On Development of Web-based Software*
A Divertimento of Ideas and Suggestions

Dines Bjørner
Fredsvej 11, DK-2840 Holte, Denmark
bjorner@gmail.com

Begun August 4, 2010; compiled October 8, 2010: 19:21

cover

Abstract

This divertimento – on the occasion of the 70th anniversary of Prof., Dr NNN[1] – sketches some observations on relationships between window- and Web-based graphic user interfaces (GUI) and underlying Internet-based Linda [16] and JavaSpaces-like spaces [15], that is, shared network-accessible repositories for arbitrary data structures. (The Preface (next) and Sect. 1.4 on page 15 presents the structure of the paper.)

• • •

- I started working on this technical document early August 2010 with two points in mind: (i) trying to understand the documents [10, 11, 29, 30] purportedly defining XVSM, and (ii) trying to see whether currently fashionable issues of *interactive media* could be understood as precise artifacts of computing. I think I am succeeding on the latter point.
- The most serious "omission" in this technical note is the absence of any serious handling of proof obligations. A beginning is laid: the considerable number of axioms point in that direction. I intend, once all the "formulas" have been given a first rough sketch, to go through the whole document, with a fine comb, identifying interesting invariants (i.e., "consistent state issues"), etc.
- As of October 8, 2010 I have not printed this document and since I – unfortunately – do not use RSL-related tools I cannot guarantee that all function types are being respected, nor that all occurrences of function and type names have been properly defined.
- An index is provided at the end of this report (Pages 170–186). It should help the untrained reader to find way around the formulas. No formula is defined using formal terms that have not already been defined.

*This document constitutes a report on "work in progress". A paper for the NNN Festschrift will be a drastically reduced version of this report.

[1]It was hoped, when this technical work was first undertaken, that I could "spin" a 15 pages LNCS 'Festschrift' paper with a background in the technical woek. But that 'hope' is dwindling!

cover

Preface

cover

- This is a mere technical report.
  - The author set out to give an as complete account of a possible relationship between conventional window-based graphical user interfaces and data spaces such as Linda [16] and JavaSpaces-like spaces [15], that is, shared network-accessible repositories for arbitrary data structures.
  - The urge to do so was an "on and off" study that the author made in the period mid April to late July 2010 of the XVSM [10, 11, 29, 30].
  - This study began during the author's lectures at the Technical University of Vienna, Austria — a most pleasant stay for which he thanks Prof., Dr Jens Knoop profusely.
  - The study first lead to an attempt to reformulate XVSM in the style that the author best likes. The reformulation was based on [10].
  - Around August 1, 2010, the author then decided to halt further work on the XVSM. The state of that formalisation is found at: `http://www2.imm.dtu.dk/~db/xvsm-p.pdf`.
- It is fun to work out the "speculated" relationship.
  - I have always wanted, since the mid 1980s, to formalise so-called human-computer interfaces (CHI, [13, 34, 3, 39, 40, 28, 19]). See also [6, Examples 19.27–19.28, Pages 435–442] of my three volume "grand d'oeuvre" !
  - I have yet to see the published literature hint at or focus on the relation: that the data structure of a window reflects data structures of data bases — or vice versa.
  - It is, to me, obvious that this must be the case.
  - How else are we "thinking", conceptualising.
  - Other than visualising the concepts.
- In this report, we develop, in Sect. 2,
  - the notions of window and shared data spaces painstakingly from basic concepts:
    - atomic values and types, via
    - curtain values and types, to
    - window values and types,
  - and from there to
    - window frames in Sect. 3 and
    - domain frames in Sect. 4.

- We first develop these information concepts, in the order listed above before we focus
  - first on domain frame operations, in Sect. 5,
  - then on window frame operations, in Sect. 6.
- We wrap all of the above "algebras" up by presenting, in Sects. 7–9,
  - In Section 7:
    - the definition of a distributed system of
    - one domain process and
    - zero, one or more window processes.
  - Whereas Sects. 2–6 are kept in a pure, functional style RSL [17, 18, 4], Sect. 7 extends this style with RSL's CSP [21].
  - Section 8: A simple (read: idealised, naïve) coordinated transaction processing system.
  - Section 9:
    - A less simple roll-back/roll-forward system
    - extending that of Sect. 8
    - to handle failures.

Contents

intro

1 Introduction

intro

1.1 Background

1.2 Intuition

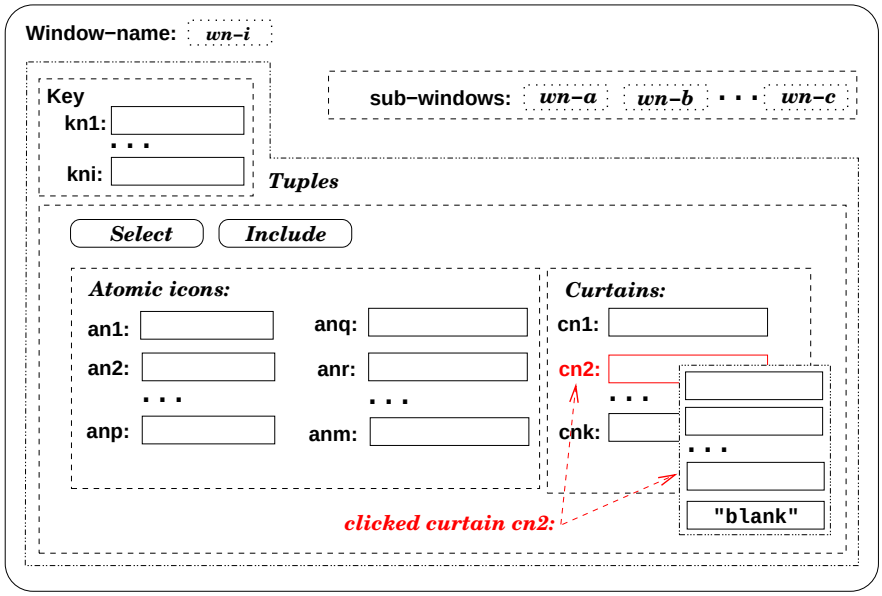


Figure 1: Schematic windows

1.2.1 Window States and Windows

Figure 1 shows a schematic snapshot of a *window*. The most recently, *"a-top-of-a-window-cactus-stack" window* – (upper-left-corner) is `named wn-i`. The window (named) `wn-i` shows *key fields* named `kn1` and `kn2`. These *key field names*, as we shall see later, are *atomic icon names* of that window. We have left the "boxed" key fields open. They are supposed to contain *key field values*. These values will, initially, be identical to those of the "matching" atomic icon fields.

1.2.2 Fields of Icons

Figure 1 shows three kinds of *fields*, i.e., icons: *atomic icons named* `ana, anb, ..., anm` in window `wn-i`; *curtain icons* named `cn1, cn2, ...` and `cnk` in the window named `wn-i`. The intuition about windows and icons, in general, is that they shall serve as a medium for information display, for initial data

input to general, space-oriented, possibly globally dispersed storage and for the (occasional) update of such stored data.

1.2.3 Atomic Icons

In the following we assume that data has already been stored in some global, say space-oriented storage. The intuition about atomic icons is the following: Atomic icon names hint at (or, not shown in Fig. 1 on the preceding page, directly embody) a description of the *type of the atomic data* of the atomic icon field. The atomic icon field either already contains some non-`"nil"` atomic data value, or contains such a `"nil"` value. The idea is that non-`"nil"` data informs the user, whereas `"nil"` data optionally "invites" the user to furnish (i.e., to *write*) a suitably typed atomic value. *Atomic values* can either be integers, natural numbers, (finitely expressible) rational numbers, Booleans (`"true", "false"`, or `"yes", "no"`, or "similar"), or texts: for example `"Dines Bjørner"`, `"4 October 1937"`, `"married"`, etc. The system to be designed in this report suggests that the user "signals" an intent to write into an atomic icon value field by *clicking* the atomic icon name; this enables the user to "type" (or otherwise) a representation of the value into the field, either overwriting a `"nil"` or whatever value was "already" posted in that field.

1.2.4 Curtain Icons

You will note that window `wn-i` (Fig. 1 on the facing page) shows that curtain name, `cn2` has been *"clicked"* and thus its curtain is *opened*. The open `cn2` shows an indefinite number of (ordered) atomic icon valued fields. The last field of a curtain value is always set to `‘‘blank’’`. As we shall see, curtain fields can be overwritten and new field values appended to the "end" – where there was a `‘‘blank’’` field value – resulting in a curtain list one longer than before overwriting the`‘‘blank’’`. The intuition about curtains is the following: A curtain is (to contain) a list of atomic values. These are to reflect sets or lists of common, related data (i.e., information). These lists or sets are of indefinite size, from empty, with just a single `"blank"` field, to some length or cardinality, as exemplified by `cn2`. As for atomic icons, curtain data can be initially input or viewed.

1.2.5 Windows and Window Icons

The intuition about windows is the following.

Windows represent pragmatically chosen complexes of information and data, either structured "flatly", in a set of atomic icons, or simply structured, i a set of curtains; or more hierarchically structured is sets of windows "embedded" within windows. The intuition about window icons is the following: A window icon can be clicked allowing an "underlying" window to open. The fields of that window can now be viewed, instantiated or updated. The intuition about keys is the following. If a window has no keys then it means that that window's field

values together represent the only *window value* for that named window. If a window has a key with one or more atomic icons, it means that that window's field values together represent one of a set of *window value* for that named window, namely a window value that is indexed by the key value. We say that if a window has a non-empty key (not shown in Fig. 1 on page 10) then it represents a relation (over window values). Figure 1 on page 10 does not hint at this relation. Figure 5 on page 23 does hint at this relation.

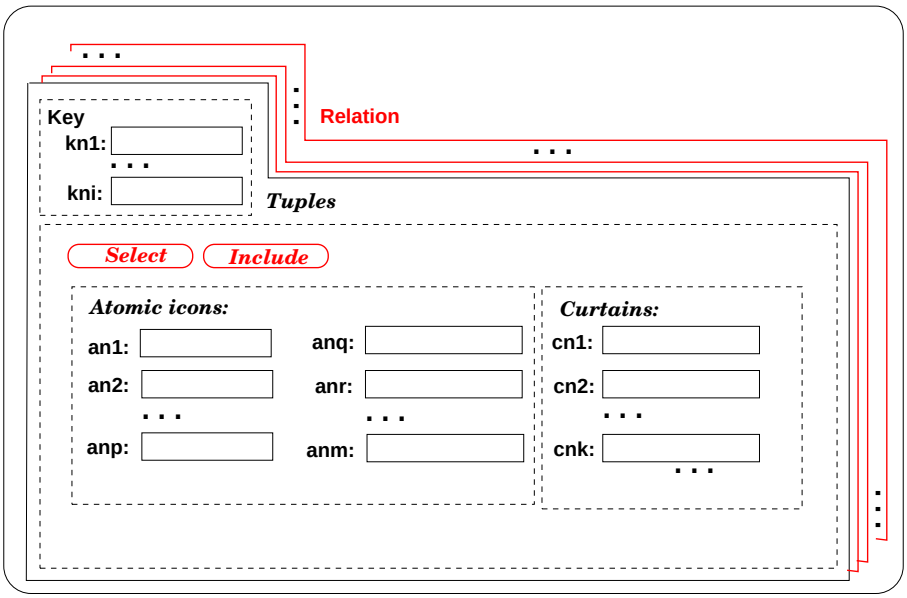


Figure 2: A window relation

You may think of the relation for a given, non-empty key window, as a set of tuples of icon and curtain values with a primary key being that of the named key fields. Once a window name is "clicked", as for window `wn-c`, then the following intermediate sequence takes place: First the dashed part of `wn-c` appears on the screen. Its key fields are left blank. For the user to fill in these fields amounts to the user selecting the tuple among the relation that matches this key. And the entire window, `window-n`, with its remaining fields (atomic icons, curtains and possibly further, embedded windows, are shown.

1.2.6 Special "Buttons"

Figure 3 on the facing page shows some additional buttons. A `"read"` button, when clicked, shall lead to an update of the window relation for that window with the field values of atomic icons and curtains. A `"write"` button, when clicked, writes that window do a domain frame. A `"take"` button, when clicked, deletes a window of that name from a domain frame while maintaining the current

window. A "close" button, when clicked, closes that window (in the window frame). We have not bothered to show these "buttons" in Figs. 1 on page 10 and 2 on the facing page.

1.2.7 Sub-windows

Figure 3 shows a "state" of the window first shown in Fig. 1 on page 10. In Fig. 1 on page 10 a number of sub-window names were listed: *wn-a*, *wn-b* and *wn-c*. In Fig. 3 sub-window name *wn-b* appears to have been *"clicked"*. As a result a window of that name has been opened. But, "to begin with", only with the hey fields displayed. The window 'user' is then expected to fill in zero, one or more of (as here, zero or one of) the key-field values. When that has been done the window frame will respond by selecting a suitable tuple from the chosen window relation and display this and the rest of the (constant) window fields.

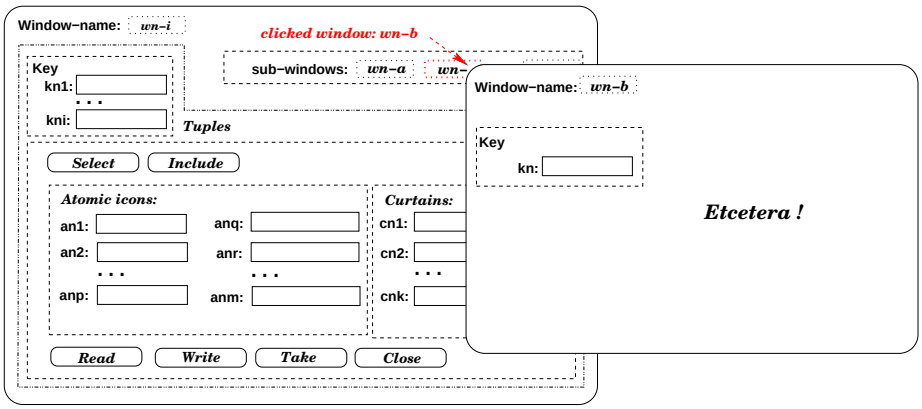


Figure 3: A sub-window

1.3 Trees, Stacks and Cacti [NIIST, US Govt.]

The successive opening and closing of windows result in an underlying window frame system "grafting" and "pruning" a cactus stack of windows. It is like a tree, but each branch of the tree, that is, a window, allows being operated upon during its "lifetime".

1.3.1 Trees

http://www.itl.nist.gov/div897/sqg/dads/HTML/tree.html

Definition: A data structure accessed beginning at the root node. Each node is either a leaf or an internal node. An internal node has one or more child

nodes and is called the parent of its child nodes. All children of the same node are siblings. Contrary to a physical tree, the root is usually depicted at the top of the structure, and the leaves are depicted at the bottom.

Formal Definition: A tree is either

```
* empty (no nodes), or
* a root and zero or more subtrees.
```

1.3.2 Stacks

http://www.itl.nist.gov/div897/sqg/dads/HTML/stack.html

Definition: A collection of items in which only the most recently added item may be removed. The latest added item is at the top. Basic operations are push and pop. Often top and isEmpty are available, too. Also known as "last-in, first-out" or LIFO.

Formal Definition: The operations new(), push(v, S), top(S), and pop(S) may be defined with axiomatic semantics as follows.

```
* new() returns a stack
* popoff(push(v, S)) = S
* top(push(v, S)) = v
```

where S is a stack and v is a value. The pop operation is a combination of top, to return the top value, and popoff, to remove the top value.

The predicate isEmpty(S) may be defined with the following additional axioms.

```
* isEmpty(new()) = true
* isEmpty(push(v, S)) = false
```

1.3.3 Cacti

http://www.itl.nist.gov/div897/sqg/dads/HTML/cactusstack.html

Definition: A variant of stack in which one other cactus stack may be attached to the top. An attached stack is called a branch. When a branch becomes empty, it is removed. Pop is not allowed if there is a branch. A branch

is only accessible through the original reference; it is not accessible through the stack.

Formal Definition: The operations new to this variant of stack, branch(S, T) and notch(v), may be defined with axiomatic semantics as follows.

```
*  top(branch(S, T)) = top(S)
*  notch(new()) = false
*  notch(push(v, S)) = false
*  notch(branch(S, T)) = true
```

Also known as saguaro stack.

1.4 Structure of Paper

We first (Sect. 2) develop (analyse and construct, narrate and formalise) a notion of windows as composed from various forms of icons: atomic, scroll down curtains and sub-windows.

Windows denote data structures where what you see on a screen is but part of that data structure. With a window is associated the property that zero, one or more of its atomic icons form a key, and, therefore, what you see on the screen (apart from references to sub-windows) is just a tuple of a relation whose "other" tuples are part of the window data structure. That part is not displayed. The screen tuple can be replaced by other tuples from the relation by changing the key values of the visible tuple. And the relation can be 'updated' by inserting the current key and tuple into the relation.

Then (Sect. 3) we develop (analyse and construct, narrate and formalise) a notion of window frames – complexes of windows on a screen, for example.

Following that (Sect. 4) we develop (analyse and construct, narrate and formalise) a notion of domain frames. That notion shall serve as the global (`Linda`[2], `JavaSpaces`[3] and `XVSM`[4]-like) storage for possibly coordinated, but till now un-coordinated users, where users are represented by window frames.

Thus window frames *"get"* windows from and *"put* windows back into a global domain frame.

A number of operations on, first domain frames, then window frames are then defined (Sects. 5–6).

The (Sect. 7) we describe (narrate and formalise) a simple transaction processing system in which one domain frame and n window frames (i.e., users) cooperate – when window frames get and put windows.

Finally (Sect. 8)[5] we describe (narrate and formalise) a simple coordinated transaction processing system (commits, etc.).[6]

[2] Linda: [16]
[3] JavaSpaces: [15]
[4] XVSM: [10, 11, 29, 30]
[5] A section (Sect. 11) on a graphic user interface design system is contemplated and will appear some day!
[6] But as of October 8, 2010, this work has yet to be done.

2 Windows

windows

windows

Windows are common to the user graphic computing interface, called window frames and covered in Sect. 3 and to the global, possibly world-wide distributed data spaces covered in Sect. 4.

2.1 A Review

Figure 4 schematizes a "generic" window. It has three sub-parts: a window name part, shown in upper left corner of Fig. 4; a *tuples* part, shown covering most of the window of Fig. 4 and a window-names part, upper right part of Fig. 4. The *tuples* part has two sub-parts: a *key* part consisting of zero, one or

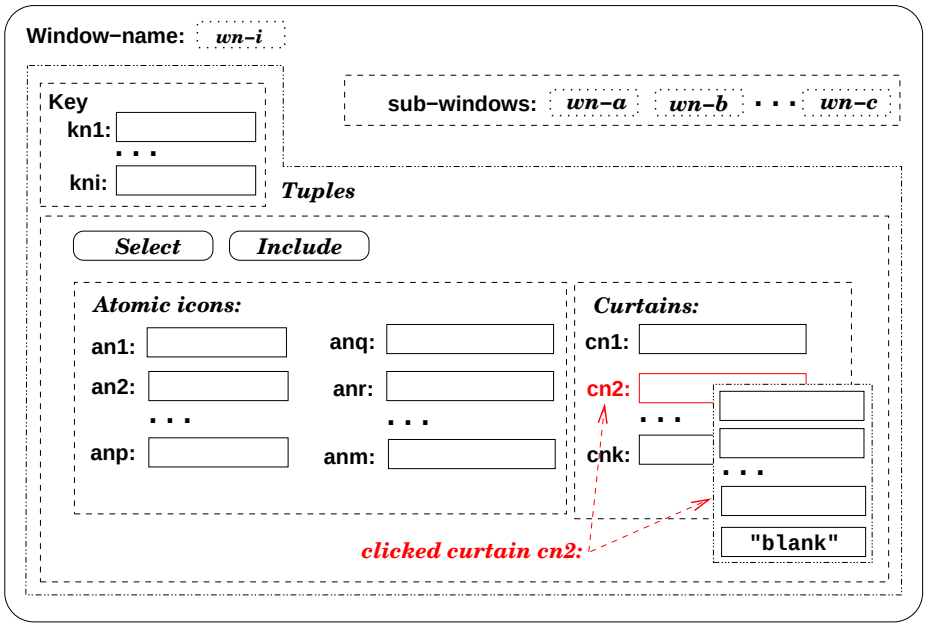


Figure 4: A Schematic Window

more distinctly named atomic icons, and a (*"remaining tuples"*) part consisting of zero, one or more atomic icons and zero, one or more curtains. All these are distinctly named. The window-names part consists of zero, one or more window-names.

