf{A}_{11}$ and the definition of $\circlearrowleft$ (3.16_5 and 3.15_5) we can conclude that $J_Q.t$. Thus, we are left with the following proof obligation:

$p.t \vee q.t$

Case $\neg(\mathsf{guard\_of}.A_Q.s)$
In this case $\mathbf{A}_{10}$ implies $s = t$, and thus assumption $\mathbf{A}_{10}$ establishes the validity of $p.t \vee q.t$.
$\Box_{(\neg(\mathsf{guard\_of}.A_Q.s))}$

Case $\mathsf{guard\_of}.A_Q.s$
$\mathbf{A}_{12}$: $\mathsf{guard\_of}.A_Q.s$

From $\mathbf{A}_1$ it follows that:

$\mathbf{A}_{13}$: $\mathbf{a}Q = \mathbf{a}Q_1 \cup \mathbf{a}Q_2 \wedge \mathsf{bitotal}.\mathcal{R}.\mathbf{a}P.\mathbf{a}Q_1$

$\mathbf{A}_{14}$: $\forall A_P\ A_Q\ :\ A_P \in \mathbf{a}P\ \wedge\ A_P\ \mathcal{R}\ A_Q\ :\ A_P \sqsubseteq_{\mathbf{w}P,J} A_Q$
$\mathbf{A}_{15}$: $\forall A_Q : A_Q \in \mathbf{a}Q_2 : \mathsf{skip} \sqsubseteq_{\mathbf{w}P,J} A_Q$

Case $A_Q \in \mathbf{a}Q_1$

$\mathbf{A}_{16}$: $A_Q \in \mathbf{a}Q_1$

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

$\mathbf{A}_{17}$: $A_P \in \mathbf{a}P$
$\mathbf{A}_{18}$: $A_P\ \mathcal{R}\ A_Q$
$\mathbf{A}_{19}$: $A_P \sqsubseteq_{\mathbf{w}P,J} A_Q$

From $\mathbf{A}_{17}$ and the always-enabledness of actions in the universe `ACTION` (3.3_4) we know that there exists a state t' such that

$\mathbf{A}_{20}$: $\mathsf{compile}.A_P.s.t'$

and consequently from $\mathbf{A}_{19}$, $\mathbf{A}_{20}$, the definition of action refinement (7.1_{12}), and $\mathbf{A}_{10}$, $\mathbf{A}_{11}$, $\mathbf{A}_{12}$ and $\mathbf{A}_3$ we can infer that:

$\mathbf{A}_{21}$: $t =_{\mathbf{w}P} t'$

Moreover, using $\mathbf{A}_2$, $\mathbf{A}_{10}$, $\mathbf{A}_{20}$, and the definition of a well-formed UNITY program (3.13_5), and the definition of ignored variables (3.5_4) we can conclude:

$\mathbf{A}_{22}$: $s =_{\mathbf{w}P^c} t'$
$\mathbf{A}_{23}$: $s =_{\mathbf{w}Q^c} t$

From $\mathbf{A}_{21}$, $\mathbf{A}_{22}$ and $\mathbf{A}_{23}$ we can prove that $t =_{W^c} t'$, which with $\mathbf{A}_7$ gives:

$\mathbf{A}_{24}$: $p.t = p.t'\ \wedge\ q.t = q.t'$

From assumptions $\mathbf{A}_8$, $\mathbf{A}_{11}$, $\mathbf{A}_{17}$, $\mathbf{A}_{20}$ we can conclude that $p.t' \wedge q.t'$, and thus $\mathbf{A}_{24}$ establishes this case.

$\Box_{A_Q \in \mathbf{a}Q_1}$

Case $A_Q \in \mathbf{a}Q_2$

From $\mathbf{A}_9$, $\mathbf{A}_{15}$, the definition of action refinement (7.1_{12}), skip (3.4_4), $\mathbf{A}_3$, $\mathbf{A}_{10}$, $\mathbf{A}_{11}$ and $\mathbf{A}_{12}$ and we can conclude:

$\mathbf{A}_{25}$: $s =_{\mathbf{w}P} t$

Again, using $\mathbf{A}_2$ and the definition of a well-formed UNITY program (3.13_5), and the definition of ignored variables (3.5_4) we can conclude:

$\mathbf{A}_{26}$: $s =_{\mathbf{w}Q^c} t$

From this we can derive $s =_{W^c} t$, which with $\mathbf{A}_{11}$ gives the desired result.