2.2 Atomic Values and Atomic Types

2.2.1 Atomic Values

1. An atomic value is either an integer, a finitely representable rational, a Boolean, a text, or a nil "value".[7]

1. AVAL == mkIV(**Int**)|mkRV(**Rat**)|mkBV(**Bool**)|mkT(**Text**)|″nil″

2.2.2 Atomic Types

Values have types and type descriptors designate sets of values of the same type.

2. There are integer, rational, Boolean, text and "nil" type designators.
3. From a value one can extract its type.

type
2. ATyp = {|″int″,″rat″,″bool″,″text″,″nil″|}
value
3. xtr_ATyp: AVAL → ATyp
3. xtr_ATyp(v) ≡
3. **case** v **of**
3. mkIV(_) → ″int″,
3. mkRV(_) → ″rat″,
3. mkBV(_) → ″bool″,
3. mkTV(_) → ″text″,
3. ″nil″ → ″nil″
3. **end**

2.2.3 Atomic Sub-types

4. We can define a notion of atomic sub-types, is_atomic_sub_type.
   a) Value type vt is a sub-type of type vt for vt being any one of "integer", "rat","boolean", "text", and "nil".
   b) Type "int" is a [proper] sub-type of type "rat".
   c) Type "nil" is a [proper] sub-type of types "int" and "text".
   d) The law of transitivity expresses that if t' is a sub-type of type t'', and if type t'' is a sub-type of type t''', then t' is a sub-type t'''.
   e) By the law of transitivity type "nat" is a [proper] sub-type of type "rat".

[7] It is easy to extend the atomic value concept to composite value structures: sets, records, vectors, etc.; but we leave that for an engineering project following the lines of this paper.

value
4. is_atomic_sub_type: ATyp × ATyp → **Bool**
4. is_atomic_sub_type(vt′,vt″) ≡
4. **case** (vt′,vt″) **of**
4a. (vt,vt) → **true**,
4b. (″`int`″,″`rat`″) → **true**,
4c. (″`nil`″,″`int`″) → **true**,
4c. (″`nil`″,″`text`″) → **true**
4. **end**
axiom
4d. ∀ t′,t″,t‴:ATyp •
4d. is_atomic_sub_type(t′,t″) ∧ is_atomic_sub_type(t″,t‴)
4d. ⇒ is_atomic_sub_type(t′,t‴)
theorem
4e. is_atomic_sub_type(″`nil`″,″`rat`″)

2.2.4 Atomic Super-types

5. One can define a super-type predicate:
   a) Any atomic type is a super-type of itself.
   b) Any non-nil atomic type is a super-type of type `"nil"`.
   c) `"rat"` is a super-type of `"int"`.

value
5. is_atomic_super_type: ATyp × ATyp → **Bool**
5. is_atomic_super_type(at,at′)
5a. at=at′
5. ∨ **case** (at,at′) **of**
5b. (″`nil`″,at′) → **true**,
5c. (″`rat`″,″`int`″) → **true**,
5. _ → **false**
5. **end**

2.3 Curtain Values and Curtain Types

We introduce a notion of scroll down curtains. We are not happy with this design choice. The choice was made in order to illustrate (read: show the reader) that our modelling can capture essential aspects of actual windows. But it gives us specification-detail problems: much too many formulas and "special cases": whither a field item is an atomic, or a curtain, and why only, as we have chosen, simple atomic values as curtain elements and not also compound values such as structures (records, Cartesians), etc. And we can model curtains with the relational window values, as we shall see somewhat later. In a design "refinement" we shall later remove curtain from windows.

2.3.1 Curtain Values

6. A curtain value is a non-empty list of atomic values of the same type and terminated by a "blank".

6. CVAL′ == mkCV(s_vl:(AVAL|{|″blank″|})*)
6. CVAL = {|vl:CVAL′•wf_CVAL(vl)|}

2.3.2 Well-formed Curtain Values

Curtain values need be well-formed.

7. All, but the last of icon value of a curtain list are of comparable types and the last value is "blank"

value
7. wf_CVAL: CVAL′ → **Bool**
7. wf_CVAL(vl) ≡
7. ∀ i:**Nat**•i ∈ **inds** vl\{**len** vl}∧i+1 ∈ **inds** vl ⇒
7. i+1≠**len** vl
7. ⇒ comp_atomic_types(xtr_ATyp(vl(i)),xtr_ATyp(vl(i+1)))
7. ∧ vl(i)≠″blank″ ∧ vl(**len** vl)=″blank″

7. comp_atomic_types: ATyp × ATyp → **Bool**
7. comp_atomic_types(t,t′) ≡ atomic_sub_type(t,t′) ∨ atomic_sub_type(t′,t)

2.3.3 Curtain Types

8. Curtain types are atomic types.[8]

8. CTyp == mkCT(s_t:ATyp)

9. From a curtain value one can extract its curtain type.

9. xtr_CTyp: CVAL → CTyp
9. xtr_CTyp(mkCVAL(cv)) ≡
9. **if** cv=⟨″blank″⟩ **then** ″nil″ **else** xtr_type(**hd** cv) **end**

The above definition is just a convenience. Extracting the atomic type of "in-between" curtain list values might yield another type. Therefore we define a notion of curtain super-types.

[8] See Footnote 7 on page 17.

2.3.4 Curtain Super-types

10. We can define a function which extracts the atomic super-type of the non-"blank" elements of a curtain value.
    a) Let a curtain list have three or more elements.
    b) Let any two distinct of these other than the last, the "blank" element, have the atomic sub-types ati and atj.
    c) If ati is an atomic sub-type of atj then ati is an atomic super-type of atj.
    d) "rat" is an atomic super-type of "int".

```
value
10.  atomic_super_type: CVAL → {|"nil"|} → Nat → Bool
10.  atomic_super_type(mkCV(vl))(at)(i) ≡
10.    if i=len vl
10.      then at
10.      else
10.        is_atomic_sub_type(at,xtr_ATyp(vl(i))) →
10.          atomic_super_type(mkCV(vl))(xtr_ATyp(vl(i)))(i+1),
10.        is_atomic_sub_type(xtr_ATyp(vl(i)),at) →
10.          atomic_super_type(mkCV(vl))(at)(i+1)
10.    end
10.    pre len vl≥3 and i=1
```

2.4 Tuples

2.4.1 Tuple Values

11. Tuples (i.e., tuple values) are sets of fields, that is, of uniquely field-named field values.
12. A field name is either
    a) an atomic (value or type) name or
    b) a simple curtain (value or type) name or
    c) a curtain name with an index.
13. Window names are (also) just names.
14. Names are further undefined quantities.
15. A field value is either an atomic value or a curtain value.

```
type
11.   TVAL     =  (ANm -m-> AVAL) ∪ (CNm -m-> CVAL)
12.   FNm      =  ANm | CNm
12a.  ANm      == mkANm(s_nm:Nm)
12b.  CNm      == mkCNm(s_nm:Nm)
12c.  CNmIx    == mkCNmIx(s_nm:Nm,s_x:Nat)
13.   WNm      == mkWNm(s_wn:Nm)
14.   Nm
15.   FVAL     =  AVAL | CVAL
```

2.4.2 Field and Tuple Types

Fields or tuples are like attributes of relations. Hence field types are such attributes. We shall "stick" to the names of field elements, tuples, tuples values and tuple types (in lieu of attributes, attribute values and attribute types, respectively).

16. A field, that is, a tuple element type is either an atomic type or a curtain type.
17. A tuple type associates field names to field types.

```
type
16. FTyp = ATyp | CTyp
17. TTyp = (Anm -m-> ATyp) ∪ (CNm -m-> CTyp)
```

18. From a field value one can extract its type.

```
value
18.  xtr_FTyp: FVAL → FTyp
18.  xtr_FTyp(fv) ≡
18.    case fv of
18.      mkCV(_) → xtr_CTyp(fv),
18.      _ → mkAT(xtr_typ(fv))
18.    end
```

19. From a tuple value one can extract its type.

```
19. xtr_TTyp: TVAL → TTyp
19. xtr_TTyp(tv) ≡ [ fn↦xtr_FTyp(tv(fn))|fn:FNm•fn ∈ dom fv ]
```

2.5 Sub-types

We extend the sub-type relation of Sect. 2.2.3 to apply to any pair of types.

20. The extended sub-type relation applies to a pair of field types.
    a) If the two field types, ft and ft′, are both atomic types of type at and at′, then ft is a sub-type of ft′ if at is an atomic sub-type of at′.
    b) If the two field types, ft and ft′, are both curtain, that is, atomic types of type at and at′, then likewise.
    c) Otherwise they are not sub-types.

```
20.  sub_type: FTyp × FTyp → Bool
20.  sub_type(ft,ft′) ≡ ft=ft′ ∨
20.    case (ft,ft′) of
20a.     (mkAT(at),mkAT(at′)) → is_atomic_sub_type(at,at′),
20b.     (mkCT(at),mkCT(at′)) → is_atomic_sub_type(at,at′)
20c.     _ → false
20.    end
```

2.6 Keys and Relations

2.6.1 Keys: Key-names, Key-Values and Key-types

21. A key-name is an atomic icon name.
22. Key-names are sets of atomic icon names.
23. Key-values associate key-names with atomic values.
24. Key-types associate key-names with atomic types.
25. One can extract the key-type of a key-value.

```
type
21. KNn = ANm
22. KNms = ANm-set
23. Key, KVAL = ANm ~m~> AVAL
24. KTyp = ANm ~m~> ATyp
value
25. xtr_KTyp: KVAL → KTyp
25. xtr_KTyp(kv) ≡ [ an↦xtr_type(kv(an))|an:ANm•an ∈ dom kv ]
```

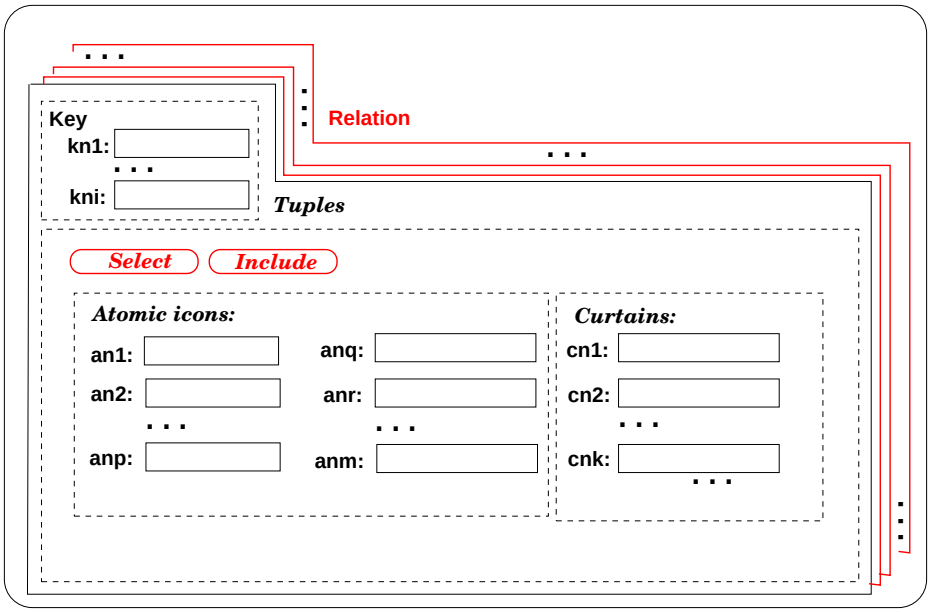


Figure 5: **Red** lines hint at a relation

2.6.2 Relations and Relation Types

We remind the reader of Fig. 2 repeated in Fig. 5.

26. A relation (a relation value) associates key values to fields. In a relation
    a) all key values have the same definition set of key-names; and
    b) all key values are sub-types of a postulated non-nil atomic super-type which is a key-type.

type
26. RVAL′ = KVAL $\overrightarrow{m}$ TVAL
26. RVAL = {|rv:RVAL′•wf_RVAL(rv)|}

value
26. xtr_RTyp: RVAL → RTyp
26. xtr_RTyp(rv) ≡ [an↦xtr_type(rv(an))|an:ANm•an ∈ **dom** rv]

26. wf_RVAL: RVAL′ → **Bool**
26. wf_RVAL(rv) ≡
26a. ∀ kv,kv′:KVal • {kv,kv′}⊆**dom** rv ⇒ **dom** kv = **dom** kv′
26b. ∧ ∃ kt:KTyp • ∀ kv:KVal•kv ∈ **dom** rv⇒super_type(kt,xtr_KTyp(kv))

27. A non-nil tuple value has none of its
    a) atomic field values being "nil",
    b) curtain values have "nil" element values.

```
value
27. is_non_nil_TVAL: TVAL → Bool
27. is_non_nil_TVAL(tv) ≡
27.     ∀ fn:FNm•fn ∈ dom tv ⇒
27.       case tv(fn) of
27a.           mkATyp(av) → av≠″nil″,
27b.           mkCTyp(av) → ″nil″∉ elems av
27.       end
```

28. A relation type associates field names with field, that is non-nil atomic icon or non-nil curtain types.

```
type
28.  RTyp′ = (Anm ~m~> ATyp) ∪ (CNm ~m~> CTyp)
28.  RTyp = {|rt:RTyp′•wf_RTyp(rt)|}

value
28.  wf_RTyp: RTyp′ → Bool
28.  wf_RTyp(rt) ≡
28.    ∀ fn:FNm•fn ∈ dom rt ⇒
28.      case rt(fn) of
28.        mkATyp(at) → at≠″nil″,
28.        mkCTyp(at) → at≠″nil″
28.      end
```

2.6.3 Auxiliary Functions on Relations

29. With a key-value, a relation value and a relation type one can construct an initial tuple for that key even though the current relation does not have a tuple with that key-value.

30. The function init_tpls generates "nil"-field values

31. of the appropriate kind.

```
29. sel_tpls: KeyVAL × RVAL × RTyp → TVAL
29. sel_tpls(kv,rval,rtyp) ≡
29.   if kv ∈ dom rval then rval(kv) else init_tpls(ttyp) end
```

30. init_tpls: TTyp → TVAL
30. init_tpls(ttyp) ≡ [fn↦init_fld_val(ttyp(fn))|fn:FNm•fn ∈ **dom** ttyp]

31. init_fld_val: FTyp → FVAL
31. init_fld_val(ftyp) ≡
31. **case** ftyp **of**
31. mkCTyp(_) → mkCVAL(⟨″`blank`″⟩)
31. _ → ″`nil`″
31. **end**

2.7 Windows

2.7.1 Window Values

32. A window value is a quadruple: a set of key names, a tuple value, a relation value and a set of window names.

32. WVAL == mkWV(s_key:KeyNms,s_tpl:TVAL,s_rel:RVAL,s_ws:WNm-**set**)

2.7.2 Window Value Types

33. Window types are tuple types.

33. WTyp = TTyp

2.7.3 Window Syntax

34. A window is a triple: a window name, a window type and a window value.

type
34. W′ = WNm × WTyp × WVAL

2.7.4 Well-formed Windows

35. A window is well-formed if

    a) Key-names: the names of the primary key are a subset of the names of the atomic values of the tuple values (and hence also tuple types).

    b) Fields and Window Type Names: the names of field values are the same as the names of the fields of the window type.

c) Subtypes I: the type of the field values is a sub_type of the window type.

d) Consistent Key Definition Sets: the relational window value definition set (called the indexes) contains same definition set keys;

e) Subtype II: the type of the relational field values is a sub_type of the window type;

f) No "Immediate Circular" Windows: the window name is not in the set of sub-window names. (This is a pragmatic design point serving to avoid confusion.)

type
35. W = {|w:W′•wf_W(w)|}
value
35. wf_W: W′ → **Bool**
35. wf_W(wn,mkWTyp(wtyp),mkWV(key,tpl,rel,wns)) ≡
35a. key ⊆ **dom** tpl
35b. ∧ **dom** wtyp = **dom** tpl
35c. ∧ sub_type(xtr_ftys(tpl),wtyp)
35d. ∧ ∀ kv:KVAL • kv ∈ **dom** rel ⇒ key = **dom** kv
35e. ∧ sub_type(xtr_TTyp(rel(kv)),wtyp)
35f. ∧ wn∉ wns

2.7.5 The "Select" and "Include" Buttons

If the key-name set is non-empty then two "button" are displayed, say close to the key. An empty key-name set designates that the window value's relation is similarly always empty. A non-empty key designates that the window value's relation is usually non-empty. It is the intention that the relation of an initial window with a non-empty key-name set contains exactly one tuple, for example:

value
kn:KeyNm = {ka,kb}, fns:FNms-**set** = {fc,fd,fe}
rel = {[ka↦″nil″,kb↦″nil″,fc↦″nil″,fd↦″nil″,fe↦mkCV(⟨″blank″⟩)]}

The is, the initial key-value, kv, displayed on the window, is

value
kv:KeyVAL = [ka↦″nil″,kb↦″nil″]

Clicking the "Select" button will then display, cf. Item 29 on page 24, the tuple:

value
tpl:TVAL = [fc↦″nil″,fd↦″nil″,fe↦mkCV(⟨″blank″⟩)]

2.7.6 Null, Initial and Nil Windows

Null Windows

36. Let us recall the syntax of windows (Items 34 and 35 on page 25).
37. A "null" window is a window with some name, say wnm, where all value fields are empty and whose sub-window name set is empty.

type
36. W′ = WNm×WTy×mkWV(ANm-**set**,Tpl,FRel,WNm-**set**)
value
37. null_W:W = (wnm,[ ],mkWV([ ],[ ],[ ],{}))

38. Null windows are well-formed for any window name.

38. **theorem:** wf_W(wnm,[ ],mkWV([ ],[ ],[ ],{}))

Initial Windows

39. We consider init_W to be a relation, a function which when invoked non-deterministically yields
40. an arbitrarily valued
41. well-formed window.
    a) The window name, the window type and the key names are thought of as arbitrarily chosen.
    b) The relation is likewise arbitrarily chosen but
        i. key names must be a subset of the field names listed in the window type;
        ii. tuple names must equal field names listed in the window type; and
        iii. for all field names of the tuples
        iv. the type of the field name-selected value must be a sub-type of the same-name named type in the window type.
    c) The relation must satisfy the following.
        i. The key name set must be a subset of the tuple names.
        ii. For all key-values of the relation
            1. the type of these key-values must be a sub-type of the correspondingly named type of the window type;
            2. the key-values must also occur in the 'fields';

3. and type of the entire indexed field relation values must be a sub-type of the window type;

d) The set of sub-window names is arbitrarily chosen.

```
value
39. init_W: WNm → W
39. init_W(wn) ≡
41a.     let wt:WTyp,
41a.         kn:KeyNm,
41b.         tv:TVAL • wf_iTVAL(kn,wt,tv),
41c.         rv:RVAL•wf_iRVAL(kn,wt,rv),
41d.         ws:WNm-set in
40.      (wn,wt,mkWV(kn,tv,rv,ws)) end

41. theorem: ∀ w:W • let w = init_W() in wf_W(w) end
```

```
value
41b.  wf_iTVAL: KeyNm × TTyp × TVAL → Bool
41b.  wf_iTVAL(kn,tt,tv) ≡
41(b)i.     kn ⊆ dom tt
41(b)ii.    ∧ dom tt = dom tv
41(b)iii.    ∧ ∀ fn:FNm • fn ∈ dom tv ⇒
41(b)iv.          sub_type(xtr_type(tv(fn),tt(fn)))

41c.  wf_iRVAL: KeyNm × WTyp × RVAL → Bool
41c.  wf_iRVAL(kn,wt,rv) ≡
41(c)i.     kn ⊆ dom wt
41(c)ii.    ∧ ∀ kv:KeyVAL • kv ∈ dom rv ⇒
41(c)ii1.        sub_type(xtr_KTyp(kv),wt)
41(c)ii2.       ∧ kv ∩ rv(kv) ≠ {} ⇒
41(c)ii3.          sub_type(xtr_FTyp(rv(kv)),wt)
```

Nil Tuple

42. To generate a nil tuple we must know its tuple type.

```
value
42.  nil_TVAL: TTyp → TVAL
42.  nil_TVAL(tt) ≡
42.     [ fn ↦ nil_FVAL(tt(fn))|fn:FNM•fn ∈ dom tt ]
```

Nil Field Values

43. To generate a nil tuple element value we must know its type.

```
value
43.  nil_FVAL: (ATyp|CTyp) → TVAL
43.  nil_FVAL(ft) ≡
43.    case ft of
43.      mkCT(at) → mkCV(″nil″),
43.      _ → ″nil″
43.    end
```

Nil Windows An example initial window shall have all its value fields have the "nil" value.

```
is_nil_W: W → Bool
is_nil_W(w:(wn,wtyp,mkWV(kn,tv,rv,wns))) ≡
      ∀ fn:FNm•fn ∈ dom tv ⇒ is_nil_FVAL(tv(fn))
    ∧ let nil_kv = [ an↦″nil″|an:ANm•an ∈ kn ] in
      dom rel = {nil_kv}
    ∧ ∀ fn:FNm•fn ∈ dom rv(nil_kv) ⇒ is_nil_FVAL((rv(nil_kv))(fn)) end

is_nil_FVAL: FVAL → Bool
is_nil_FVAL(v) ≡ case v of mkCV(cv)→cv=⟨″blank″⟩,_→v=″nil″ end
```

3 A Window Frame System

window-frames

window-frames

Operations on windows are operations on windows of a window frame.That is, the somehow structured cluster of windows "seen" on a computer (mobile phone or pad) display screen. Think of it as follows: There is a window and there is an indefinite space of uniquely window-named window frames (all displayable on a screen)[9].

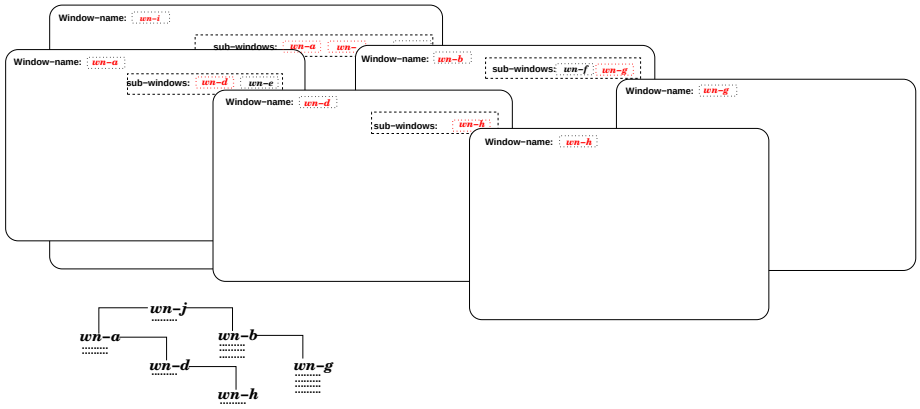


Figure 6: A cactus stack of windows

Figure 6 shows such a cactus stack. A possible sequence of openings is: the first window to be opened has name *wn-j*; from that window two windows were opened: *wn-a* and *wn-b*; from window *wn-b* window *wn-g* was opened; from window *wn-a* window *wn-d* was opened and from window *wn-d* window *wn-h* was opened. The dotted lines (. . . . .) under the window names in the small lower left cactus stack summary shall indicated that between these window openings work may "progress" on already opened windows with respect to creating and updating tuples of the window relations. At the moment the snapshot of Fig. 6 "was taken" the number of relation tuples of these windows are: window *wn-j* one tuple; window *wn-a* two tuples; window *wn-b* three tuples; window *wn-d* one tuple; window *wn-g* four tuples and window *wn-h* one tuple; This is why what is "grown" here is not a conventional tree but a cactus. The snapshot does not indicate the order in which these relation tuples were operated upon (created or updated).

3.1 Window States

44. A window frame consists of a "root" window state and zero, one or more window named window frames.

[9] We shall call the windows of thedomains part of the (window,domains) pairs for the *sub-window* of the window.

45. A window state is a triple: a window, a ("highlighted") field value (of the field designated by a cursor) and the cursor.

46. The cursor designates one of the fields of the window, that is, the cursor "is" a field name, or an index curtain name, or a window name, or the `"update"` "button" name, or the `"close"` "button" name.

47. A "bottom-most" window has the "built-in" name `"wn_bottom_WF"`.

type
44. WF = WΣ × (WNm $\xrightarrow[m]{}$ WF)
45. WΣ = W × FVAL × Cursor
46. Cursor = FNm | CNmIx | WNm | ″`update`″ | ″`close`″

value
47. wn_bottom_WF: WNm

That is, the cursor is positioned at

- an atomic icon, or
- a curtain, or
- a curtain element, or
- a window name, or
- the `"update"` "button",
- the `"close"` "button",
- the `"select"` "button", or
- the `"include"` "button".

3.2 Well-formed Window Frames

48. A well-formed window frame (in the context of a window name) must have
    a) a well-formed window state;
    b) the window name be the same as the context window name; and not in the window's (sub-domain) window names;
    c) the definition set of the window named window frames be a subset of the window's (sub-domain) window names; and
    d) all the opened sub-windows are well-formed.

value
48. wf_WF: WF → WNm → **Bool**
48. wf_WF(wσ:(w:(wn,_,mkWV(_,_,_,wns)),fval,c),wfs)(wnm) ≡
48a. wf_WΣ(wσ)
48b. ∧ wn = wnm ∧ wn∉ wns
48c. ∧ **dom** wfs ⊆ wns
48d. ∧ ∀ wn′:WNm•wn′ ∈ **dom** wfs ⇒ wf_WF(wfs(wn′))(wn′)

3.3 Well-formed Window States

49. A window state is well-formed

    a) if the (as can be assumed, the window and) window name selected window frame is well-formed, and

    b) if the cursor is positioned at a proper field (c ∈**dom** tpl) and the window state's field value equals the value of the cursor named (and possibly indexed) field, fv=tpl(c) etc.,

    c) otherwise is the window state is ill-formed.

49. wf_WΣ: → **Bool**
49. wf_WΣ(w:mkWV(_,_,mkWV(_,tpl,_,_)),fv,c) ≡
49a. ∧ wf_W(w)
49. **case** c **of**
49b. mkANm(nm) → c ∈ **dom** tpl ∧ fv = tpl(c),
49b. mkCNm(nm) → c ∈ **dom** tpl ∧ fv = tpl(c),
49b. mkCNmIx(nm,x) →
49b. nm ∈ **dom** tpl ∧ x ∈ **inds** tpl(nm) ∧ fv = (tpl(nm))(x)
49c. _ → **false**
49. **end**

3.4 Forests of Window Frames

We extend the notion of window frames into sets of these. Figure 7 intends to show a computer display screen with more than one window tree.

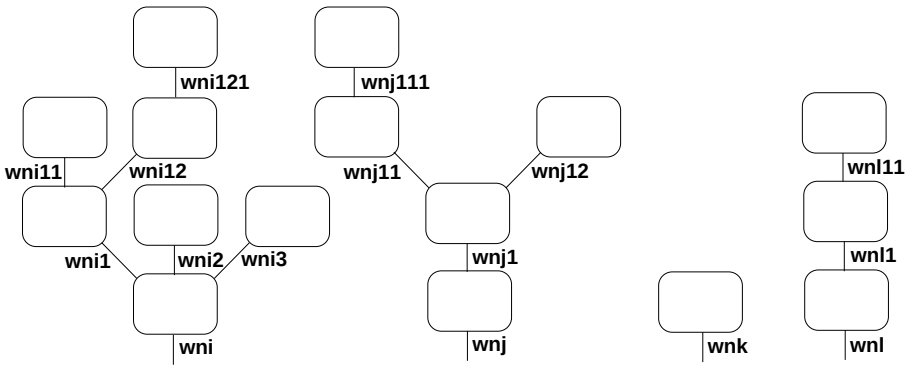


Figure 7: A forest of window frames

3.4.1 The Syntax

50. A forest of window frames "turns" the notion of a window frame around:
    - a window frame has a window state and a set of uniquely named window frames.
    - a forest of window frames has a set of uniquely named pairs of window states and sets of uniquely name window frames.

type
50. FoWF = WNm $\overrightarrow{m}$ (WΣ × FoWF)

3.4.2 The Well-formedness

51. The well-formedness of a forest of window frames is the well-formedness of
    a) all of its subsidiary
    b) window states and
    c) forest of window frames.

value
51. wf_FoWF: FoWF → **Bool**
51. wf_FoWF(fowf) ≡
51. ∀ wn:WNm•wn ∈ **dom** fowf ⇒
50a. **let** (wσ,fofw′) = fowf(wn) **in**
50b. wf_WΣ(wσ)(wn)
50c. ∧ wf_FoFW(fofw′) **end**

3.5 Paths of Window Frames and Forests

3.5.1 Syntax

51. A path is either a 0-path or it is a 1-path.
    a) A 0-path is a sequence of zero, one or more window names.
    b) A 1-path is a non-empty 0-path.

The empty path identifies the bottom window frame (otherwise "named": `"wn_bottom_WF"`).

51. P = WNm*
51a. P0 = P
51b. P1 = {|p:P•p≠⟨⟩|}

3.5.2 Window Frames Define Paths

52. A window frame wf:(w,wfs) defines a set of window paths.
    a) The empty path is a window path.
    b) If wn is a name of a window frame, wf' of wfs, then ⟨wn⟩ is a path. That path "points to" a pair: (window-state,window-frames).
    c) If a path p points to a pair (window-state,window-frames) then for every window name, wn in window-frames, p⌢⟨wn⟩ is a path of wf.

We define the paths function to apply to window frames. The latter are introduced later!

```
52.  paths: WF → P1-set
52.  paths(((wn,_,),_,_),wfs) ≡
52a.     let ps = {⟨⟩} ∪
52b.                ∪{paths(wf)|wf:WF•wf ∈ rng wfs} in
52c.     ∪{⟨wn⟩,⟨wn⟩⌢p|p ∈ ps} end
```

3.5.3 Forests of Window Frames Define Paths

53. Paths of forests of window frames are defined very much like the paths of window frames, Item 52.

```
type
50.  FoWF = WNm ~m~> (WΣ × FoWF)
value
53.  paths: FoWF → P1-set
53.  paths(fofw) ≡
53.    if fofw=[ ]
53.      then {}
53.      else let ps = ∪{let (_,fofw′) = fofw(wn) in
53.                        {⟨wn⟩} ∪
53.                        ∪{paths(fofw′)|wn:WNm•wn ∈ dom fofw} end} in
53.           ∪{⟨wn⟩,⟨wn⟩⌢p|p ∈ ps}
53.    end end
```

3.5.4 Selection Functions

54. From a window (or a domain[10]) frame one can select a sub-window-frame given a possibly empty path.

[10] Domain frames will be introduced in Sect. 4

Select Window Frames

value
54. s_FoWF: P0 × FoWF $\xrightarrow{\sim}$ FoWF
54. s_FoWF(p,fowf) ≡
54. **case** p **of**
54. ⟨wn⟩ → fowf(wn),
54. ⟨wn⟩⌢p′ → s_Frame(p′,fowf(wn))
54. **end**
54. **pre** p ∈ paths(fowf)

Select Window States

55. From a window frame one can select a window state given a possibly empty path.

value
55. s_WΣ: P0 × FoWF $\xrightarrow{\sim}$ WΣ
55. s_WΣ(p,fowf) ≡ **let** (wσ,_) = s_FoWF(p,fowf) **in** wσ **end**
55. **pre** p ∈ paths(fowf)

Select Windows

56. From a forest of window frames one can select a window given a possibly empty path.

value
56. s_W: P0 × FoWF $\xrightarrow{\sim}$ W
56. s_W(p,fowf) ≡ **let** ((_,w,_,_),_) = s_FoWF(p,fowf) **in** w **end**
56. **pre** p ∈ paths(fowf)

Select Window Names

57. From a forest of window frames one can select a the sub-window names of the sub-forest of window frames at a possibly empty path position.

value
57. s_WNms: P0 × FoWF $\xrightarrow{\sim}$ WNm-**set**
57. s_WNms(p,fowf) ≡ **let** (_,fowf) = s_FoWF(p,fowf) **in dom** fowf **end**
57. **pre** p ∈ paths(fowf)

domain-frames

4 The Domain Frame System domain-frames

The concept of domain frames is introduced in order to cover a notion of a possibly world-wide distributed data space, i.s., a repository. from where users can populate their window frames and to where they can deposit windows of their window frames.

Operations on domains are operations on domain frames.Think of it as follows: There is a window and there is an indefinite space of window-named domains.

4.1 The Syntax of Domain Frames

58. Domain frames map window names into a pair of windows and a set of uniquely window named domain frames[11]

type
58. DF = W × (WNm $\overrightarrow{m}$ DF)

4.1.1 Well-formed Domain Frames

59. A domain frame is well-formed
    a) if windows of (window,domain frames) pairs are well-formed;
    b) if the domain frames part of the (window,domain frames) pairs records exactly the window names of the window; and
    c) if each domain frame of the domain frames part of the (window,domain frames) pairs is well-formed.

value
59. wf_DF: DF → **Bool**
59. wf_DF(w:(wn,_,mkWV(_,_,_,wns)),dfs) ≡
59a. wf_W(w)
59b. ∧ wns=**dom** dfs **compare:** Item 48c on page 31.
59c. ∧ ∀ wn:WNm • wn ∈ **dom** dfs ⇒ wf_DF(dfs(wn))

4.2 The Syntax of Forests of Domain Frames

60. A forest of domain frames "turns" the notion of a domain frame around:
    - a domain frame has a window and a set of uniquely named domain frames.

[11] We shall call the domains of the domains part of the (window,domains) pairs for the *sub-domain* of the window. of the window.

- a forest of domain frames has a set of uniquely named pairs of windows and sets of uniquely name domain frames.

60. FoDF = WNm → (W × FoDF)

4.2.1 Well-formed Forests of Domain Frames

```
value
  wf_FoDF: FoDF → Bool
  wf_FoDF(fodf) ≡
      ∀ wn:WNm•wn ∈ dom fowf ⇒
          let (w,fodw′) = fodf(wn) in
          wf_W(w)(wn)
       ∧ wf_FoDW(fodw′) end
```

4.3 Paths of Domain Frames and Forests of Domain Frames

This section is "isomorphic" to Sect. 3.5.

61. If wn is a domain name of d then ⟨wn⟩ is a path. That path "points to" a pair: (window,domain frames).

62. If a path p points to a pair (window,domain frames) then for every window name, wn in domain frames, p⌢⟨wn⟩ is a path of df.

4.3.1 Domain Frames Define Paths

Parametrically identically to Item 52a on page 34's definition of paths over window frames we can define paths over domain frames.

```
52a. paths: DF → P1-set
52a. paths((wn,_),dfs) ≡
52b.     let ps′ = ∪{paths(df)|df:DF•df ∈ rng dfs} in
52b.     ∪{⟨wn⟩,⟨wn⟩⌢p|p ∈ ps′} end
```

4.3.2 Forests of Domain Frames Define Paths

63. Paths of forests of domain frames are defined very much like the paths of window frames, Item 52.

```
type
63. FoDF = WNm ~m~> (W × FoWF)
value
63. paths: FoDF → P1-set
63. paths(fodw) ≡
63.   if fodw=[ ]
63.     then {}
63.     else let ps = ∪{let (_,fodw′) = fodw(wn) in
63.                     {⟨wn⟩} ∪
63.                     ∪{paths(fodw′)|wn:WNm•wn ∈ dom fodw} end} in
63.          ∪{⟨wn⟩,⟨wn⟩⌢p|p ∈ ps}
63.   end end
```

4.3.3 Selection Functions

Select Window from Domain Frame

64. The window selection function (Item 56 on page 35) is slightly different.

```
value
64. s_W: P0 × DF ~→ W
64. s_W(p,df) ≡ let (w,_) = s_WF(p,df) in w end
64. pre p ∈ paths(df)
```

Select Window from Forest of Domain Frames

65. From a forest of domain frames one can select a window given a path.

```
value
65. s_W: P0 × FoDF ~→ W
65. s_W(p,fodf) ≡ let (w,_) = s_FoWF(p,fowf) in w end
65. pre p ∈ paths(fowf)
```

Select Window Names from FoDFs

66. From a forest of domain frames one can select a the sub-window names of the sub-forest of domain frames at a path position.

```
value
66. s_WNms: P1 × FoWF ~→ WNm-set
66. s_WNms(p,fodf) ≡ let (_,fodf) = s_FoWF(p,fodf) in dom fodf end
66. pre p ∈ paths(fowf)
```

5 Domain Frame Operations domain-ops

domain-ops

Commands are syntactic structures. Commands, when applied to states become operations. The meanings of commands are usually state-to-state changing functions. We then prefix the names of the semantics functions over such commands int_ for semantics interpretation functions. When the prefix of the name of a semantic function is eval_ then it is because the operation applies to a state, does not change it, but yields a value.

5.1 Commands

5.1.1 Narrative

67. The are the following forest of domain frame commands: initialize forest, create domain frame, remove domain frame, create window, get window, update window, delete window.
    a) The initialize domain need not present any arguments (other than that it is an initialize domain command).
    b) The create domain frame command presents a path and a window name.
    c) The remove domain frame command presents a non-empty path.
    d) The update window command presents a path, the name of the window, that window.
    e) The get window command presents a name path and a window name.

5.1.2 Formalisation

type
67. DFCmd = DFIniDF | DFCreDF | DFRmDF | DFPutWi | DFGetW
67a. DFIniDF == "`initialize`"
67b. DFCreDF == mkDFCDF(s_p:P0,s_wn:WNm)
67c. DFRmDF == mkDFRDF(s_p:P0,s_wn:WNm)
67d. DFPutW == mkDFPW(s_p:P0,s_wn:WNm,s_w:W)
67e. DFGetW == mkDFGW(s_p:P0)

5.2 Operations

All operations apply in the context of a domain.