$\Box_{A_Q \in \mathbf{a}Q_2}$

$\Box_{\mathsf{guard\_of}.A_Q.s}$

end of proof 7.11

C.2 Preservation of $\rightarrowtail$

Theorem 7.7 *P_ref_SUPERPOSE_AND_DECR_FUNC_PRSRVS_REACHe_GEN*
P_ref_SUPERPOSE_AND_DECR_FUNC_PRSRVS_CONe_GEN

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

$$\begin{array}{c}
P \sqsubseteq_{R,J} Q \wedge \mathsf{Unity}.P \wedge \mathsf{Unity}.Q \wedge (\ {}_{Q}{\vdash}\circlearrowright J_P \wedge J_Q) \wedge (J_P \wedge J_Q \Rightarrow J) \\
\exists W :: (\mathbf{w}Q = \mathbf{w}P \cup W) \wedge (J_P \ \mathcal{C}\ W^{\mathsf{c}}) \wedge (\mathbf{w}P \subseteq W^{\mathsf{c}}) \\
\forall A_Q : A_Q \in \mathbf{a}Q \wedge (\exists A_P :: (A_P \in \mathbf{a}P) \wedge (A_P \ \mathcal{R}\ A_Q)) : (\mathsf{guard\_of}.A_Q \ \mathcal{C}\ \mathbf{w}Q) \\
\forall A_P : A_P \in \mathbf{a}P : (J_P \wedge J_Q)\ {}_{Q}{\vdash}\ \mathsf{guard\_of}.A_P \rightarrowtail (\exists A_Q :: (A_P \ \mathcal{R}\ A_Q) \wedge \mathsf{guard\_of}.A_Q) \\
\exists M :: (M \ \mathcal{C}\ \mathbf{w}Q) \wedge (\forall k : k \in A :\ {}_{Q}{\vdash}\ (J_P \wedge J_Q \wedge M = k)\ \mathsf{unless}\ (M \prec k)) \\
\wedge\, \forall k\ A_P A_Q : k \in A \wedge A_P \in \mathbf{a}P \wedge A_P \ \mathcal{R}\ A_Q : \\
{}_{Q}{\vdash}\ (J_P \wedge J_Q \wedge \mathsf{guard\_of}.A_Q \wedge M = k)\ \mathsf{unless}\ (\neg(\mathsf{guard\_of}.A_P) \vee M \prec k) \\
\hline
((J_P\ {}_{P}{\vdash}\ p \rightarrowtail q) \Rightarrow (J_P \wedge J_Q\ {}_{Q}{\vdash}\ p \rightarrowtail q)) \ \wedge \ ((J_P\ {}_{P}{\vdash}\ p \rightsquigarrow q) \Rightarrow (J_P \wedge J_Q\ {}_{Q}{\vdash}\ p \rightsquigarrow q))
\end{array}$$

proof of 7.7 ($\rightarrowtail$-part)

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

A$_1$: $P \sqsubseteq_{\mathcal{R},J} Q$
A$_2$: $\mathsf{Unity}.P \ \wedge\ \mathsf{Unity}.Q$
A$_3$: ${}_{Q}{\vdash}\circlearrowright (J_P \wedge J_Q) \ \wedge\ (J_P \wedge J_Q \Rightarrow J)$
A$_4$: $\mathbf{w}Q = \mathbf{w}P \cup W \ \wedge\ J_P \ \mathcal{C}\ W^{\mathsf{c}} \ \wedge\ \mathbf{w}P \subseteq W^{\mathsf{c}}$
A$_5$: $\forall A_Q : A_Q \in \mathbf{a}Q \wedge (\exists A_P :: (A_P \in \mathbf{a}P) \wedge (A_P \ \mathcal{R}\ A_Q)) : (\mathsf{guard\_of}.A_Q \ \mathcal{C}\ \mathbf{w}Q)$
A$_6$: $\forall A_P : A_P \in \mathbf{a}P : (J_P \wedge J_Q)\ {}_{Q}{\vdash}\ \mathsf{guard\_of}.A_P \rightarrowtail (\exists A_Q :: (A_P \ \mathcal{R}\ A_Q) \wedge \mathsf{guard\_of}.A_Q)$
A$_7$: $M \ \mathcal{C}\ \mathbf{w}Q$
A$_8$: $\forall k ::\ {}_{Q}{\vdash}\ (J_P \wedge J_Q \wedge M = k)\ \mathsf{unless}\ (M \prec k)$
A$_9$: $\forall k\ A_P\ A_Q : A_P \in \mathbf{a}P \wedge A_P \ \mathcal{R}\ A_Q :$
$\qquad {}_{Q}{\vdash}\ (J_P \wedge J_Q \wedge \mathsf{guard\_of}.A_Q \wedge M = k)\ \mathsf{unless}\ (\neg(\mathsf{guard\_of}.A_P) \vee M \prec k)$

We have to prove that:

$$J_P\ {}_{P}{\vdash}\ p \rightarrowtail q \Rightarrow (J_P \wedge J_Q\ {}_{Q}{\vdash}\ p \rightarrowtail q)$$