68. The get window operation possibly yields a window but does not change the forest of domain frames.

69. All other domain operations either changes the forest of domain frames or are undefined.

68. eval_DFGetW: DFGetW → FoDF$\xrightarrow{\sim}$ W
69. int_″X″: ″X″ → FoDF $\xrightarrow{\sim}$ FoDF

where "X" is one of DFIniDF, DFCreDF, DFRmDF or DFPutW.

5.2.1 The Initialize Domain Frame Operation

70. The initialize domain command takes only the command argument but yields an empty domain.

70. int_DFIniDF: DFIniDF → DF
70. int_DFIniDF(″initialize″) ≡ [ ]

5.2.2 The Create Domain Frame Operation

The domain to be created is initially empty so we need only give a path to the position in the forest of domain frames the empty domain is to be installed.

71. The create domain frame operation, int_DFCreDF(mkDFCDF(p,wn))(fodf), is partial; it produces a changed domain.
72. To express the changed domain, from fodf to fodf′, we use the concept of all paths, ps, ps′, of respectively fodf and fodf′. **Precondition:**
    a) The path, p, of the command must be a path of the forest of domain frames; and
    b) the window name, wn, of the command must not be of a[n opened] window of the forest of window frames of the forest of window frames selected by p.

    Postcondition:

    c) First the only change with respect to paths is that ps′ is equal to ps union with the new path ({p⌢⟨wn⟩})
    d) afforded by extending the forest of domain frames of the domain frame designated by p in fodf with an initial window, named wn, as part of the new forest of domains[12].
    e) Then all paths common to ps and ps′, that is, all paths of ps, designate the same domain frames in fodf and fodf′,
    f) except that the window (w) of the selected window frame has its sub-window names part augmented (into w′) with wn (all other parts (of w) are the same as in w′).

[12] How that window is initialised we do not specify – other than through the non-determinism of the init_W() operation [Item 39 on page 27.].

71. int_DFCreDF: DFCreDF → FoDF $\xrightarrow{\sim}$ FoDF
71. int_DFCreDF(mkDFCDF(p,wn))(fodf) **as** fodf′
72. **let** ps = paths(fodf), ps′ = paths(fodf′) **in**
72a. **pre** p ∈ paths(fodf)
72b. ∧ {p⌢⟨wn⟩}∉ ps
72c. **post** ps′ = ps ∪ {p⌢⟨wn⟩} ∧ {p⌢⟨wn⟩}∉ ps
72e. ∧ ∀ p′:P•p′ ∈ ps•s_DF(p′,fodf)=s_DF(p′,fodf′)
72d. ∧ **let** (w:(wn′,wt′,wv′),fodfb)=s_DF(pp,fodf),
72d. (w′:(wn″,wt″,wv″),fodfa)=s_DF(pp,fodf′) **in**
72d. wn′=wn″ ∧ wt′=wt″ ∧ fodfa = fodfb ∪ [wn↦(init_W(),[ ])]
72f. ∧ **let** mkWV(kn′,tpl′,rel′,ws′)=wv′,mkWV(kn″,tpl″,rel″,ws″)=wv″ **in**
72f. kn′=kn″ ∧ tpl′=tpl″ ∧ rel′=rel″ ∧ ws″=ws′ ∪ {wn}
71. **end end end**

5.2.3 The Remove Domain Frame Operation

73. The remove operation, int_DFRmDF(mkDFRDF(p,wn))(fodf), is partial.

Precondition:

a) The window path {p⌢⟨wn⟩} of the command must be a window path of fodf.

Postcondition:

b) To express the changed domain, from fodf to fodf′ we use the concept of all paths, ps, ps′, of respectively fodf and fodf′.

c) First the change with respect to paths is that ps′ is equal to ps with all those paths, p, in ps,

   i. which are a proper prefix of paths, {p⌢⟨wn⟩}, in ps′
   ii. removed as a result of removing the domain frame designated by {p⌢⟨wn⟩} in fodf.

d) Then all paths common to ps and ps′, that is, all paths of ps′ designate the same domain frames in fodf and fodf′

e) except that the window (w) of the selected window frame has its sub-window names part reduced (into w′) with wn (all other parts (of w) are the same as in w′).

73. int_DFRmDF: DFRmDF → FoDF $\xrightarrow{\sim}$ FoDF
73. int_DFRmDF(mkDFRDF(p,wn))(fodf) **as** fodf′
73. **let** ps = paths(fodf), ps′ = paths(fodf′) **in**
73a. **pre** p⌢⟨wn⟩ ∈ ps
73b. **post** ps′ = ps \ rm_paths(ps)(p⌢⟨wn⟩)
73d. ∧ ∀ p′:P•p′ ∈ ps′•s_DF(p′,fodf)=s_DF(p′,fodf′)
73e. ∧ **let** (w:(wn′,wt′,wv′),fodfb)=s_DF(p,df),

73e. (w′:(wn″,wt″,wv″),fodfa)=s_DF(p,df′) **in**
73e. wn′=wn″ ∧ wt′=wt″ ∧ fodfa = fodfb ∪ [wn↦(init_W(),[ ])]
73e. ∧ **let** mkWV(kn,tpl,rel,ws)=wv,mkWV(kn′,tpl′,rel′,ws′)=wv′ **in**
73e. kn=kn′ ∧ tpl=tpl′ ∧ rel=rel′ ∧ ws′=ws \ {wn}
73. **end end end**

73(c)i. is_prefix: P1 × P1 → **Bool**
73(c)i. is_prefix(p,p′) ≡ **len** p<**len** p′ ∧ ∀ i:**Nat**•i ∈ **inds** p⇒ p(i)=p′(i)

73(c)ii. rm_paths: P0-**set** → P1 → P0-**set**
73(c)ii. rm_paths(ps)(p^⟨wn⟩) ≡ {p′|p′:P0•∼prefix(p,p′)}

Identity of Remove Composed with Create

74. Given an domain, d, any path p in d and any window name wn not at p in d.

75. The effect of creating a(n initially null) domain at p in d and then removing the domain named wn at p in d is the initial domain d.

theorem
74. ∀ df:DF, p:P0, wn:WNm • p ∈ paths(fodf) ∧ wn ∉ s_WNms(p,fodf)
75. ⇒ **let** fodf′ = int_DFCreDF(mkDFCDF(p,wn))(fodf) **in**
75. int_DFRmDF(mkDFRDF(p,wn))(fodf′) = fodf **end**

5.2.4 The Put Window Operation

76. The int_DFPutW(mkDFPW(p,wna,wa:(wnb,wta,wva)))(fodf) operation is partial. **Precondition:**

    a) To express the changed domain, from d to d′ we use the concept of all paths, ps, ps′, of respectively fodf and fodf′.
    b) The window path, p, of the command must be a window path of fodf;
    c) the window name, wn, of the second argument of the command must be the same as in the window of the command, and must be in the domain selected by p of fodf;
    d) the argument window must be well-formed; and
    e) the argument window type must conform to the type of the window in fodf at {p^⟨wn⟩}.

    Postcondition:

    e) First these two sets of paths are identical.

f) Then all paths, except the path to the updated window, must designate, pairwise, the same domain frames before and after the operation.

g) The window names, types and sub-domains must be unchanged.

h) The window tuple is replaced by the argument tuple.

i) The relations is updated with respect to (wrt.) what they were in the domain frame version of the (before) window and wrt. the argument relation.

j) The sub-domain frame window names are unchanged.

```
76.  int_DFPutW: DFPutW → FoDF ~→ FoDF
76.  int_DFPutW(mkDFPW(p,wna,wa:(wnb,wta,wva)))(fodf) as fodf′
76a.   let ps = paths(fodf), ps′ = paths(fodf′),
76.        mkWV(kn,tpla,rela,wsa) = wva in
76b.   pre p ∈ ps ∧ p^⟨wna⟩ ∈ ps
76c.     ∧ wna = wnb ∧ wna ∈ s_WNms(p,fodf)
76d.     ∧ wf_W(wa)
76e.     ∧ let ((wnc,vtd,_),_) = s_DF(p^⟨wna⟩,fodf) in
76e.         wna = wnc ∧ wta = wtd end
76e.   post   ps′ = ps
76f.       ∧ ∀ p′:P•p′ ∈ ps′\{p^⟨wn⟩} •s_DF(p′,fodf)=s_DF(p′,fodf′)
76.        ∧ let ((wn,wt,wv),fodf″) = s_DF(p^⟨wna⟩,fodf),
76.              ((wn′,wt′,wv′),fodf‴) = s_DF(p^⟨wn⟩,fodf′) in
76g.         wna=wn=wn′ ∧ wt=wt′ ∧ df″=df‴
76.        ∧ let mkWV(kn,tpl,rel,ws) = wv,
76.              mkWV(kn′,tpl′,rel′,ws′) = wv′ in
76h.         tpl′ = tpla
76i.       ∧ rel′ = rel † [ fn↦rela(fn)|fn:FNm•fn ∈ kn ]
76j.       ∧ ws′ = ws = wsa
76.    end end end
```

5.2.5 The Get Window Operation

77. The get window does not change the domain – otherwise the operation result seems obvious !.

78. The path of the command must be in the domain of the operation.

```
77.  eval_DFGetW: DFGetW → FoDF ~→ W
77.  eval_DFGetW(mkDFGW(p,wn))(fodf) ≡ s_W(p,fodf,wn)
78.    pre p^⟨wn⟩ ∈ paths(fodf)
```

5.3 Discussion

5.3.1 Mon. 30 Aug. and Thu. 23 Sept., 2010

Some "loose" remarks – expressed at a time (Mon. 30 Aug. and Thu. 23 Sept., 2010) when I had not yet had time to carefully study my own narratives and formalisations from the point of view of the following issues:

- It seems that several of the annotation items can be expressed (even) more concisely, also less operationally, and perhaps even "schematised", see next item.
- It seems that several of the **pre-** and **post-**condition formalisations can, perhaps, be expressed in terms of "pre-cooked" predicate functions: many (∧-)clauses appears to share commonalities that could be "put" into so, appropriately named functions.
- Either of the above actions would undoubtedly lead to a clearer understanding of the system being designed as I (up till at least this day, Mon. 30 Aug., 2010) proceed in completing this section !

6 Window Frame Commands and Operations window-ops

window-ops

6.1 Commands

6.1.1 Narratives and Brief Descriptions

79. These commands are suggested to designate a set of operations on windows: open window, put widow, close window, click (on) window field or "button", write into window field, select relation fields, and include fields.

    a) The open window command presents a path, a window name and a key-value. The path designates a domain frame (i.e., a (window,forest of window frames)) pair. The window name designate the window. That window is to be the result of the open window operation. If the path is a singleton window name then a new window frame in the current forest of window frames is created.

    b) The close window command closes the designated window frame window. At the same time it closes (i.e., "removes") all the window frames at the designated path position and sub-windows and sub-frames thereof. If the path is a singleton window name then and existing window frame in the current forest of window frames is removed.

    c) The put (or store window value in a forest of domain frames) window command is, in a sense, the reverse of the get window operation; it "puts" back into the forest of domain frames a copy of a current window (which can then, subsequently, be closed).

    d) The click on (or select) window command positions the cursor at a field of the tuple.

    e) The write window command updates such a field with an atomic or curtain value.

    f) The select command "retrieves" a tuple from the (not displayed) relation (which always "underlies" a window). The tuple which is selected from the relation is the one which, in a sense to be made precise, "satisfies" the key-value of the currently (displayed) tuple,

    g) The include command is, in a sense, the reverse of the select command. It updates the relation with a "pair": the [key↦tuple], where key is from that part of the currently dispalyed window whose field names match the window's key-names and whose tuple are the remaining fields of the tuple.

6.1.2 Formalisations

type

79. WFCmd = WFOpnW|WFCloW|WFPutW|WFClkW|WFWrW|WFSelTpl|WFIncTpl

79a. WFOpnW == mkWFOpnW(s_p:P1,s_wn:WNm,s_kv:KVAL)
79b. WFCloW == mkWFCloW(s_p:P1,s_wn:WNm)
79c. WFPutW == mkWFPW(s_p:P1,s_wn:WNm)
79d. WFClkW == mkWFClkW(s_p:P1,s_wn:WNm,s_f:FPos)
79d. FPos = ANm | CNm | CNmIx
79e. WFWrW == mkWFWrW(s_p:P1,s_wn:WNm,v:FVAL)
79f. WFSelTpl == mkWFSel(s_p:P1,w_wn:WNm,s_kv:KVAL)
79g. WFIncTpl == mkWFInc(s_p:P1,w_wn:WNm)

6.2 Operations

To express the changed fowf (into fowf′) resulting from the operations we express some predicates over the sets of paths, ps, ps′, of fowf and fowf′, that is, before and after execution of the open window operation. This style of expressing "storage" changes was used in rather much the same way for defining domain operations (in Sect. 5.2).

6.2.1 Open Window (Frame)

The *open* window operation, int_WFOpnW(mkWFOpnW(p,wn,kv))(fowf)(fodf), is intended to open a "fresh" window frame, (window,forest of window frames), at *location* p under the name of wn and with an initially void forest of window frames. That window, window, is obtained from the forest of domain frames, fodf, at location p⌢⟨wn⟩. The intention is now for the user to *click* (see Sect. 6.2.3 on page 49) on that window, to *write* (see Sect. 6.2.4 on page 50) into fields of that window (which thereby is updated – to become window′), to, most likely *put* (see Sect. 6.2.5 on page 52) that window back into the (global) forest of domain frames (fodf), and, finally to *close* (see Sect. 6.2.2 on page 48) the window′.

80. The open window operation, int_WFOpnW(mkWFOpnW(p,wn,kv))(fowf)(fodf), is partial.

Precondition:

a) The path p⌢⟨wn⟩ must be a path of the forest of window frames fowf.

Postcondition:

b) The set ps′ of paths of d′ equals the set ps of paths of p with the addition of the path to the newly opened window.

c) A window, w, is obtained from the appropriate path location in the domain frame with

d) the result window frame arising from the insertion of, possibly a key-value-modified version of that window in the window frame.

value
80. int_WFOpnW: WFOpnW → FoWF → FoDF ~→ FoWF
80. int_WFOpnW(mkWFOpnW(p,wn,kv))(fowf)(fodf) **as** fowf′
80. **let** ps=paths(fowf),ps′=paths(fowf′) **in**
80a. **pre** p ∈ ps ∧ p^⟨wn⟩ ∉ ps
80b. **post** ps′ = ps ∪ {p^⟨wn⟩}
80c. ∧ **let** w = eval_DFGetW(mkDFGW(p^⟨wn⟩))(fodf) **in**
80d. fowf′ = WF_Insert_W(p^⟨wn⟩,kv,w)(fowf) **end**
80. **end**

81. insert_W is an auxiliary function whose purpose it is to insert a possibly a key-value-modified version of ab argument window at an argument specificied path location in the window frame.

a) Same precondition as for the calling function (80a.).
b) The set ps′ of paths of fowf′ equals the set ps of paths of fowf with the addition of the path to the newly opened window – as also expressed in the postcondition of the calling function (80b.).
c) Paths common to fowf and fowf′, that is, in ps, designate the same window frames in both fowf and fowf′.
d) The window at location p^⟨wn⟩ in fowf′ –
e) with the window and key name as expected –
f) is a possibly key-value modified version of the argument window (which was "copied" from location p^⟨wn⟩ in the domain),
g) The window frame at p^⟨wn⟩ in fowf′ is empty.

81. WF_Insert_W: P0 × KVAL × W → FoWF ~→ FoWF
81. WF_Insert_W(p^⟨wn⟩,kv,w)(fowf) **as** fowf′
81. **let** ps=paths(fowf),ps′=paths(fowf′) **in**
81a. **pre** p^⟨wn⟩ ∉ ps
81b. **post** ps′ = ps ∪ {p^⟨wn⟩}
81c. ∧ ∀ p:P • p ∈ ps ⇒ s_WF(p,fowf)=s_WF(p,fowf′)
81. ∧ **let** (wn′,wtyp,mkWV(kn,tpl,rel)) = w **in**
81f. **let** w′ = (wn,wtyp,mkWV(kn,sel_flds(kv,rel,wtyp),rel)) **in**
81e. wn=wn′ ∧ **dom** kv = kn
81d. w′ = s_W(p^⟨wn⟩,fowf′)
81g. ∧ s_DF(p^⟨wn⟩,fowf′) = [ ]
81d. **end end end**

6.2.2 Close Window Frame

We saw, in Sect. 5.2.3, Items 73–73e (Page 41), how the *remove domain frame* operation can delete an entire 'cactus stack' of domain frames[13]. The remove window frame operation, now to be defined, will follow the same design principle. Whereas the windows populating such a 'cactus stack' of window frames have first come from the domain frame sub-system we shall, also as a design principle, not "save" (i.e., bring "back") the windows of such a removed window frame. The user is, if such savings (restorations) be needed, to first perform an appropriate number of *put* window operations.

82. The close window operation, int_DFCloW(mkDFCloW(p,wn))(fowf), is partial.

    Precondition:

    a) The path $\mathsf{p}^\frown\langle\mathsf{wn}\rangle$ must be a path of the forest of window frames fowf.

    Postcondition:

    b) First the change with respect to paths is that ps' is equal to ps with all those paths, p, in ps,

       (Cf. Item 73(c)i on page 41.) which are a proper prefix of paths, $\{\mathsf{p}^\frown\langle\mathsf{wn}\rangle\}$, in ps'

       (Cf. Item 73(c)ii on page 41.) removed as a result of removing the window frame designated by $\{\mathsf{p}^\frown\langle\mathsf{wn}\rangle\}$ in fowf.

    c) Then all paths common to ps and ps', that is, all paths of ps' designate the same domains in fowf and fowf'

The reader is encouraged to compare the *remove domain frame* and the *close window frame* operation: Sect. 5.2.3 on page 41 versus this section.

```
value
82.  int_WFCloW: WFCloW → FoWF ~→ FoWF
82.  int_WFCloW(mkWFCloW(p,wn))(fowf) as wf′
82.    let ps = paths(fowf), ps′ = paths(fowf′) in
82a.   pre p^⟨wn⟩ ∈ ps
82b.   post ps′ = ps \ rm_paths(ps)(p^⟨wn⟩) ∧ {p^⟨wn⟩} ∉ ps
82c.        ∧ ∀ p′:P•p′ ∈ ps′•s_WF(p′,fowf)=s_WF(p′,fowf′)
82.    end
```

[13] It might be considered "bad programming" to do so, but there you are.

6.2.3 Click Window

The intention of the click operation int_WFClkW(mkWFClkW(p,wn,fp))(fowf) is to reflect that not only has a cursor been moved to a well-defined position of the window (i.e., screen) but also clicked on that position. The "click" is to indicate that a subsequent *write* operation "writes" a given field value into that position. The user may, instead of an immediately subsequent *write* operation, decide to follow any *click* operation by other than *write* operation.

83. The click on window operation, int_WFClkW(mkWFClkW(p,wn,fp))(fowf), is partial.

Precondition:

a) The path p⌢⟨wn⟩ must be a path of the window frame fowf.

b) The field position, fp, of the argument must be a proper position within the window selected by p⌢⟨wn⟩.