For this we use the following theorem directly taken from [Pra95]; it states an induction principle for the $\rightarrowtail$ operator that corresponds to the latter's definition (3.19_5):

Theorem C.1 $\rightarrowtail$ INDUCTION *REACHe_INDUCT1*
For transitive and disjunctive R:

$$P, J : \ \frac{(\forall p, q :: (p \ \mathcal{C}\ (\mathbf{w}P) \ \wedge\ q \ \mathcal{C}\ (\mathbf{w}P) \ \wedge\ (\circlearrowright J) \wedge (J \wedge p\ \mathsf{ensures}\ q)) \Rightarrow R.p.q)}{(p \rightarrowtail q) \Rightarrow R.p.q}$$

take $R = (\lambda p\, q.\, J_P \wedge J_Q\ {}_{Q}{\vdash}\ p \rightarrowtail q)$. Since we already have $\rightarrowtail$-TRANSITIVITY, and $\rightarrowtail$-DISJUNCTION, we are left with the following proof-obligation:

$$\forall p\ q :: (p \ \mathcal{C}\ \mathbf{w}P \wedge q \ \mathcal{C}\ \mathbf{w}P \wedge \circlearrowright J_P \wedge (J_P \wedge p\ \mathsf{ensures}\ q)) \Rightarrow (J_P \wedge J_Q\ {}_{Q}{\vdash}\ p \rightarrowtail q)$$

Choose arbitrary p and q, and assume:

A$_{10}$: $p \ \mathcal{C}\ \mathbf{w}P \wedge q \ \mathcal{C}\ \mathbf{w}P$
A$_{11}$: ${}_{P}{\vdash}\circlearrowright J_P$
A$_{12}$: ${}_{P}{\vdash}\ J_P \wedge p\ \mathsf{ensures}\ q$

Theorem 3.2_3, stating confinement monotonicity, together with **A**$_4$ and **A**$_{10}$ gives:

A$_{13}$: $p \ \mathcal{C}\ \mathbf{w}Q \wedge q \ \mathcal{C}\ \mathbf{w}Q$

Rewriting $\mathbf{A}_{12}$ with the definition of $\mathsf{ensures}$ (3.18_5), we know that there exists an action A_P, such that:

$\mathbf{A}_{14}$: ${}_P\vdash J_P \wedge p \;\mathsf{unless}\; q$
$\mathbf{A}_{15}$: $A_P \in \mathbf{a}P$
$\mathbf{A}_{16}$: $\forall s\; t :: (J_P.s \wedge p.s \wedge \neg(q.s) \wedge \mathsf{compile}.A_P.s.t) \Rightarrow q.t$
Now we have to prove that:

$$J_P \wedge J_Q \;{}_Q\vdash p \rightarrowtail q$$

$\Leftarrow$ ($\rightarrowtail$ Bounded Progress $(\mathrm{A}.10_{31})$, $\mathbf{A}_7$, $\prec$ is well-founded)

$$\forall k ::\; J_P \wedge J_Q \;{}_Q\vdash p \wedge M{=}k \rightarrowtail (p \wedge M{\prec}k) \vee q$$

$\Leftarrow$ ($\rightarrowtail$ Case distinction $(\mathrm{A}.6_{31})$)

$$\left.\begin{array}{ll}\forall k ::\; J_P \wedge J_Q \;{}_Q\vdash & p \wedge M{=}k \wedge \neg(\mathsf{guard\_of}.A_P)\\ & \rightarrowtail\\ & (p \wedge M{\prec}k) \vee q\end{array}\right\}\textbf{false-guard-}A_P\textbf{-part}$$
$$\left.\begin{array}{ll}\forall k ::\; J_P \wedge J_Q \;{}_Q\vdash & p \wedge M{=}k \wedge \mathsf{guard\_of}.A_P\\ & \rightarrowtail\\ & (p \wedge M{\prec}k) \vee q\end{array}\right\}\textbf{true-guard-}A_P\textbf{-part}$$

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

lemma 1: $\forall s :: (J_P.s \wedge p.s \wedge \neg(\mathsf{guard\_of}.A_P.s)) \Rightarrow q.s$
Choose an arbitrary state s, and assume that:

$$J_P.s \;\wedge\; p.s \;\wedge\; \neg(\mathsf{guard\_of}.A_P.s)$$

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

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

which equals $q.s$.
$\square_{\textbf{lemma1}}$

false-guard-A_P-part
Theorem $\rightarrowtail$ Introduction $(\mathrm{A}.3_{31}$, implication-part), assumptions $\mathbf{A}_3$, $\mathbf{A}_7$ and $\mathbf{A}_{13}$, and **lemma 1** establish this case.
$\square_{\textbf{false-guard-}A_P\textbf{-part}}$

true-guard-A_P-part
For arbitrary k we have to prove that:

$$J_P \wedge J_Q \;{}_Q\vdash\; p \wedge M{=}k \wedge \mathsf{guard\_of}.A_P \rightarrowtail (p \wedge M{\prec}k) \vee q$$
$$= \text{(logics)}$$
$$J_P \wedge J_Q \;{}_Q\vdash\; p \wedge M{=}k \wedge \mathsf{guard\_of}.A_P \rightarrowtail ((p \wedge M{\prec}k) \vee q) \;\vee\; ((p \wedge M{\prec}k) \vee q)$$