Postcondition:

c) The paths of the forest of window frames is unchanged.

d) All paths common to ps (except path p⌢⟨wn⟩) and ps′, that is, all paths of ps′ (except path p⌢⟨wn⟩) designate the same window frames in fowf and fowf′.

e) Path p⌢⟨wn⟩ designates window mkWV(wnb,wtyp,flds,frel,wns) in fowf and window mkWV(wna,wtyp′,flds′,frel′,wns′) in fowf′ where, pairwise, wnb (before, in fowf) and wna (after, in textsffowf), wtyp and wtyp′, flds and flds′, frel and frel′ and unchanged, i.e., the same.

f) The field position, fp, designates appropriate window positions in w and w′.

g) The cursor, c, of the window states, wσ, becomes c′=fp of window state wσ′.

h) and where the field value, fv, of window w becomes fv′, the feld value at the position designated by fp in window w′.

value

```
83.  int_WFClkW: WFClkW → FoWF ~→ FoWF
83.  int_WFClkW(mkWFClkW(p,wn,fp))(fowf) as fowf′
83.    let ps = paths(fowf), ps′ = paths(fowf′),
83a.   pre  {p⌢⟨wn⟩} ∈ ps
83.         ∧ let (wσ:(w,fv,c),wfpwn) = s_Frame(p⌢⟨wn⟩,fowf),
83.           let mkWV(wnb,wtyp,flds,frel,wns) = w in
83b.          appropriate_FPos(fp,flds,wns)
83c.   post ps′ = ps \ rm_paths(ps)(p⌢⟨wn⟩) ∧ p⌢⟨wn⟩∉ ps
83d.         ∀ p′:P•p′ ∈ ps′⇒s_Frame(p′,fowf)=s_Frame(p′,fowf′)
83.         ∧ let (wσ′:(w′,fv′,c′),wfpwn′) = s_Frame(p⌢⟨wn⟩,fowf′) in
```

```
83.          let mkWV(wna′,wtyp′,flds′,frel′,wns′) = w′ in
83e.         wnb=wna∧wtyp=wtyp′∧flds=flds′∧frel=frel′∧wns=wns′
83f.       ∧ appropriate_FPos(fp,flds′,wns′)
83g.       ∧ c′=fp
83h.       ∧ fv′ = select_value(flds′)(fp)
83.    end end end end end
```

```
83b.,83f.   appropriate_FPos: FPos × Fields × WNm-set → Bool
83b.,83f.   appropriate_FPos(fp,flds,wns) ≡
83b.,83f.     case fp of
83b.,83f.       mkCNmIx(cnm,x)
83b.,83f.         → appropriate_FPos(mkCNm(cnm),flds,wns)
83b.,83f.           ∧ x ∈ inds flds(mkCNm(cnm)),
83b.,83f.       _ → fp ∈ dom flds ∪ wns
83b.,83f.     end
```

```
83d.  select_value: Fields → FPos ~→ FV
83d.  select_value(flds)(fp) ≡
83d.    case fp of
83d.      mkCNmIx(cnm,x) → (flds(mkCNm(cnm)))(x),
83d.      _ → flds(fp)
83d.    end
```

6.2.4 Write Window

84. The write window operation, int_WFWrW(mkWFWrW(p,wn,fv))(fowf), is partial.

Precondition:

a) The path $\mathsf{p}^\frown\langle\mathsf{wn}\rangle$ must be a path of the forest of window frames fowf.

b) If the cursor position must be appropriate.

c) The type of the value, fv, to be inserted must be a sub-type of the field in which it is to be inserted.

Postcondition:

d) The forest of window frame structure is unchanged.

e) All paths except, the path to the designated window being written upon, designate unchanged window frames.

f) The window being written upon does not change name, type, field names, relation or sub-window name set.

g) Only a tuple field (including a key value) is updated.

h) The cursor does not move.

```
value
84.  int_WFWrW: WFWrW → FoWF ~→ FoWF
84.  int_WFWrW(mkWFWrW(p,wn,fv))(fowf) as fowf′
84.     let ps = paths(fowf), ps′ = paths(fowf′),
84a.    pre  p⌢⟨wn⟩ ∈ ps
84.         ∧ let (wσ:(w,fv,c),wfpwn) = s_Frame(p⌢⟨wn⟩,fowf),
84.           let mkWV(wnb,wtyp,flds,frel,wns) = w in
84b.           appropriate_FPos(fp,fields,{})
84c.         ∧ sub_type(xtr_typ(fv),wtyp(c)) [check!]
84d.    post ps′ = ps
84e.         ∧ ∀ p′:P•p′ ∈ ps′\{p⌢⟨wn⟩}⇒s_Frame(p′,fowf)=s_Frame(p′,fowf′)
84.          ∧ let (wσ′:(w′,fv′,c′),wfpwn′) = s_Frame(p⌢⟨wn⟩,fowf′) in
84.            let mkWV(wna′,wtyp′,flds′,frel′,wns′) = w′ in
84f.           wnb=wna ∧ wtyp=wtyp′ ∧ dom flds=dom flds′ ∧ frel=frel′ ∧
84g.         ∧ flds′ = update_field(flds,c,fv)
84h.         ∧ c′=c
84.     end end end end end
```

85. One field, designated by the current cursor position is updated with the argument value, fv.

    Precondition:

    a) The cursor designates a proper field within the tuple of the window.

    Postcondition:

    b) For each field name, whether of a key or of the remaining tuple is either unchanged or changed, and only one is changed.

    c) If the cursor designates a curtain value, then fv must be a curtain value which becomes the new curtain value.

    d) If the cursor designates a curtain element value, then fv must be an atomic value which becomes the new curtain element value.

    e) If the cursor designates an atomic icon, then fv must be an atomic value which becomes the new icon value.

    f) Otherwise field values are unchanged.

```
value
85.  update_field: Fields × Cursor × FVAL → Fields
85.  update_field(flds,c,fv) as flds′
85a.    pre  appropriate_FPos(c,flds,{})
85b.    post ∀ fn:FNm•fn ∈ dom flds ⇒
85.        case (c,fv,fn) of
85c.          (mkCNm(n),mkCV(_),mkCNm(n)) → flds′(c)=fv,
85d.          (mkCNmIx(n,x),_,mkCNm(n)) → select_value(flds′)(c)=fv,
```

85e. (mkANm(n),_,mkANm(n)) → flds′(c)=fv,
85f. _ → flds(fn)=flds′(fn)
85. **end**

6.2.5 Put Window

86. The put window operation, int_WFPutW(mkWFPW(p,wn))(fowf)(fodf), is partial.

Precondition:

a) The path p⌢⟨wn⟩ must be a path of the forest of window frames fowf.

Postcondition:

b) The forest of window frames is unchanged.
c) Let w be the window
d) being put in the domain frame sub-system which thereby "changes" state to fodf″.
e) The (only) effect of this window frame operation is that the domain frame has changed to fodf′ = fodf″.

value
86. int_WFPutW: WFPutW → FoWF → FoDF → (FoDF × FoWF)
86. int_WFPutW(mkWFPW(p,wn))(fowf)(fodf) **as** (fodf′,fowf′)
86. **let** ps = paths(fodf) **in**
86a. **pre** {p⌢⟨wn⟩} ∈ ps
86b. **post** fowf′ = fowf
86c. ∧ **let** w = s_W(p⌢⟨wn⟩,wf) **in**
86d. **let** fodf″ = int_DFPutW(mkDFPW(p,wn,w))(fodf) **in**
86e. fodf′ = fodf″
86. **end end end**

"Life is like a sewer ..."

- **2 Theorems:**
  - **A General:**
    - An earlier version of the window, named wn at *path* position p in wf put "back" into the domain frame system df:
      int_WFPutW(mkWFPW(p,wn))(fowf′)(fodf′)
    - originated from that position, specifically:
      let (fodf′,fowf′) = int_WFOpnW(mkWFOpnW(p,wn,kv))(fowf)(fodf) **in** ... **end**

    * That is:
      p=p′, wn=wn′
  - **A Specific:**
    * **let** (fodf',fowf') = int_WFOpnW(mkWFOpnW(p,wn,kv))(fowf)(fodf) **in**
    * **let** (fodf'',fowf'') = int_WFOpnW(mkWFOpnW(p,wn,kv))(fowf')(fodf') **in**
    * **let** (fodf''',fowf''') = int_WFPutW(mkWFPW(p,wn))(fowf'')(fodf'') **in**
    * **let** (fodf'''',fowf'''') = int_WFCloW(mkWFCloW(p,wn))(fowf''')(fodf''') **in**
    * fowf'=fowf''=fowf'''=fowf''''=fowf ∧ fodf'=fodf''=fodf'''=fodf''''= fodf
    * **end end end end** **to be checked !**

6.2.6 Select Tuple

87. The int_WFSelTpl(mkWFSel(p,wn))(fowf) operation is partial.

Precondition:

a) The path to the window to be updated is in the paths of the forest of window frames.

Postcondition:

b) The paths of the before and after forests of window frames are unchanged.

c) For all paths, other than to the window affected, the forest of window frames are unchanged.

d) The selected "before" and "after" windows are almost the same,

e) with the exception of the tuple value: it is that of the argument key-value joined with the "remainder" tuple value obtained from the relation with the proviso that if the relation does not associate the current key-value to a "remaining" tuple then an appropriate such nil-tuple is constructed.

```
value
87.  int_WFSelTpl: WFSelTpl → FoWF → FoWF
87.  int_WFSelTpl(mkWFSel(p,wn))(fowf) as fowf′
87.    let ps = paths(fowf), ps′ = paths(fowf′), pn=p^⟨wn⟩ in
87a.    pre  pn ∈ ps
87b.    post ps = ps′
87c.      ∧ ∀ p:P•p ∈ ps\{pn} ⇒ s_WF(p,fowf)=w_WF(p,fowf′)
87.       ∧ let w = s_W(pn,fowf), w′ = s_W(pn,fowf) in
87.         let (wn′,rt,mkWV(kn,tpl,rel,wns))=w,
87.             (wn″,rt′,mkWV(kn′,tpl′,rel′,wns′))=w′
87.             kv = [ fn↦tpl(fn)|fn:FNm•fn ∈ kn ],in
87d.          wn′=wn″=wn ∧ rt=rt′ ∧ kn=kn′ ∧ rel=rel′ ∧ wns=wns′
87e.        ∧ tpl′ = if kv ∈ dom rv
87e.                 then kv ∪ rv(kv)
87e.                 else kv ∪ nil_FVAL(rt\dom kn)
87.    end end end end
```

6.2.7 Include Tuple

88. The int_WFIncTpl(mkWFInc(p,wn))(fowf) operation is partial.

Precondition:

a) The path to the window to be updated is in the paths of the forest of window frames.

Postcondition:

b) The paths of the before and after window frames are unchanged.

c) For all paths, other than to the window affected, the window frames are unchanged.

d) The selected "before" and "after" windows are almost the same,

e) with the exception of the relation value: it is either overwritten by or extended with the association of the current key value with the current "remaining" tuple value.

```
value
88.  int_WFIncTpl: WFIncTpl → FoWF → FoWF
88.  int_WFIncTpl(mkWFInc(p,wn))(fowf) as fowf′
88.     let ps = paths(fowf), ps′ = paths(fowf′), pn=p⌢⟨wn⟩ in
88a.     pre  pn ∈ ps
88b.     post ps = ps′
88c.       ∧ ∀ p:P•p ∈ ps\{pn} ⇒ s_WF(p,fowf)=w_WF(p,fowf′)
88.        ∧ let w = s_W(pn,fowf), w′ = s_W(pn,fowf) in
88.            let (wn′,rt,mkWV(kn,tpl,rel,wns))=w,
88.                (wn″,rt′,mkWV(kn′,tpl′,rel′,wns′))=w′,
88.                kv′ = [ fn↦tpl(fn)|fn ∈ kn ] in
88d.             wn′=wn″=wn ∧ rt=rt′ ∧kn=kn′ ∧ rel=rel′ ∧ wns=wns′
88e.          ∧ rel′ = rel † [ kv′↦tpl\dom kv′ ]
88.     end end end
```

6.3 Discussion

It seems that many of the remarks made in Sect. 5.3.1 also apply here.

7 A Simple Transaction System transactions

transactions

In this section we shall present a simple, highly "idealised" transaction system. The simplification is that we consider only one forest of domain frames process but n window frame processes all operating in "perfect" harmony, i.e., without any mutual coordination. In the next section, Sect. 8, we shall unravel a version of the present section's "naïve" model, one which at least coordinates transactions according to a *two phase commit protocol*. Failures of equipment: hardware and communication will only be considered in the subsequent section, Sect. 9. It is here we show careful narratives and formalisations of such techniques as rollbacks and roll-forwards.

In this section we consider the pair of a *forest of domain frames* and a *forest of window frames* to be processes. One *forest of domain frames process* and n *forest of window frames processes*. The *forest of domain frames process* is able at its own cognition to perform all the forest of domain frame operations as well as honouring requests from any forest of window frames process ($i : \{1...n\}$) for obtaining windows (Get Window) and storing windows (Put Window).

7.1 What Is a Transaction ?

A transaction is also referred to as a *unit of work*.

89. A forest of window frames or a forest of domain frames operation is a simple transaction.

90. A general transaction is either a simple transaction

    a) or the sequential combination of two or more general transactions

    b) or the parallel combination of two or more general transactions.

That is, these operations may occur sequentially, in some order, or concurrently, or both, that is, some sequentially, some concurrently. Let $\tau_i, \tau_j, \ldots, \tau_k$ designate transactions, then

$$(\tau_1; \ || \{\tau_{2_1}, (\tau_{2_{2_1}} ; \tau_{2_{2_2}} ; \ldots ; \tau_{2_{2_m}}), \ldots, \tau_{2_n}\} ; \tau_3)$$

designates a transaction.

7.1.1 Transaction Syntax

type
89. ST = WFCmd | DFCmd
90. GT = ST | QT | PT
90a. QT == mkQT(s_qt:GT*)
90b. PT == mkPT(s_pt:GT-**set**)

7.1.2 On Transaction Semantics

91. Let $\mathcal{M}_{ST}$ and $\mathcal{M}_{GT}$ stand for the meaning function of simple, respective general transactions. The meaning function of simple transactions were defined in Sects. 5 and 6

Then the meanings, $\mathcal{M}_{QT}$ and $\mathcal{M}_{PT}$, of sequential and concurrent transactions are:

91. $\mathcal{M}_{ST}, \mathcal{M}_{GT}, \mathcal{M}_{QT}, \mathcal{M}_{PT}$: ... → ...
90a. $\mathcal{M}_{QT}(\langle t_1, ..., t_{n-1}, t_n \rangle)(...) \equiv \mathcal{M}_{;}(\mathcal{M}_{GT}(t_n)(\mathcal{M}_{GT}(t_{n-1})(...(\mathcal{M}_{GT}(t_1)(...)))))$
90b. $\mathcal{M}_{PT}(\{t_1, t_2, ..., t_n\})(...) \equiv \mathcal{M}_{||}\{\mathcal{M}_{GT}(t_1)(...), \mathcal{M}_{GT}(t_2)(...), ..., \mathcal{M}_{GT}(t_n)(...)\}$

We shall first define the $\mathcal{M}_{GT}$ meaning function in this section (Sect. 7) — although not based on the above syntax for general transactions. Section 8 will redefine the $\mathcal{M}_{GT}$ meaning function — basically returning to the above syntax for general transactions.

7.2 An Analysis

We have defined, for domain frames and for window frames a number of operations. Their signatures are listed in the next paragraphs.

7.2.1 Domain Frame Elaboration (Function) Signatures

70. int_DFIniDF: DFIniDF → FoDF
71. int_DFCreDF: DFCreDF → FoDF $\xrightarrow{\sim}$ FoDF
73. int_DFRmDF: DFRmDF → FoDF $\xrightarrow{\sim}$ FoDF
76. int_DFPutW: DFPutW → FoDF $\xrightarrow{\sim}$ FoDF
77. eval_DFGetW: DFGetW → FoDF $\xrightarrow{\sim}$ W

7.2.2 Window Frame Elaboration Function Signatures

80. int_WFOpnW: WFOpnW → FoWF → FoDF $\xrightarrow{\sim}$ FoWF
82. int_WFCloW: WFCloW → FoWF $\xrightarrow{\sim}$ FoWF
83. int_WFClkW: WFClkW → FoWF $\xrightarrow{\sim}$ FoWF
84. int_WFWrW: WFWrW → FoWF $\xrightarrow{\sim}$ FoWF
86. int_WFPutW: WFPutW → FoWF → FoDF → (FoDF × FoWF)
87. int_WFSelTpl: WFSelTpl → FoWF → FoWF
88. int_WFIncTpl: WFIncTpl → FoWF → FoWF

7.2.3 Window Frame to Domain Frame Invocations

Of the above referenced forest of window frames operations only two involve the forest of domain frames. Their interface invocations are:

value
80. int_WFOpnW: WFOpnW → FoWF → FoDF $\xrightarrow{\sim}$ FoWF
80. int_WFOpnW(mkWFOpnW(p,wn,kv))(fowf)(fodf) **as** fowf′
80. ...
80c. ... eval_DFGetW(mkDFGW(p^⟨wn⟩)(fodf))
80. ...

and:

value
86. int_WFPutW: WFPutW → FoWF → FoDF → (FoDF × FoWF)
86. int_WFPutW(mkWFPW(p,wn))(fowf)(fodf) **as** (fodf′,fowf′)
86. ...
86d. ... int_DFPutW(mkDFPW(p,wn,w))(fodf)
86. ...

7.2.4 Changes

- The forest of domain frames process shall have a local variable, fodfv, replacing the need for an interpretation function argument (fodf).
- Each forest of window frames process shall have a local variable, fowfv, replacing the need for an interpretation function argument (fowf).
- The int_WFOpnW(mkWFOpnW(p,wn,kv))(fowf)(fodf) function is interpreted in the window frame processes.
  - The function invokes the eval_DFGetW(mkDFGW(p,wn))fodf) function which is evaluated in the forest of domain frames process.
  - That process yields, i.e., communicates, a window w.
  - Hence two communications shall be offered:
    - A request from a forest of window frames process to the forest of domain frames process to perform eval_DFGetW(mkDFGW(p)(fodf);
    - followed by the domain process "returning" (hopefully) the window w (or else that an `"error"` has occurred).
- The int_WFPutW(mkWFPW(p,wn))(fowf)(fodf) function is interpreted in one of the forest of window frames processes.
  - The function invokes the int_DFPutW(mkDFPW(p,wn,w))(fodf) function which is interpreted in the forest of domain frames process,

– which yields a side-effect on that forest of domain frames process state.
– Hence two communications shall be offered:
  * The request from a forest of window frames process to the forest of domain frames process to perform the int_DFPutW(mkDFPW(p,wn,w))(fodf).
  * The reply from the domain process that the request was honoured, that is, `"ok"`, or that something went wrong, that is, `"error"`.

7.3 The System

92. Channels are indexed by an otherwise undefined quantity.
93. The system process is the parallel composition of one (i.e., a) domain_frame process and the parallel composition of a number of window_frame processes.
94. A domain_frame process ...
95. A window_frame process ...

type
92. WIdx
value
92. wis:WIdx-**set**

93. system: WIdx-**set** → **Unit**
93. system(cxs) ≡ domain_frame(...) ‖ (‖{window_frame(wi,...)|wi:WIdx•wi ∈ wis})
94. domain_frame: DF → **in**,**out** ... **Unit**
95. window_frame: i:WIdx × WF → **in**,**out** ... **Unit**

7.3.1 Channels

92. Channels are indexed by an otherwise undefined quantity.
96. Channels express willingness to accept and offer messages of, for the moment, further unspecified nature.
97. The is an array of channels.

type
92. WIdx
96. MSG
channel
97. {ch[wi]|wi:WIdx•wi ∈ wis}:MSG

7.3.2 The System Process

98. There is an initial association, fowfs:FoWFS, window frame indexes into initial, not necessarily identical forests of window frames.

99. The overall system process definition can now be completed.

type
98. FoWFS = WIdx $\xrightarrow{m}$ FoWF
value
98. wfs:WFS
99. system: WFS → **in**,**out** {ch[wi]|wi:WIdx•wi ∈ wis} **Unit**
99. system(fowfs) ≡ domain_frame() ‖ (‖{window_frame(wi,fofws(wi))|wi:WIdx•wi ∈ wis})

7.3.3 The Forest of Domain Frames Process

The forest of domain frames process, at its own volition, "alternates" between honouring either of the five domain frame operations or accepting either of two requests from any forest of window frames process.

100. To help determine which of these seven alternatives a command "kind", cmd:Cmd, of six alternative "tokens" is defined. One of these tokens, inut (for input/output), stands for "willingness to accept either of two inputs from any forest of window frames process.

101. The forest of domain frames process takes no argument and

102. "cycles forever".

103. The elaboration function parameter, fodf, is provided by the contents of an assignable domain frame valued variable, fodfv – which is initialised to an initial forest of domain frames (cf. 70 on page 40).

104. In each step (well, cycle) of the forest of domain frames process a non-deterministic internal choice is made (is taken) as to which kind of operation to perform.

105. If the choice is to re-initialise the domain frame then that is done.

106. If the choice is

    a) to create a domain frame, or
    b) to remove a domain frame, or
    c) to put a window (into a domain frame), or
    d) to obtain (get) a window from a domain frame,

then a corresponding command is internally non-deterministically chosen such that this command satisfies the **pre** condition for the corresponding interpretation function — and the forest of domain frames variable is set to the result of the corresponding operation.

107. If the choice is to be willing to accept a request from a forest of window frames process
    a) then the forest of domain frames process external non-deterministically ([]) choose to accept from any forest of window frames process, i:WIdx,
    b) either a get window or a put window request;
    c) if the request satisfies the precondition of the corresponding forest of domain frames elaboration function,
    d) then that operation is performed and its result communicated back to the requesting forest of window frame process, or
    e) else an `"error"` message is returned.

```
type
100.  Cmd == init | crea | rmdf | putw | getw | inut
value
101.  forest_of_domain_frames: Unit → in,out {ch[wi]|wi:WIdx•wi ∈ wis}  Unit
101.  forest_of_domain_frames() ≡
103.    variable fodfv:FoDF := int_iniFoDF();
102.    while true do
104.      let cmd = init⌈⌉crea⌈⌉remv⌈⌉put⌈⌉get⌈⌉inut in
101.      case cmd of
105.       init → fodfv := elab_FoDFinit(),
106a.      crea → fodfv := elab_FoDFCre(c fdfv),
106b.      rmdf → fodfv := elab_FoDFRmv(c fdfv),
106c.      putw → fodfv := elab_FoDFPutW(c fdfv),
106d.      getw → fodfv := elab_FoDFGetW(c fdfv),
107.       inut → interaction()
100.    end end end

105.  elab_FoDFinit: Unit → FoDF
105.  elab_FoDFinit() ≡ int_DFiniDF()

106a.  elab_DFCre: FoDF → FoDF
106a.  elab_DFCre(fodf) ≡
106a.    if ∃ cdf:DFCreDF • pre:int_DFCreDF(cdf)(fodf)
106a.      then let cdf:DFCreDF • pre:int_DFCreDF(cdf)(fodf) in
106a.             int_DFCreDF(mkDFCDF(p,wn))(fodf) end
106a.      else df end
```

```
106b.  elab_FoDFRmv: FoDF → FoDF
106b.  elab_FoDFRmv(fodf) ≡
106b.    if ∃ rdf:DFRmDF • pre:int_DFRmDF(rdf)(fodf) in
106b.      then let rdf:DFRmDF • pre:int_DFRmDF(rdf)(fodf) in
106b.            int_DFRmDF(cdf)(fodf) end
106b.      else df end

106c.  elab_DFPutW: FoDF → FoDF
106c.  elab_DFPutW(fodf) ≡
106c.    if ∃ put:DFPutW • pre:int_DFPutW(put)(fodf) in
106c.      then let put:DFPutW • pre:int_DFPutW(put)(fodf) in
106c.            int_DFPutW(put)(df) end
106c.      else df end

106d.  elab_DFGetDF: FoDF → FoDF
106d.  elab_DFGetDF(fodf) ≡
106d.    if ∃ get:DFGetW • pre:eval_DFGetW(get)(fodf) in
106d.      then let get:DFGetW • pre:eval_DFGetW(get)(fodf) in
106d.            eval_DFGetW(get)(fodf) end
106d.      else df end

107.   interaction: FoDF → FoDF
107.   interaction(fodf) ≡
107.     variable lfodfv:FoDF
107a.    []{let req = ch[i]? in
107a.       case req of
107b.         mkDFGW(p) →
107c.           if pre: eval_DFGetW(mkDFGW(p))(fodf)
107d.             then ch[i] ! eval_DFGetW(mkDFGW(p))(fodf)
107e.             else ch[i] ! "error" end ; chaos,
107b.         put →
107c.           if pre:int_DFPutW(put)(c vdf)
107d.             then lfodfv := int_DFPutW(put)(df);ch[i]!"ok"
107e.             else ch[i]!"error" ; chaos end
107a.       end | i:WIdx•i ∈ wis end} ; lfodf
```

7.3.4 The Forest of Window Frames Processes

Forest of window frames processes, at their own volition, "alternates" between honouring either of the window frame operations – two of which requires interaction with the forest of domain frames process.

108. To help determine which of these seven alternatives a command "kind", cmd:Cmd, of sevem alternative "tokens" is defined.

109. The forest of window frames process takes no argument and

110. "cycles forever".

111. The elaboration function parameter, fowf, is provided by the **c**ontents of an assignable forest of window frames-valued variable, fowfv – which is initialised to a "pre-set" initial forest of window frames value (cf. 98 on page 59).

112. In each step (well, cycle), of a forest of window frames process, a non-deterministic internal choice, ⌈⌉, is made as to which kind of operation to perform.

113. If the choice is to *open* a window then an open command is internally non-deterministically chosen such that this command satisfies the **pre** condition for the open window operation;

    a) the forest of window frames process communicates a get window command request (mkDFGW(p⌢⟨wn⟩)) to the forest of domain frames process and awaits a response.

    b) If the forest of domain frames process could not find the window at the communicated path position then an `"error"` result is received and **chaos** ensues.

    c) If the forest of domain frames process does find the (a) window, w, at the communicated path position then it is returned (from the forest of domain frames process to the communicating forest of window frames process) and that window, w, is *inserted* at the chosen path position – by means of the auxiliary function: int_Insert_W(p⌢⟨wn⟩,kv,w)(fowf).

114. If the choice is to *put* the current window back into the domain frame

    a) then a corresponding command is internal non-deterministically chosen such that this command satisfies the **pre** condition for the put window operation;

    b) the window process communicates a mkDFGW(p,wn) request to the domain process and receives, in turn a result;

    c) if the result is `"error"`, that is, the domain process could not find a window at the designated path location, p, then **chaos** ensues, otherwise nothing – the window frame state is unchanged.

115. If the choice is

    a) to `close` a window, or

    b) to `click` on a window, or

    c) to `write` onto a window, or

    d) to `select` a tuple from the current window relation, or

    e) to `include` the current tuple with the current key-value in that relation,

then a corresponding command is internal non-deterministically chosen such that this command satisfies the **pre** condition for the corresponding interpretation function —

116. and the window frame variable is set to the result of the corresponding operation.

```
type
108.  Cmd = opn | put | clk | wri | sel | inc | clo
value
109.  forest_of_window_frames: i:WIdx × FoWF → in,out ch[i] Unit
109.  forest_of_window_frames(wi,fowf) ≡
111.    variable fowfv:FoWF := fowf;
110.    while true do
108.      let cmd = opn⌈clo⌈put⌈clk⌈wri⌈sel⌈inc in
112.      case cmd of
113.        opn → fowfv := elab_WFOpnW(wi,fowf),
114.        put → fowfv := elab_WFPutW(wi,fowf),
115b.        clk → fowfv := elab_WFClkW(fowf),
115c.        wri → fowfv := elab_WFWriW(fowf),
115d.        sel → fowfv := elab_WFSelW(fowf),
115e.        inc → fowfv := elab_WFIncW(fowf),
115a.        clo → fowfv := elab_WFCloW(fowf)
108.    end end end

113.  elab_WFOpnW: WIdx → FoWF → in,out ch[wi] FoWF
113.  elab_WFOpnW(wi)(fowf) ≡
113.    if ∃ ow:OpenW • pre:int_OpnW(ow)(wf) in
113.      then let mkOpnW(p,wn,kv):OpenW•pre:int_OpnW(mkOpnW(p,wn,kv))(fowf)
113a.           let w = ch[wi]!mkGW(p^⟨wn⟩) ; ch[wi]? in
113b.           if w=″error″ then chaos else skip end;
113c.           insert_W(p^⟨wn⟩,kv,w)(fowf) end end
114.      else wf end

114.  elab_WFPutW: WIdx → FoWF → in,out ch[wi] FoWF
114.  elab_WFPutW(wi)(fowf) ≡
114.    if ∃ putw•pre:int_WFPutW(puw)(fowf)
114.      then let mkWPW(p,wn):WFPutW •
114.                pre:int_WFPutW(mkWPW(p,wn))(fowf) in
114a.           let w = s_W(p^⟨wn⟩,fowf) in
114b.           let result = ch[wi]!mkDFPW(p^⟨wn⟩,wn,w) ; ch[wi]? in
114c.           if result = ″error″ then chaos else skip end
114.          end end end
114.      else fowf end

115b.  elab_WFClkW: FoWF → FoWF
115b.  elab_WFClkW(fowf) ≡
```

```
115b.    if ∃ clow:ClkW • pre:int_WFClkW(clow)(fowf)
115b.      then let clow:ClkW • pre:int_WFClkW(clow)(fowf) in
115b.          int_ ClkW(clow)(fowf) end
115b.      else wf end
115c.  elab_WFWriW: FoWF → FoWF
115c.  elab_WFWriW(fowf) ≡
115c.    if ∃ wriw:WFWrW • pre:int_WFWrW(wriw)(wfof)
115c.      then let wriw:WFWrW • pre:int_WFWrW(wriw)(fowf) in
115c.          int_WFWrW(wriw)(fowf) end
115c.      else wf end
115d.  elab_WFSelW: FoWF → FoWF
115d.  elab_WFSelW(fowf) ≡
115d.    if ∃ sel:Sel • pre:int_Sel(sel)(fowf)
115d.      then let sel:Sel • pre:int_Sel(sel)(fowf) in
115d.          int_ClkW(sl)(fowf) end
115d.      else wf end
115e.  elab_WFIncW: FoWF → FoWF
115e.  elab_WFIncW(fowf) ≡
115e.    if ∃ incl:WFIncTpl • pre:int_Inc(incl)(fowf)
115e.      then let ic:Inc • pre:int_Inc(ic)(fowf) in
115e.          int_ClkW(ic)(fowf) end
115e.      else wf end
115a.  elab_WFCloW: FoWF → FoWF
115a.  elab_WFCloW(fowf) ≡
115a.    if ∃ cl:CloW • pre:int_CloW(clo)(fowf)
115a.      then let cl:CloW • pre:int_CloW(clo)(fowf) in
115a.          int_CloW(clo)(fowf) end
115a.      else wf end
```

7.4 Discussion

There are (quite) a number of problems with the definition of the system process as composed from one family_of_domain_frames and n family_of_window_frames processes. In the following we will list some of these problems.

- Interference: While one forest of window frames process is operating on a number of windows obtained from the forest of domain frames process other forest of window frames processes put similarly named windows back into the forest of domain frames process where they were also first obtained.

  The independence of and lack of coordination between forest of window frames processes and their prompting their common forest of domain frames process to put windows back into the forest of domain frames' process "storage" is therefore the cause of domain frame data being "out of synchronisation" with window frame data.

- Ghost Windows: While one window process is "happily" operating upon windows these same window may have been deleted by forest of domain frames process 'delete' operations.
- :
- :
- :
- :

8 Coordinated Transaction Processing tp-system

tp-system

In this section we shall present a corordinated version of the model of the previous section. The new thing here is the use of *the two phase commit protocol*.

8.1 An Overview of A System of Processes

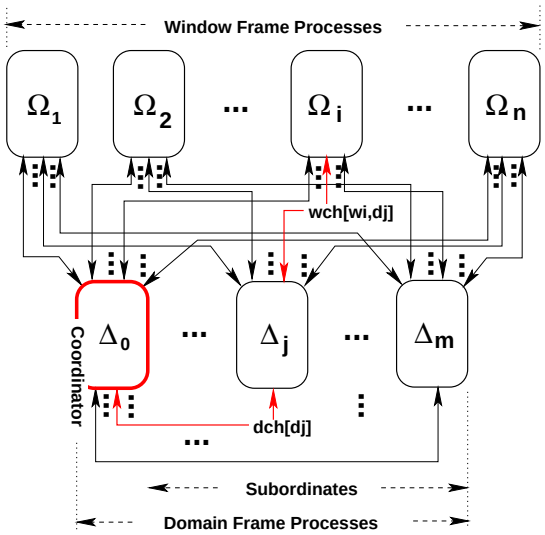


Figure 8: Window and Domain Frame Processors and Their Channels

Take a look at Fig. 8.

- There are $m+1$ domain frame processes, $\Delta_0, \Delta_1, \ldots, \Delta_j, \ldots, \Delta_m (m \geq 2)$.
  - Each domain frame process $\Delta_j, j \in \{1..m\}$ manages a number of domain frames, $dfs_j = \{\mathbb{DF}_{j_1}, \mathbb{DF}_{j_2}, \ldots, \mathbb{DF}_{j_\iota}\}$ (for $j \in \{1..m\}$).
  - ι may vary from domain frame process to domain frame process) such that no two domain frame processes, Δ_k, Δ_ℓ, manage the same domain frames, that is: $\mathsf{dfs}_k \cap \mathsf{dfs}_\ell = \{\}$.
  - Domain frame process Δ_0 is called the *coordinator process*.
  - Domain frame processes Δ_j (for $j \in \{1..m\}$) are called *subordinate processes* (or *cohorts*).

and

- There are n window frame processes, $\Omega_1, \Omega_2, \ldots, \Omega_i, \ldots, \Omega_n (n \geq 1)$, each Ω_i managing one window frame, $\mathbb{WF}_i$ (for $i \in \{1, 2, \ldots, n\}$).

- **States:** Each of the processes evolve around a state. The states differ. The coordinator state, $\Delta_0\Sigma$, keeps track of which subordinate forest of domain frame processes (i.e., their states, $\Delta_j\Sigma$) holds which domain frame windows. The forest of window state processes (i.e., their states, $\Omega_i\Sigma$) holds a copy of a global *window schema* which catalogues the sum total of all forest of domain frames (i.e., their windows) [without, however any information as to which subordinate process maintains those windows]. And the subordinate forest of domain frames processes (i.e., their states, $\Delta_j\Sigma$) not only holds the forest of domain frames windows, but also which such windows, if any, are, at any time, allocated to forest of window frames processes Ω_i. States are defined in Sect. 8.5 (Sects. 8.5.3–8.5.2). But first the notion of *window schemas*, a notion which cuts across all four state notions, is defined in Sect. 8.3.1.
- **Channels:** The processes can synchronise and communicate over channels as indicated in Fig. 8 on the preceding page and otherwise outlined in Items 167a–167g (Pages 79–80). Channels are defined in Sect. 8.4.
- **Process Actions and Interactions:** The processes are defined in Sect. 8.6.

8.2 An Adaptation of The Two Phase Commit Protocol 2pc

2pc

8.2.1 A First Overview Narrative of the Two Phase Commit Protocol

117.

118.

119.

120.

121.

122.

123.

8.2.2 A Second Narrative of the Two Phase Commit Protocol

We shall adopt the *two phase commit* (transaction processing) protocol [35, 36, 26, 37] and adapt it to the the system of processes as outlined in the previous section, i.e., Sect. 8.1. There are many variants of the two phase commit protocol. Some optimise performance in one way, some in another way. Some handle machine (i.e., computer, including storage) or data communication crashes, in one way or another. Etcetera [32]. We shall only present a precise description in this section (i.e., Sect. 8.2) and a formalisation in the remainder of this overall section (i.e., Sect. 8). Using our formalisation approach the reader can then formalise any of the optimising or failure handling versions of the two phase commit protocol.

To follow the informal description of the two phase commit protocol, which now follows, the reader may be well adviced in using a large (A4 sheet) rendition of Fig. 8 on page 66 and marking it with, for example coloured and number attributed flow lines according to the very many enumerated items below !

124. There is a notion of a **coordinator** (or transaction) process. We decide to let domain process Δ_0 be the coordinator.

125. There is a notion of **subordinate** (or cohort) processes. We decide to let the forest of domain frames processes $\Delta_1 \ldots \Delta_m$ be the subordinates.

126. And there is a notion of **user** process. We decide to let the forest of window frames processes $\Omega_1 \ldots \Omega_n$ be the users.

127. There is a notion of *transaction*: a job to be done, usually by one or more other processes than the user. We decide that a transaction is the set of actions that, from the point of view of the user,

    a) *starts* with obtaining (*open*ing, *get*ting) path-designated windows from subordinates,
    b) *continues* through the handling of (*write* on) these windows by the users,
    c) and *completes* when the last of these requested (path-designated) windows have been returned (*close*/*put* windows) to the appropriate subordinate.

128. Users *request* the coordinator to coordinate that a number of one or more subordinates effect the transaction of the requested job.

129. A user sends a **"request transaction *'ps'*"** to the coordinator.

    A transaction request consists of a set, ***'ps'***, of paths to forest of domain frames windows.

2pc-co

130. The coordinator handles the transaction request as follows:

    a) The coordinator receives a **"request transaction *'ps'*"** [Item 129] from a user.
    b) The coordinator assigns a fresh, that is, hitherto unused, whence a *unique transaction identifier* τ to the transaction *'ps'* [Item 129].
    c) The coordinator finds, for each path, which subordinate forest of domain frames process holds (i.e. stores) the window designated by that path.
    d) Once that is done for all paths, the coordinator sends a **'**Ω_i **'commit request** τ ***'ps'*"** to each of subordinator so designated with inquiring whether they are free to handle exactly the paths (i.e., sending the designated windows to the requesting users, and waiting for their being returned).

e) Once all such designated subordinators have been sent the inquiry, the coordinator waits for their response.

f) Two possibilities now arise:

i. Either the coordinator receives a negative response, **"no, unable to handle τ"** [Item 132(a)iii on the next page], from such an inquired subordinator (before all have responded or as the last expected response).

1. In this case the coordinator sends an **"abort τ"** to all those subordinates which have either yet to respond or have responded positively.
2. The coordinator then sends a **"request τ: *'ps'* cannot be accepted"** message to the requesting user.
3. The coordinator then awaits **"abort τ acknowledgement"**s of subordinators [Item 132b] having received the abort message. Once all have acknowledged the coordinator terminates this transaction.

ii. Or all inquired subordinators have responded positively.

1. Now the coordinator sends a **"commit τ"** message to all subordinators (involved in this transaction);
2. then awaits their **"commit τ acknowledgement"**s [Item 132(c)i];
3. and once all such have been received, the coordinator sends a **"request τ: path index map: *'p_to_wi'*"** [Item 134(a)ii] confirmation to the requesting user with the following message:
   - the τ identifier,
   - and, for each requested path, the identifier, didx of the subordinator which handles that path window.
4. Then the coordinator ends its involvement with this transaction — leaving it to the user and the subordinators to continue and end their involvements (the subordinators accepting *get window* and *put window* commands, the user issuing these commands while, in-between, operating (*write*) on these windows.