$\Leftarrow$ ($\rightarrowtail$ Cancellation $(\mathrm{A}.7_{31})$, $\mathbf{A}_7$, $\mathbf{A}_{13}$)

$$\left.\begin{array}{l}J_P \wedge J_Q \;{}_Q\vdash p \wedge M{=}k \wedge \mathsf{guard\_of}.A_P\\ \qquad\qquad\rightarrowtail\\ \qquad\qquad(p \wedge M{\prec}k) \vee q \vee (p \wedge M{=}k \wedge (\exists A_Q :: A_P \mathcal{R} A_Q \wedge \mathsf{guard\_of}.A_Q))\end{array}\right\}\mathbf{C}_1$$
$$J_P \wedge J_Q \;{}_Q\vdash p \wedge M{=}k \wedge (\exists A_Q :: A_P \mathcal{R} A_Q \wedge \mathsf{guard\_of}.A_Q) \rightarrowtail (p \wedge M{\prec}k) \vee q \;\}\,\mathbf{C}_2$$

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

lemma 2: ${}_Q\vdash (J_P \wedge J_Q) \wedge p \;\mathsf{unless}\; q$

$= \text{(logics)}$

${}_Q\vdash J_Q \wedge (J_P \wedge p) \;\mathsf{unless}\; q$

$\Leftarrow$ (preservation of unless (7.11_{15}), $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{A}_3$, $\mathbf{A}_4$, $\mathbf{A}_{14}$)

$(J_P \wedge p)\, \mathcal{C}\, W^{\mathsf{c}} \;\wedge\; q\, \mathcal{C}\, W^{\mathsf{c}}$

$\Leftarrow$ (confinement of binary operators on first conjunct, and $\mathbf{A}_4$)

$p\, \mathcal{C}\, W^{\mathsf{c}} \;\wedge\; q\, \mathcal{C}\, W^{\mathsf{c}}$

$\Leftarrow$ (confinement monotonicity (3.2_3) on both conjuncts)

$p\, \mathcal{C}\, \mathbf{w}P \;\wedge\; q\, \mathcal{C}\, \mathbf{w}P \;\wedge\; \mathbf{w}P \subseteq W^{\mathsf{c}}$

Assumption $\mathbf{A}_{10}$ and $\mathbf{A}_4$ establishes this case.

$\Box_{\mathbf{lemma2}}$

proof of $\mathbf{C}_1$

$$\left.\begin{array}{ll} J_P \wedge J_Q \;{}_Q\vdash & p \wedge M{=}k \wedge \mathsf{guard\_of}.A_P \\ & \rightarrowtail \\ & (p \wedge M{\prec}k) \vee q \vee (p \wedge M{=}k \wedge (\exists A_Q :: A_P \mathcal{R} A_Q \wedge \mathsf{guard\_of}.A_Q)) \end{array}\right\}\mathbf{C}_1$$

$\Leftarrow$ ($\rightarrowtail$ PSP ($\mathrm{A}.8_{31}$), $\mathbf{A}_{13}$, **lemma 2**)

$$J_P \wedge J_Q \;{}_Q\vdash M{=}k \wedge \mathsf{guard\_of}.A_P \rightarrowtail M{\prec}k \vee (M{=}k \wedge (\exists A_Q :: A_P \mathcal{R} A_Q \wedge \mathsf{guard\_of}.A_Q))$$

$\Leftarrow$ ($\rightarrowtail$ PSP ($\mathrm{A}.8_{31}$), $\mathbf{A}_7$)

$J_P \wedge J_Q \;{}_Q\vdash \mathsf{guard\_of}.A_P \rightarrowtail (\exists A_Q :: A_P \mathcal{R} A_Q \wedge \mathsf{guard\_of}.A_Q)$

${}_Q\vdash M = k \wedge J_P \wedge J_Q \;\mathsf{unless}\; M \prec k$

Assumptions $\mathbf{A}_6$, $\mathbf{A}_{15}$ and $\mathbf{A}_8$ establish this.

$\Box_{\mathbf{C}_1}$

proof of $\mathbf{C}_2$

$$J_P \wedge J_Q \;{}_Q\vdash p \wedge M{=}k \wedge (\exists A_Q :: A_P \mathcal{R} A_Q \wedge \mathsf{guard\_of}.A_Q) \rightarrowtail (p \wedge M{\prec}k) \vee q \;\Big\}\,\mathbf{C}_2$$

$\Leftarrow$ ($\rightarrowtail$ SUBSTITUTION ($\mathrm{A}.2_{31}$), $\mathbf{A}_5$, $\mathbf{A}_7$, and $\mathbf{A}_{13}$)

$$J_P \wedge J_Q \;{}_Q\vdash \exists A_Q :: A_P \mathcal{R} A_Q \wedge p \wedge M{=}k \wedge \mathsf{guard\_of}.A_Q \rightarrowtail \exists A_Q :: A_P \mathcal{R} A_Q \wedge ((p \wedge M{\prec}k) \vee q)$$

$\Leftarrow$ ($\rightarrowtail$ DISJUNCTION ($\mathrm{A}.9_{31}$), $\mathbf{A}_5$, $\mathbf{A}_7$, and $\mathbf{A}_{13}$)

$$\forall A_Q : A_P \mathcal{R} A_Q : J_P \wedge J_Q \;{}_Q\vdash p \wedge M{=}k \wedge \mathsf{guard\_of}.A_Q \rightarrowtail (p \wedge M{\prec}k) \vee q$$

$\Leftarrow$ ($\rightarrowtail$ INTRODUCTION ($\mathrm{A}.3_{31}$), $\mathbf{A}_3$, $\mathbf{A}_5$,$\mathbf{A}_7$ and $\mathbf{A}_{13}$)

$$\forall A_Q : A_P \mathcal{R} A_Q : \;{}_Q\vdash J_P \wedge J_Q \wedge p \wedge M{=}k \wedge \mathsf{guard\_of}.A_Q \;\mathsf{ensures}\; (p \wedge M{\prec}k) \vee q$$

Assume:

$\mathbf{A}_{18}$: $A_P \mathcal{R} A_Q$

We are left with the proof obligations (definition of $\mathsf{ensures}$ (3.18_5))

$${}_Q\vdash J_P \wedge J_Q \wedge p \wedge M{=}k \wedge \mathsf{guard\_of}.A_Q \;\mathsf{unless}\; (p \wedge M{\prec}k) \vee q \;\Big\}\,\textbf{unless-part}$$

$$\left.\begin{array}{l} \exists A_Q : A_Q \in \mathbf{a}Q : \\ \quad \{J_P \wedge J_Q \wedge p \wedge M{=}k \wedge \mathsf{guard\_of}.A_Q \wedge \neg((p \wedge M{\prec}k) \vee q)\}a\{(p \wedge M{\prec}k) \vee q\} \end{array}\right\}\textbf{exists-part}$$

proof of the unless-part

Assume for arbitrary actions a, and states s and t:

$\mathbf{A}_{19}$: $a \in \mathbf{a}Q$
$\mathbf{A}_{20}$: $\mathsf{compile}.a.s.t$
$\mathbf{A}_{21}$: $J_P.s \wedge J_Q.s \wedge p.s \wedge (M.s = k) \wedge \mathsf{guard\_of}.A_Q.s$
$\mathbf{A}_{22}$: $\neg(M.s \prec k) \wedge \neg(q.s)$

We have to prove that:

$J_P.t \wedge J_Q.t \wedge p.t \wedge (M.t = k) \wedge \mathsf{guard\_of}.A_Q.t$
$\vee$
$p.t \wedge (M.t \prec k)$
$\vee$
$q.t$

From **lemma 2** and assumptions $\mathbf{A}_{19}$, $\mathbf{A}_{20}$, $\mathbf{A}_{21}$ and $\mathbf{A}_{22}$ we know that:

$\mathbf{A}_{23}$: $(J_P.t \wedge J_Q.t \wedge p.t) \vee q.t$

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

$\mathbf{A}_{24}$: $\neg(q.t)$

Then assumptions $\mathbf{A}_{23}$ and $\mathbf{A}_{24}$ leave us with the proof obligation:

$((M.t = k) \wedge \mathsf{guard\_of}.A_Q.t) \vee (M.t \prec k)$

From $\mathbf{A}_9$, $\mathbf{A}_{19}$, $\mathbf{A}_{20}$, $\mathbf{A}_{21}$, $\mathbf{A}_{22}$ and the definition of unless (3.15_5) we can deduce:

$\mathbf{A}_{26}$: $\mathsf{guard\_of}.A_P.s \Rightarrow (\mathsf{guard\_of}.A_Q.t \wedge (M.t = k)) \vee \neg(\mathsf{guard\_of}.A_P.t) \vee (M.t \prec k)$

From $\mathbf{A}_1$, $\mathbf{A}_3$, $\mathbf{A}_{18}$, $\mathbf{A}_{21}$, and the definition of action refinement (7.1_{12}), we can conclude $\mathsf{guard\_of}.A_P.s$, and hence assumption $\mathbf{A}_{26}$ gives:

$\mathbf{A}_{27}$: $(\mathsf{guard\_of}.A_Q.t \wedge (M.t = k)) \vee \neg(\mathsf{guard\_of}.A_P.t) \vee (M.t \prec k)$

Suppose $\mathsf{guard\_of}.A_P.t$ holds, then $\mathbf{A}_{27}$ establishes the proof. To reach a contradiction, we assume that:

$\mathbf{A}_{28}$: $\neg(\mathsf{guard\_of}.A_P.t)$

Now **lemma 1**, $\mathbf{A}_{28}$, $\mathbf{A}_{23}$, $\mathbf{A}_{24}$ imply $q.t$ which obviously contradicts $\mathbf{A}_{24}$.

$\Box_{\mathbf{unless-part}}$

proof of the exists-part:

The action that does the trick is A_Q (introduced in $\mathbf{A}_{18}$). From $\mathbf{A}_1$ we know that $\mathcal{R}$ is bitotal, and hence using $\mathbf{A}_{15}$, $\mathbf{A}_{18}$, and the definition of a bitotal relation we can infer that A_Q is indeed an action in $\mathbf{a}Q$. Assume for arbitrary states s and t that:

$\mathbf{A}_{29}$: $J_P.s \wedge J_Q.s \wedge p.s \wedge (M.s = k) \wedge \mathsf{guard\_of}.A_Q.s$
$\mathbf{A}_{30}$: $\neg(M.s \prec k) \wedge \neg(q.s)$
$\mathbf{A}_{31}$: $\mathsf{compile}.A_Q.s.t$

We are left with the proof obligation:

$(p.t \wedge (M.t \prec k)) \;\vee\; q.t$

From $\mathbf{A}_{15}$ and the always-enabledness of actions in the universe `ACTION` (3.3_4) we know that there exists a state t' such that

$\mathbf{A}_{32}$: $\mathsf{compile}.A_P.s.t'$

and consequently from $\mathbf{A}_1$, $\mathbf{A}_{18}$, the definition of action refinement (7.1_{12}), and assumptions $\mathbf{A}_3$, $\mathbf{A}_{29}$, $\mathbf{A}_{31}$, $\mathbf{A}_{32}$ we can infer that:

$\mathbf{A}_{33}$: $t =_{\mathbf{w}P} t'$

From assumption $\mathbf{A}_{16}$, $\mathbf{A}_{29}$, and $\mathbf{A}_{32}$ we can conclude that:

$\mathbf{A}_{34}$: $q.t'$

Finally from $\mathbf{A}_{33}$, $\mathbf{A}_{34}$, and $\mathbf{A}_{10}$ we can conclude $q.t$.

$\Box_{\textbf{exists}-\textbf{part}}$

$\Box\ \mathbf{C}_2$

$\Box_{\textbf{true-guard-}A_P\textbf{-part}}$

end of proof of Theorem 7.7 ($\rightarrowtail$-part)

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proceedings of the 3rd IEEE Symposium on Logic in Computer Science*, pages 165–175, 1988. Also available as DEC SRC Technical Report 29, 1988.

[Bac78] R.J.R. Back. *On the correctness of refinement steps in program development.* PhD thesis, University of Helsinki, 1978.

[Bac80] R.J.R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications.*, volume 131 of *Mathematical Centre Tracts.* Mathematical Centre, Amsterdam, the Netherlands, 1980.

[Bac81] R.J.R. Back. On Correct Refinement of Programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1981.

[Bac88] R.J.R. Back. A Calculus of Refinements for Program Derivations. *Acta Informatica*, 25:593–624, 1988.

[Bac89] R.J.R. Back. A Method for Refining Atomicity in Parallel Algorithms. In E. Odijk, M. Rem, and J.C. Syre, editors, *PARLE '89*, volume 366 of *LNCS*, pages 199–216. Springer-Verlag, 1989.

[Bac90] R.J.R. Back. Refinement Calculus, Part II: Parallel and Reactive Programs. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *LNCS*, pages 67–93. Springer-Verlag, 1990.

[Bac93] R.J.R. Back. Atomicity Refinement in a Refinement Calculus framework. Reports on computer science and mathematics Series A, No. 141, Åbo Akademi, 1993.

[BD77] R.M. Burstall and J. Darlington. Some transformations for developing recursive programs. *Journal of the ACM*, 24(1):44–676, 1977.

[BKS83] R.J.R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralized control. *2nd ACM SIGACT-SIGOPS Symposium on Distributed Computing*, pages 131–142, 1983.

[BKS84] R.J.R. Back and R. Kurki-Suonio. A case study in constructing distributed algorithms: distributed exchange sort. In *Proceedings of the Winter School on Theoretical Computer Science*, pages 1–33. Finnish Society of Information Processing Science, 1984.