2pc-su

131. The subordinators handles transaction requests from the coordinator and from the users.

- That is, in any one time interval a subordinator is engaged
  - with the coordinator concerning a transaction request, or
  - with any of a number of forest of window frames processes concerning a command (get/put) request.

- That is, the subordinators multiplex between the coordinator and the up to n user processes.

132. The subordinates handle the transaction request from the coordinator as follows:

a) When it receives a **"commit request** τ ***'ps'"*** [Item 130d] from the coordinator

i. then it examines [Item 132(a)i2 below] whether it is already committed the handling of certain **"request** τ ***'ps'"*** path-designated windows to other ongoing actions or not.

1. The subordinate performs this examination, for example, on the basis of a subordinate state component

   type
   132(a)i1. COM = UTId $\xrightarrow[m]{}$ P-**set**.

2. com:COM registers current commitments, and, for example, as follows:

   value
   132(a)i2. **if** ps ∩ ∪ **rng** com ≠ {} **then** inability **else** ability **end**

3. Each time a **"commit request** τ ***'ps'"*** is accepted the com:COM is augmented as follows:

   value
   132(a)i3. com ∪ [τ ↦ ps] ∈

4. Each time the subordinator receives a τ-related close window command for a given p∈ps it removes that path from com:COM:

   value
   132(a)i4. com † [τ ↦ com(τ) \ {p}]

5. Once com(τ) is empty, i.e., {}, the subordinate has completed its part of the τ transaction.

ii. If it cannot commit itself [Item 132(a)i2 just above] to the servicing of all the path designated windows then it sends an **"no, unable to handle** τ**"** response to the coordinator [Item 130(f)i] and "forgets" about τ.

iii. If it can pre-commit itself [Item 132(a)i2 just above] to the servicing of all the path designated windows, ps, then it sends an **"yes, able to handle** τ**"** response to the coordinator [Item 130(f)i3], and awaits further message [Item 130(f)i1 or Item 130(f)ii1 on the preceding page] from the coordinator.[14]

[14]This item (132(a)iii.) and Items 132(a)iii1.–132(a)iii2., defines the concept of *pre-commitment*

1. This pre-commitment is recorded in a pre-commit state component:

   type
   132(a)iii1. preCOM = UTId $\overrightarrow{m}$ P-**set**
   value
   132(a)iii1. precom ∪ [$\tau \mapsto$ ps]

2. If, as can be expected, the subordinate receives an **"abort τ"** message from the coordinator then the pre-commitment is abandoned:

   132(a)iii2. precom \ {τ}

b) If a subordinator receives an **"abort τ"** [Item 130(f)i3] then it sends an **"abort τ acknowledgement"** to the coordinator and "releases" a possible earlier commitment to handle as set, ps, of path designated windows (and "forgets" about τ).

c) If, instead, it receives a **"commit τ"** [Item 130(f)ii1], then it proceeds as follows:

   i. First it sends a **"commit τ acknowledgement"** to the coordinator.
   ii. Then it awaits τ-identified command requests from a user [Item 133 next] — it does so interleaved with other message handling, with the coordinator or with other users.
   iii. The subordinator sends no messages when it determines that a τ-transaction has been completed.

133. The subordinators handle transaction requests from users as follows:

a) These transaction requests are in the form of

   i. **"τ: get window '*p*'"**
   ii. **"τ: put window '*w*' at '*p*'"**

b) To the "τ: get window '*p*'" request the subordinator responds as follows:

   i. If it finds that path '*p*' is not a path of its forest of domain frames then it responds: **"τ: cannot find window at '*p*'"**.
   ii. If it finds that path '*p*' is a path of its forest of domain frames then it responds: **"τ: here is path '*p*' window: '*w*'"**.

c) To the "τ: put window '*w*' at '*p*'" request the subordinator responds as follows:

   i. If it finds that path '*p*' is not a path of its forest of domain frames then it responds: **"τ: cannot store window at '*p*'"**.
   ii. If it finds that path '*p*' is a path of its forest of domain frames then it responds: **"τ: path '*p*' window has been stored"**.

2pc-us

134. The users handle the transaction requests as follows:
    a) A user starts, [Items: 129→130a], with a **"request transaction *'ps'*"** to the coordinator.
        The user has decided upon the set, *'ps'*, of window-designating paths, by marking its copy of the *window schema*. This window schema catalogues the forest of domain frame windows of all forest of domain frames processes. The *marking* function is described later.
    In response to this the user awaits a response from the coordinator.
        i. Either the user receives a **"request τ: *'ps'* cannot be accepted"** rejection [Item 130(f)i2];
        ii. or the user receives a **"request τ: path index map: *'p_to_wi'*"** confirmation [Item 130(f)ii3].
    b) In the case of a "request τ: *'ps'* cannot be accepted" message the user waits some time or resigns.
    c) In the case of a "request τ: path index map: p_to_wi" confirmation the user proceeds as follows:
        i. In order to do anything on windows these must first be obatined. The user therefore sends (one or more) **"τ: get window *'p'*"** commands to the subordinator wi indicated by the index map: p_to_wi(p). Responses to these requests are:
            1. The subordinator either responds with: **"τ: cannot find window at *'p'*"** [Item 133(c)i],
            2. or it responds with: **"τ: here is path *'p'* window: *'w'*"** [Item 133(c)ii].
        ii. In the former case the user checks whether it has used an erroneous path. (We do not explain the consquences of doing so here.)
        iii. In the latter case the user proceeds.
    d) The user then operates on these windows now within it own forest of window frames.
    e) Terminitating its work on requested windows the user issues (one or more) **"τ: put window *'w'* at *'p'*"** commands to the subordinator wi indicated by the index map: p_to_wi(p). Responses to these requests are:
        i. If it finds that path *'p'* is not a path of its forest of domain frames then it responds: **"τ: cannot store window at *'p'*"**.
        ii. If it finds that path *'p'* is a path of its forest of domain frames then it responds: **"τ: path *'p'* window has been stored"**.
    f) In the former case the user checks whether it has used an erroneous path. (We do not explain the consquences of doing so here.)
    g) In the latter case the user proceeds to close windows and terminates the τ transaction.

8.2.3 Two Phase Commit Protocol Messages, a Resumé 2pc-msgs

2pc-msgs

135. Items 134a,129→130a: **"request transaction *'ps'*"** $\Omega_i \rightarrow \Delta_0$
136. [Item 204b] **"path(s) *'ps'* are not defined"** $\Delta_0 \rightarrow \Omega_i$
137. Item 130d: **"please, commit request τ *'ps'*"** $\Delta_0 \rightarrow \Delta_j$
138. Item 132(a)ii: **"no, unable to handle τ"** $\Delta_j \rightarrow \Delta_0$
139. Item 130(f)i1: **"abort τ"** $\Delta_0 \rightarrow \Delta_j$
140. Item 132b: **"abort τ acknowledgement"** $\Delta_j \rightarrow \Delta_0$
141. Item 132(a)iii: **"yes, able to handle τ"** $\Delta_j \rightarrow \Delta_0$
142. Item 130(f)i2: **"request τ: *'ps'* cannot be accepted"** $\Delta_0 \rightarrow \Omega_i$
143. Item 130(f)ii1: **"commit τ"** $\Delta_0 \rightarrow \Delta_j$
144. Items 132(c)i→130(f)ii2: **"commit τ acknowledgement"** $\Delta_j \rightarrow \Delta_0$
145. Item 130(f)ii3: **"request τ: path index map: *'p_to_wi'*"** $\Delta_0 \rightarrow \Omega_i$
146. Item 134(c)i→133(a)i: "τ**: get window *'p'*"** $\Omega_i \rightarrow \Delta_j$
147. Item 133(a)ii: "τ**: put window *'w'* at *'p'*"** $\Omega_i \rightarrow \Delta_j$
148. Item 133(b)i: "τ**: cannot find window at *'p'*"** $\Delta_j \rightarrow \Omega_i$
149. Item 133(b)ii: "τ**: here is path *'p'* window: *'w'*"** $\Delta_j \rightarrow \Omega_i$
150. Item 133(c)i: "τ**: cannot store window at *'p'*"** $\Delta_j \rightarrow \Omega_i$
151. Item 133(c)ii: "τ**: path *'p'* window has been stored"** $\Delta_j \rightarrow \Omega_i$

type
MSG = RTPS | IRPPSND | PCORTPS | NUTHT | AT | ATA |
YATHT | RTPSCBA | CT | CTA | RTPIM | TGWP |
TPWPW | CFWAP | HIPPW | CSWAP | PPWHBS

135. RTPS == mkRTPS(P-**set**)
"request transaction *'ps'*"
136. IRPPSND == mkIRPPSND(P-**set**)
"path(s) *'ps'* are not defined"
137. PCORTPS == mkPCORTPS(ϒ,P-**set**)
"please, commit request τ *'ps'*"
138. NUTHT == mkNUTHT(ϒ)
"no, unable to handle τ"
139. AT == mkAT(ϒ)
"abort τ"
140. ATA == mkATA(ϒ)
"abort τ acknowledgement"
141. YATHT == mkYATHT(ϒ)
"yes, able to handle τ"
142. RTPSCBA == mkRTPSCBA(UTid,P-**set**)
"request τ: *'ps'* cannot be accepted"
143. CT == mkCT(ϒ)
"commit τ"
144. CTA == mkCTA(ϒ)
"commit τ acknowledgement"
145. RTPIM == mkRTPIM(UTid,PIM)
"request τ: path index map: *'p_to_wi'*"
146. TGWP == mkTGWP(ϒ,P)
"τ: get window *'p'*"
147. TPWPW == mkTPWPW(ϒ,W,P)
"τ: put window *'w'* at *'p'*"
148. CFWAP == mkCFWAP(ϒ,P)
"τ: cannot find window at *'p'*"
149. HIPPW == mkHIPPW(ϒ,P,W)
"τ: here is path *'p'* window: *'w'*"
150. CSWAP == mkCSWAP(ϒ,P)
"τ: cannot store window at *'p'*"
151. PPWHBS == mkPPWHBS(ϒ,P)
"τ: path *'p'* window has been stored"

8.2.4 Analysis

This ends our informal, second description of our adaptation of the two phase commit protocol. We shall now, somewhat informally, analyse this narrative explication.

152. A user is free to pursue any activity between
    - [Item 134a] a **"transaction request *'ps'*"**

    and its receiving either
    - [Item 130(f)i2] a **"request τ: *'ps'* cannot be accepted"** reject or
    - [Item 130(f)ii3] an **"request τ: path index map: *'p_to_wi'*"** accept.

153. The coordinator between its receiving
    - [Item 134a] a **"transaction request *'ps'*"**

    and its terminating its involvement in the τ transaction:
    - [Item 130(f)i2] **"request τ: *'ps'* cannot be accepted"** $\Delta_0 \rightarrow \Omega_i$ or
    - [Item 130(f)ii3] **"request τ: path index map: *'p_to_wi'*"** $\Delta_0 \rightarrow \Omega_i$

    must complete the pre-commit phase consisting of interactions:
    - [Item 130d] **"commit request τ *'ps'*"** $\Delta_0 \rightarrow \Delta_j$
    - [Item 132(a)ii] **"no, unable to handle τ"** $\Delta_j \rightarrow \Delta_0$
    - [Item 130(f)i1] **"abort τ"** $\Delta_0 \rightarrow \Delta_j$
    - [Item 132(a)iii] **"yes, able to handle τ"** $\Delta_j \rightarrow \Delta_0$
    - [Item 130(f)i2] **"request τ: *'ps'* cannot be accepted"** $\Delta_0 \rightarrow \Omega_i$
    - [Item 130(f)ii1] **"commit τ"** $\Delta_0 \rightarrow \Delta_j$
    - [Items 132(c)i] **"commit τ acknowledgement"** $\Delta_j \rightarrow \Delta_0$

    that is, must not interact with any other process.

154. The subordinator between its receiving
    - [Item 130d] **"commit request τ *'ps'*"** $\Delta_0 \rightarrow \Delta_j$

    and either its
    - [Item 132b] **"abort τ acknowledgement"** or $\Delta_j \rightarrow \Delta_0$
    - [Item 132(c)i] **"commit τ acknowledgement"** $\Delta_j \rightarrow \Delta_0$

    must not interact with any other process.

155. The subordinator can engage in any number and order of user interactions between between a terminating pre-commit phase
    - [Item 132b] **"abort τ acknowledgement"** or $\Delta_j \rightarrow \Delta_0$
    - [Item 132(c)i] **"commit τ acknowledgement"** $\Delta_j \rightarrow \Delta_0$

and a subsequent next (i.e., "first" next) opening pre-commit phase

- [Item 130d] **"commit request τ' *'ps'*"** $\Delta_0 \rightarrow \Delta_j$

156. A user is free to pursue any activity, also with respect to subordinators, after having received either

- [Item 130(f)i2] a **"request τ: *'ps'* cannot be accepted"** reject or
- [Item 130(f)ii3] an **"request τ: path index map: *'p_to_wi'*"** accept.

If the user issues no

- Item 134(c)i→133(a)i: **"τ: get window *'p'*"** $\Omega_i \rightarrow \Delta_j$
- Item 133(a)ii: **"τ: put window *'w'* at *'p'*"** $\Omega_i \rightarrow \Delta_j$

or if it issues

- Item 133(a)ii: **"τ: put window *'w'* at *'p'*"** $\Omega_i \rightarrow \Delta_j$

before a

- Item 134(c)i→133(a)i: **"τ: get window *'p'*"** $\Omega_i \rightarrow \Delta_j$

then, of course the user must expect problems !

In other words there need be a reasoanbly strict discipline, called a protocol, with respect to the users' issuance of **get** and **put** commands.

8.3 Some Auxiliary State Notes

8.3.1 Window Schemas

schemas

schemas

Window schemas are abstractions of forests of domain frames and of forests of window frames. Let us remind ourselves of the overall definition of forests of window frames and forests of domain frames (Item 36 on page 27, Item 35 on page 25 and Item 58 on page 36):

50. FoWF = WNm $\overrightarrow{m}$ (WΣ × FoWF)
63. FoDF = WNm $\overrightarrow{m}$ (W × FoDF)
45. WΣ = W × FVAL × Cursor
36. W′ = WNm × WTy × mkWV(ANm-**set**,TVAL,RVAL,WNm-**set**)
35. W = {|w:W′•wf_W(w)|}

A window schema is a "stripped" forest of domain frames where, however, window names are either marked or unmarked, that is, where windows are either "fully" present or are just schematically so. A schematically shown window consists of just its marked or unmarked named and a set of such (cf. Item 36. above: the first and the last type expression). Notice that a window schema is

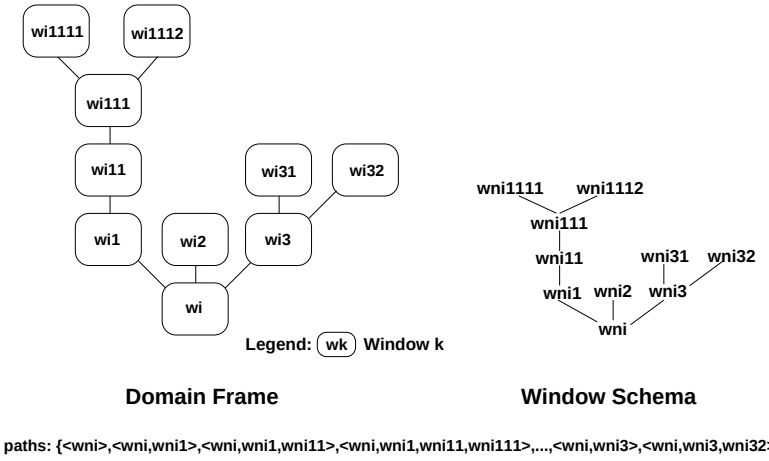


Figure 9: Commensurate domain frame, window schema and paths

the same as a forest of window schemas — although, in Fig. 9 we show a forest of only one tree.

157. A window name is now
    a) either marked
    b) or it is unmarked.

    and the two kinds are distinct.

158. A window schema has

159. From a window schema one can (as for forests of domain and window frames) extract the set of all marked paths to nodes in the window schema.

160. Marked paths are sequences of marked or unmarked window names.

161. Designating paths are marked paths such that the last window name is marked.

162. One can strip the elements of a marked path of their marking status.

163. From a forest of domain frames one can (as for a forest of window frames) extract a window schema.

type
157. muWNm = mWNm | uWNm
157a. mWNm == mark(s_wn:WNm)

```
157b. uMWn  == unmk(s_wn:WNm)
158.  WS      = muWNm m→ WS
160.  MP      = muWNm*
161.  DP      = {|dp:MP•is_designating_path(dp)|}
value
160.  is_designating_path: MP → Bool
160.  is_designating_path(mp⌢⟨mark(_)⟩) ≡ true
160.  is_designating_path(mp⌢⟨unmk(_)⟩) ≡ false

159.  paths: WS → MP-set
52.   paths(ws) ≡
52b.      let ps′ = ∪{paths(ws′)|ws′:WS•ws′ ∈ rng ws} in
52c.      ∪{⟨wn⟩,⟨wn⟩⌢p|wn:muWNm,p:P • wn ∈ dom ws∧p ∈ ps′} end

162.  strip: MP → P
162.  strip(mp) ≡ ⟨s_wn(mp(i))|i in [1..len mp]⟩

162.  strips: MP-set → P-set
162.  strips(mps) ≡ {strip(mp)|mp:MP•mp ∈ mps}

163.  xtr_WS: FoDF → WS
163.  xtr_WS(f) ≡
163.      [n↦[n′↦xtr_WS(f(n′))|n′:muWNm•n′∈dom f(n)]n:muWNm•n∈dom f]
```

The idea is that somehow all domain frame processes and all window frame processes share one and the same window schema, that is, have access to or keep a continuously updated copy of a window schema, and that "the sum total" of domain frames managed by the subordinate domain frame processes are abstracted by this window schema.[15]

The idea is furthermore that forest of window frames users, i.e., the proverbial "end users", while preparing for a session of window viewing and updating, can display this, the "global" window schema. At least those parts to which they may have been granted read/write access,[16]. The display can, for example, be two-dimensionally, such as either of the two "trees" of Fig. 9 on the previous page. A reason for a simple display could be for purposes of surveying windows, i.e., documents. A related reason could be to select which windows they wish to obtain (Get Window) and update (Put Window).

Since many users, that is, different forest of window frames processes, may wish to operate on sets of possibly overlapping windows during, most likely, overlapping time intervals, and in order to avoid that two or more users update an initially, when first obtained, same window, we need to inform the domain

[15] How these processes initially obtain and regularly update identical window schema copies is left as a simple exercise for the reader to narrate and formalise.

[16] We do not deal, in the current report, with window access authorisation. Since it is not a defensive issue appears to be an "add on" with which the current design can be extended: systematically and not too convolutely!

frame system of our intents. With the domain frame system of m forest of domain frames processes each handling their disjoint parts of a "sum total" forest of domain frames, being coordinated by a distinguished domain frames process,[17] Δ_0, the idea is to ensure that any two forest of window frames processes, Ω_{i_k} and Ω_{i_ℓ}, perform their operations on sets of overlapping windows in disjoint time intervals.