[BKS88] R.J.R. Back and R. Kurki-Suonio. Distributed co-operation with action systems. *ACM Transactions on Programming languages and Systems*, 10(4):513–554, 1988.

[BKS98] M. Bonsangue, J.N. Kok, and K. Sere. An Approach to Object-Orientation in Action Systems. In *Proceedings of Mathematics of Program Construction (MPC'98)*, volume 1422 of *LNCS*, pages 68–95. Springer-Verlag, 1998.

[BS91] R.J.R. Back and K. Sere. Stepwise Refinement of Action Systems. *Structured Programming*, (12):17–30, 1991.

[BS96] R.J.R. Back and K. Sere. From Action Systems to Modular Systems. *Software – Concepts and Tools*, (17):26–39, 1996.

[BvW89] R.J.R. Back and J. von Wright. Command lattices, variable environments and data refinement. Reports on computer science and mathematics Series A, Åbo Akademi, 1989.

[BvW90] R.J.R. Back and J. von Wright. Refinement Calculus, Part I: Sequential Nondeterministic Programs. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *LNCS*, pages 42–66. Springer-Verlag, 1990.

[BvW94] R.J.R. Back and J. von Wright. Trace Refinement of Action Symstem. In *Proceedings of the 5th International Conference on Concurrency Theory (CONCUR'94)*, volume 836 of *LNCS*, pages 367–384. Springer-Verlag, 1994.

[BvW98] R.J.R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer-Verlag, 1998.

[BW96] M. Butler and M. Waldén. Distributed System Development in B. In *Proceedings of the First B Conference*, pages 155–168, 1996. Also available as Technical Report TUCS 1996, No. 53.

[BW98] M. Butler and M. Waldén. Parallel Programming with the B Method. In E. Sekerinski and K. Sere, editors, *Program Development by Refinement: Case Studies Using the B Method*, chapter 5, pages 183–195. Springer-Verlag, 1998.

[CM89] K.M. Chandy and J. Misra. *Parallel Program Design.* Addison-Wesley, Austin, Texas, May 1989.

[Dij74] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communication of the ACM*, 17(11):643–644, 1974.

[Dij76] E.W. Dijkstra. *A discipline of programming.* Prentice-Hall, 1976.

[Din97] Jürgen Dingel. Approximating UNITY. In *Proceedings of the 2nd International Conference on Synchronization Models and Languages (COORDINATION'97)*, volume 1282 of *LNCS*, pages 320–337, Berlin, Germany, September 1997. Springer Verlag.

[Ger75] S.L. Gerhart. Correctness preserving program transformations. In *Proceedings of the 2nd ACM Conference of Principles of Programming Languages*, pages 54–66, 1975.

[GKSU98] H.J.M. Goeman, J.N. Kok, K. Sere, and R.T. Udink. Coordination in the ImpUnity Framework. *Science of computer programming*, 31:313–334, 1998.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computers programs. *Commun. Ass. Comput. Mach.*, 12:576–583, 1969.

[Hoa72] C.A.R. Hoare. Proof of Correctness of Data Representation. *Acta Informatica*, (1):271–281, 1972.

[Inc95] Random House Inc. *Random House Webster's college dictionary.* 1995. ISBN 0-679-43886-6.

[Jac91] Jeremy Jacob. The varieties of refinement. In J. M. Morris and R. C. Shaw, editors, *Proceedings of the 4th Refinement Workshop*, pages 441–455. Springer-Verlag, 1991.

[Jon90] B. Jonsson. On Decomposing and Refining Specifications of Distributed Systems. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *LNCS*, pages 361–385. Springer-Verlag, 1990.

[Kor91] J. Kornerup. Refinement in UNITY. Technical Report TR-91-18, Department of Computer Sciences, University of Texas at Austin, 1991.

[Lam83] L. Lamport. Specifying Concurrent Program Modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[Lam89] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, 1989.

[Lam91] Leslie Lamport. The temporal logic of actions. Technical Report 29, DEC SRC, 1991.

[Lam96] Leslie Lamport. Refinement on state-based formalisms. Technical Report 01, DEC SRC, 1996.

[LS84] S.S. Lam and A. U. Shankar. Protocol Verification via Projections. *IEEE Transactions on Software Engineering*, SE-10(4):325–342, 1984.

[MG90] C. Morgan and P.H.B. Gardiner. Data Refinement by Calculation. *Acta Informatica*, (27):481–503, 1990.

[Mis90] J. Misra. More on strengthening the guard. *Notes on UNITY*, 19-90, 1990. http://www.cs.utexas.edu/users/psp/notesunity.html.

[Mor88] C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3), 1988.

[Mor89] J.M. Morris. Laws of Data Refinement. *Acta Informatica*, (26):287–308, 1989.

[Mor90] C. Morgan. *Programming from Specifications.* Prentice Hall, 1990.

[Pra95] Wishnu Prasetya. *Mechanically Supported Design of Self-stabilizing Algorithms.* PhD thesis, UU, Oct 1995.

[San90] Beverly A. Sanders. Stepwise Refinement of Mixed Specifications of Concurrent Programs. In M. Broy and C.B. Jones, editors, *Programming Concepts and Methods*, IFIP, pages 1–256. Elsevier Science Publishers B.V., 1990.

[San91] Beverly A. Sanders. Eliminating the Substitution Axiom from UNITY Logic. *Formal Aspects of Computing*, (3):189–205, 1991.

[Ser90] Kaisa Sere. *Stepwise Derivation of Parallel Algorithms.* PhD thesis, Åbo Akademi, may 1990.

[Sin89] A.K. Singh. On strengthening the guard. *Notes on UNITY*, 07-89, 1989. http://www.cs.utexas.edu/users/psp/notesunity.html.

[Sin91] A. K. Singh. Program refinement in fair transition systems. In *Conference on Parallel Architectures and Languages Europe (PARLE '91)*, volume 506 of *LNCS*, pages 128–147. Springer-Verlag, 1991.

[Sin93] Ambuj K. Singh. Program refinement in fair transitions systems. *Acta Informatica*, (30):503–535, 1993.

[SO89] A. K. Singh and R. Overbeek. Derivation of efficient parallel programs: An example from genetic sequence analysis. *International Journal of Parallel Programming*, 18(6):447–484, December 1989.

[SW94a] Kaise Sere and Marina Waldén. Verification of a distributed algorithm due to chu. In *Proceedings of the Thirteenth Annual Symposium on Principles of Distributed Computing (PODC'94)*, page 391, 1994. Full report available as Technical Report Åbo Akademi 1994, No. 156.

[SW94b] Kaise Sere and Marina Waldén. Verification of a distributed algorithm due to tajibnapis. In *Proceedings of the 5th Nordic Workshop on Program Correctness*, pages 161–172, 1994.

[SW96] Kaise Sere and Marina Waldén. Reverse Engineering Distributed Algorithms. *Software Maintenance: Research and Practice*, (8):117–144, 1996.

[SW97] Kaise Sere and Marina Waldén. Data Refinement of Remote Procedures. In *Proceedings of the International Symposium on Theoretical Aspects of Computer Software (TACS'97)*, volume 1281 of *LNCS*, pages 267–294, 1997.

[Udi95] Rob Udink. *Program Refinement in UNITY-like Environments.* PhD thesis, University of Utrecht, 1995.

[UK96] R.T. Udink and J.N. Kok. The RPC-Memory Specification Problem: UNITY + Refinement Calculus. In *Formal Systems Specification*, volume 1169 of *LNCS*, pages 521–540. 1996.

[Vos00] Tanja E.J. Vos. *UNITY in Diversity, A stratified approach to the verification of distributed algorithms.* PhD thesis, Utrecht University, 2000.

[VS01] T.E.J. Vos and S.D. Swierstra. Proving distributed hylomorphisms. Technical Report UU-CS-2001-40, University of Utrecht, 2001.

[vW92a] J. von Wright. Data Refinement and the simulation method. Reports on computer science and mathematics Series A, No. 137, Åbo Akademi, 1992.

[vW92b] J. von Wright. Data Refinement with stuttering. Reports on computer science and mathematics Series A, No. 138, Åbo Akademi, 1992.

[Wal96] Marina Waldén. Formal derivation of a distributed load balancing algorithm. In *Proceedings of the 7th Nordic Workshop on Programming Theory*, pages 508–527, 1996. Also technical report Åbo Akademi, Reports on computer science and mathematics, Series A, No. 172.

[Wal98a] Marina Waldén. Distributed load balancing. In E. Sekerinski and K. Sere, editors, *Program Development by Refinement: Case Studies Using the B Method*, chapter 7, pages 255–300. Springer-Verlag, 1998.

[Wal98b] Marina Waldén. Layering Distributed Algorithms within the B-Method. In *Proceedings of the 2nd International B Conference*, volume 1393 of *LNCS*, pages 243–260. Springer-Verlag, 1998.

[Wir71] N. Wirth. Program development by stepwise refinement. *CACM*, (14):221–227, 1971.

[WS96] Marina Waldén and Kaise Sere. Refining Action Systems within B-Tool. In *Proceedings of Formal Methods Europe (FME'96)*, volume 1051 of *LNCS*, pages 85–104, 1996. Also available as Technical Report TUCS 1996, No. 31.

[ZGK90] S. Zhou, R. Gerth, and R. Kuiper. Transformation Preserving Properties and Properties Preserved by Transformations in Fair Transition Systems. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *LNCS*, pages 353–367. Springer-Verlag, 1990.

Index