8.3.2 Unique Transaction Identifiers

utids

utids

The idea is now that any window process can send a request, ps, in the form of a set of designated paths (to the coordinating domain frame process) for performing a set of domain frame operations in some time interval such that no other window process' similar, concurrent requests, ps′, target at least one domain frame window already designated in request ps. The coordination domain frame process, Δ_0, upon receiving any request from a window frame process Ω_i acknowledges the receipt with a unique transaction identifier, τ_i. This identifier (τ_i) will be used in communication concerning this transaction between processes.

164. Thus there is an indefinite set, i.e., a type of unique transaction identifiers.
165. And there is an "oracle" function which, when invoked, yields a unique transaction identifier.
166. This function makes use, for technical reasons, see below, of a domain frame process, Δ_0, local variable uτs.

type
164. Υ
variable
165. uτs:Υ**-set** := {}
value
166. uniq_τ: **Unit** → Υ
166. uniq_τ() ≡ **let** τ:Υ•τ∉**c** uτs **in** uτs := **c** uτs ∪ {τ}; τ **end**

8.4 Channels

channels

channels

167. **Frame Process Indices and Channels:**
    a) The set of domain frame processes are uniquely indexed over DIdx. So is the set of window frame processes which are indexed over WIdx.
    b) d0 is the index of the unique coordinator (domain frame processes).
    c) wis is the set of window frame process indices.

[17] This distinguished domain frames process, Δ_0, turns out, we have decided, not itself to administer an own forest of domain frames.

d) dis is the set of domain frame process indices.

e) d0 is in DIdx and there are $m+1$ domain frame processes and there are n window frame processes.

f) Each window frame process can communicate with any domain frame process – via (WIdx×DIdx)-indexed channels – and vice versa.
   Thus there are $n \times m+1$ window (to/from domain) channels.
   No Ω_i or Δ_j can communicate with itself.

g) Domain frame process Δ_0 can communicate with any other domain frame process – via DIdx-indexed channels.
   Thus there are m domain frame channels.
   Δ_0 cannot communicate with itself and Δ_{j_k} cannot communicate with Δ_{j_ℓ} for j_k and j_ℓ different from 0.

type
167a. DIdx, WIdx
value
167b. d0:DIdx
167c. wis:(WIdx×DIdx)**-set**
167d. dis=DIdx**-set**
axiom
167e. d0 ∈ dis ∧ **card** dis=m ∧ **card** wis=n
channel
167f. {wch[wi,dj]|wi:WIdx,dj:DIdx•wi ∈ wis∧dj ∈ dis}:M[18]
167g. {dch[dj]|dj:DIdx}:M

8.5 States

8.5.1 Window Frame Process State $\Omega_i\Sigma$

window-i-state

window-i-state

Marking Window Schemas

168. There is a select and mark function: It internal non-deterministically selects such window names that is wishes to mark, or unmark; and proceeds to do so.

value
168. mark: WS → WS
168. mark(ws) **as** ws′ **post** strips(paths(ws)) = strips(paths(ws′))[19]

[18] Mnemotechnics: we have chosen wi's to be **wi**ndow indexes and dj's to be **d**oma**"j"**n indexes.

[19] Personally, we quite like this function definition: it shows abstraction at its best. Any marking and un-marking is, of course, possible; a user do not wish to be constrained; and the function definition gives "absolutely" no hint as to how one could, algorithmically, implement it !

• • •

Please keep these distinctions in mind:

- paths, p:P, Item 51 on page 33;
- marked paths, mp:MP, Item 160 on page 77; and
- designated paths, dp:DP, Item 161 on page 77.

The pragmatics is:

- Paths locate windows, w:W, Item 32 and Item 35 on page 25.
- Marked paths locate actual or un-selected windows.

The actual and un-selected window notions will be explained shortly.

- For actual windows, aw:AW, see Item 169b.
- For un-selected windows, uw:UW, see Item 169a.

Forests of Designated Window Frames The idea is then that forest of window frame processes work on not exactly the window frames of Sect. 3.1 on page 30 but on modified versions, called forest of designated window frames. We modify the window frames of Sect. 3.1 on page 30 into designated window frame. A forest of designated window frames is like a forest of window frames except that where the forest of window frames process, in its designated window paths, for example, dp:DP, have indicated that a window name is unmarked the "corresponding" window, in a forest of designated window frames, is an closed window and where window name is marked the "corresponding" windows, in a forest of designated window frames, is either a closed or an opened window.

169. A designated window frame, cf. Fig. 10 on the following page, consists of a pair of an unmarked or an actual window and a possibly empty set of uniquely marked or unmarked window name-identified actual, respectively unmarked windows.
    a) A closed window is a pair of a marked or unmarked window name and a set of marked or unmarked window names.
    b) An opened window is a window whose window name is a marked window name.

Thus marked windows can be open or closed. Unmarked windows can on be (shown) closed.

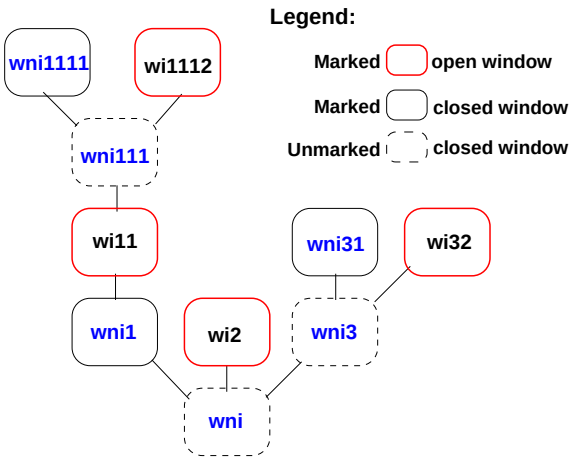


Figure 10: A designated window frame of a forest of such

type
169. FoDWF = muWNm $\overrightarrow{m}$ (CoOW × FoDWF)
169. CoOW = CW | OW
169a. CW == mkCW(s_wn:muWNm,s_wns:muWNm-**set**)
169b. OW == mkOW(w:W)
axiom
169b. ∀ mkOW(w):OW•s_wn:w ∈ mkWNm

170. Designated window frames must be well-formed.

- For now we omit definition of well-formedness of designated window frames.
- We refer instead to Sect. 3.2 Item 48 on page 31 [Well-formed Window Frames] and Sect. 4.1.1 Item 59 on page 36 [Well-formed Domain Frames], for examples of what has to be expressed.

Transactions We now "assemble" disjoint sets of forest of designated window paths and unique transaction identifiers into one state component: transactions. The idea is that a set of forest of window frames processes can pursue one or more transactions concurrently. These transactions are sets, {fodwp$_{i_j}$, fodwp$_{i_k}$, ... fodwp$_{i_\ell}$}, of forest of designated window paths. They will first have been OK'ed by the forest of domain frames coordinator on behalf of those subordinate forest of domain frames processes, indexed by dj$_{j_q}$, dj$_{j_2}$, ..., dj$_{j_p}$:DIdx, which "hold" the windows requested by the forest of window frames processes. The OKs are in the form of proper *commit* communications from Δ_0. These *commit*s

provide the transaction requesting forest of window frames process with a a pair: a unique transaction identifier and map from designated forest of window frame paths to forest of domain frames process identifiers.

171. Commit transactions, ctrs:CTRS, are pairs of a unique transaction identifier and a forest of domain frames window allocation, fdfwas:FoDFWAs.
    a) Domain frame window allocations, fdfwas:FoDFWAs, map each requested and OK'ed designated window path into the identity of the identifier of the forest of domain frames process which stores the designated window.
    b) For simplicity we assume that the coordinating forest of domain frames process, Δ_0, identified by d_0 holds no windows.[20]

type
171. CTRS = TId × FoDFWAs
171a. FoDFWAs = DP $\xrightarrow[m]{}$ DIdx
axiom
171b. ∀ fodfwas:FoDFWAs•d0 ∉ **rng** fodfwas

The Forest of Window Frames Process State

172. Any forest of window frames process evolves around accessing and updating a local state, $\Omega\Sigma_{i_k}$ which consists of the following separate components:
    and a window schema,
    a) a transaction map, tm:TM, from unique transaction identifiers to domain frame window allocations, and
    an evolving forest of window frames.

173. No two domain frame window allocations of a transaction map can share designated paths.

type
172. ΩΣ = WS × TM × FoWF
172a. TM = UTId $\xrightarrow[m]{}$ FoDFWAs

theorem:
173. ∀ tm:TM,uti,uti′:UTId • uti≠uti′ ⇒ **dom** tm(uti) ∩ **dom** tm(uit′) = {}

[20] The coordinating forest of domain frames process may have "its own" forest of domain frames. But the windows of that forest cannot be operated upon by any forest of window frames processes, only by the coordinator process.

The left-to-right order of the three state components, WS×TM×FoWF is pragmatically motivated: From the less dynamic, that is, to the user, least often changing – usually not changing from session to session[21] – via session-stable window schemas and transaction maps, to the most dynamic, that is, to the user, most often changing component values.

Usually the following is a common pattern of operations on a forest of window frames state: (i) A session starts with the user marking the window schema. (ii) Then requesting permission to perform window operations on the designated windows (as marked). (iii) Then performing a time-wise possibly length sequence of window operations: (iii.a) initially some opening of windows, (iii.b) then some writing on these windows, and (iii.c) then the putting of updated windows back to the forest of domain frames processes from where (cf. (iii.a)) possibly earlier versions of these windows were first obtained. (iv) Finally all opened windows have been returned and the session is over.

State Well-formedness

174. Forest of window frames process states, $\omega\sigma : \Omega\Sigma$, must be well-formed.

174.a Let the state be composed: $\omega\sigma$:(ws,fowf,tm).

174.b The set of stripped forest of designated window frame paths, fowf, must be equal to the set of stripped designated paths of the transactiom map tm.

This constraint expresses that the forest of designated window frames, fowf, is initially constructed from the transactiom map tm and that openings, closings and other operations on marked windows of that forest does not change its structure, i.e., does not add or delete any windows.

174.c The set of stripped designated paths of the transactiom map, tm, must be a subset of the set of stripped designated paths of the window schema ws.

This constraint expresses that transaction map tm was the result, from the forest of domain frames coordinator, of a request from the current forest of window frames process to commit a number of windows based on a set of designated window paths. This set of designated window paths was first constructed (i.e., indicated) by the current forest of window frames process by a marking, say on a temporary copy of the window schema.

value
174. wf_ΩΣ: ΩΣ → **Bool**
174.a wf_ΩΣ(ws,fowf,tm) ≡
174.b-.c stripped_fowf_paths(fowf) = stripped_tm_paths(tm) ⊆ stripped_ws_paths(ws)

[21] By a session we shall understand a time interval during which a user works on a dedicated set of windows.

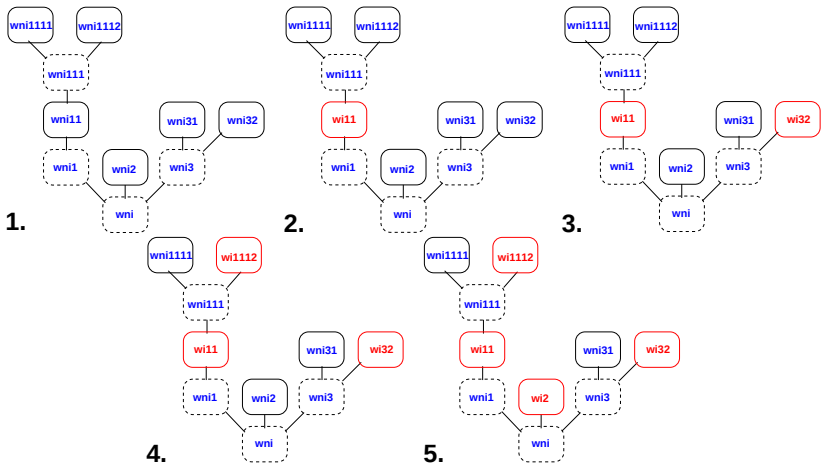
174.b stripped_fowf_paths: FoDWF → P-**set**
174.b stripped_tm_paths: TM → P-**set**
174.c stripped_ws_paths: WS → P-**set**

Window Process State Function Signatures A number of functions apply to forest of window frames process states and either evaluate to a value (but) with no state transition or effect a state transition. These functions and their signatures are:

175. Cre_DFW: create a designated window frame, cf. Appendix A.2.1
176. Opn_W: open an indicated marked window, cf. Appendix A.2.2
177. Clo_W: close an indicated marked window, cf. Appendix A.2.3
178. Wri_W: click and write into a window field, cf. Appendix A.2.4
179. Del_DFW: delete the designated window frame, cf. Appendix A.2.5

value
175. Cre_DWF: ΩΣ~→ΩΣ
176. Opn_W: P→ΩΣ~→ΩΣ
177. Clo_W: P→ΩΣ~→ΩΣ
178. Wri_W: P→FNm×FVAL→ΩΣ~→ΩΣ
179. Del_DFW: ΩΣ~→ΩΣ

We now show a sequence of "opening" the windows of a window schema.



A snapshot of a series of window frame window openings

You may think of snapshot 1. as the result of executing

1. Create_DFW(dwf,tm) ⟶ (dwf$_\alpha$,tm)[22]

[22] ⟶ denotes: "resulting in".

where what we see in snapshot 1. is indeed the dwf_α. It has been constructed sôlely from the transaction map tm.

You can then think of snapshots 2.–5. as the result of four successive executions:

2. $\mathsf{Opn\_W}(\langle wn_i, wn_{i_1}, wn_{i_{1_1}}\rangle)(\mathsf{dwf}_\alpha,\mathsf{tm}) \longrightarrow (\mathsf{dwf}_\beta,\mathsf{tm})$,
3. $\mathsf{Opn\_W}(\langle wn_i, wn_{i_3}, wn_{i_{3_2}}\rangle)(\mathsf{dwf}_\beta,\mathsf{tm}) \longrightarrow (\mathsf{dwf}_\gamma,\mathsf{tm})$,
4. $\mathsf{Opn\_W}(\langle wn_i, wn_{i_1}, wn_{i_{1_1}}, wn_{i_{1_{1_1}}}, wn_{i_{1_{1_{1_2}}}}\rangle)(\mathsf{dwf}_\gamma,\mathsf{tm}) \longrightarrow (\mathsf{dwf}_\zeta,\mathsf{tm})$,
5. $\mathsf{Opn\_W}(\langle wn_i, wn_{i_2}\rangle)(\mathsf{dwf}_\zeta,\mathsf{tm}) \longrightarrow (\mathsf{dwf}_\eta,\mathsf{tm})$.

If these five functions remind you of the window frame operations of Sect. 6 then that is no coincidence. Their [re]definition is found in Appendix A. But we need first "clean-up" the window frame process state.

Discussion

180. We have made a number of design simplifications. These are:
181. Instead of separate close and put window operations we merge these into one, the $\mathsf{Clo\_W(p)}(\omega\sigma)$ operation.
    - Where the former close operation involved only the window frame
    - and the put operation involved both the window and the domain frames,
    - the current close operation is both a designated window frame process and a domain frame process operation.
182. Instead of separate click on window and write to window operations, we merge these into one, the $\mathsf{Wri\_W(p)}(\omega\sigma)(\mathsf{p})(\mathsf{fn},\mathsf{fv})(\omega\sigma)$ operation.
    - We use this "merger" to simplify our window design.
    - Now there is no need to "remember"
        - which field (etc.) was most recently clicked,
        - nor the value of the most recent write.
183. Finally we remove the concept of (scroll down) curtains.

Redesign of Window Frame Process Windows With the design re-decisions of Item 181, Item 182 and Item 183 and with notions of designated window paths, window schemas, and designate window frames, the former design can be reduced.

Main State Components

type
172. ΩΣ = WS × TM × FoWF
158. WS = muWNm $\overrightarrow{m}$ WS
172a. TM = UTId $\overrightarrow{m}$ FoDFWAs
171a. FoDFWAs = DP $\overrightarrow{m}$ DIdx
169. FoDWF = CoOW × (muWNm $\overrightarrow{m}$ CoOW)
169. CoOW = CW | OW
169a. CW == mkCW(s_wn:muWNm,s_wns:muWNm-**set**)
169b. OW == mkOW(w:W)
36. W′ = mkWNm × WTy × WVAL
35. W = {|w:W′•wf_W(w′)|}
32. WVAL == mkWV(s_key:KNms,s_tpl:TVAL,s_rel:RVAL,s_ws:muWNm-**set**)

Values and Types

1. AVAL == mkIV(**Int**)|mkRV(**Rat**)|mkBV(**Bool**)|mkT(**Text**)|″`nil`″
2. ATyp = {|″`int`″,″`rat`″,″`bool`″,″`text`″,″`nil`″|}
11. TVAL = ANm $\overrightarrow{m}$ AVAL
16. FTyp = ATyp
17. TTyp = ANm $\overrightarrow{m}$ ATyp
21. KNm = ANm
22. KNms = KNm-**set**
23. KVAL = ANm $\overrightarrow{m}$ AVAL
24. KTyp = ANm $\overrightarrow{m}$ ATyp
26. RVAL′ = KVAL $\overrightarrow{m}$ TVAL
26. RVAL = {|rv:RVAL′•wf_RVAL(rv)|}
33. WTyp = TTyp

Names and Paths of Various Forms

12. FNm = ANm
12a. ANm == mkANm(s_nm:Nm)
13. WNm == mkWNm(s_wn:Nm)
14. Nm
157. muWNm == mWNm | uWNm
157a. mWNm == mark(s_wn:WNm)
157b. uWNn == unmk(s_wn:WNm)
160. MP = muWNm*
161. DP = {|dp:MP•is_designating_path(dp)|}

8.5.2 Subordinate Domain Frame Process State $\Delta_j\Sigma$ delta-j-state

delta-j-state

Forest of Domain Frames

184. The main state component of any subordinator is the forest of domain frames.

type
184. FoDF = WNm $\overrightarrow{m}$ (W × FoDF)

Auxiliary State Components

185. Auxiliary state components of any subordinator are recordings, pcom:PreCom, of which commitments are being tentative,

186. and which are committed, com:Com.

187. A window-designating path three states: `"unread"`, `"read"` or `"written"`.

type
185. Wuse == unread | **read** | written
186. PreCOM = UTid $\overrightarrow{m}$ P1-**set**
187. COM = UTid $\overrightarrow{m}$ (P1 $\overrightarrow{m}$ Wuse)

Auxiliary and Enduring States

188. We decide to "split" the subordinator state into two, the auxiliary $\Delta_j\Sigma$ and the more "enduring"[23] forest of domain frames.

189. Thus the signature of the Δ_j processes contain both:

type
188. $\Delta_j\Sigma$ = PreCOM × COM
value
189. Δ_j: ... → $\Delta_j\Sigma$ → FoDF → **Unit**

State Transition Functions

[23] By an 'enduring' state we mean one whose values transgress transactions.

PreCommit

190. When a subordinator replies with a **"yes, able to handle τ"** message to a coordinator's **"please, commit request τ *'bs'*"** message, then that subordinator places the pair $[\tau \mapsto \text{bs}]$ in the pre-commitment state component.

value
190. PreCommit: $(\Upsilon \times \text{Bucket}) \to \Delta_j\Sigma \to \text{PreCOM}$
190. PreCommit(τ,ps)($\delta_j\sigma$:(pcom,com)) $\equiv$ pcom $\cup$ $[\tau \mapsto \text{ps}]$

DeCommit

191. When a subordinator receives the **"abort τ"** message then the subordinator removes the τ entry from the pre-commitment state component.

value
191. DeCommit: $\Upsilon \to \Delta_j\Sigma \to \text{COM}$
191. DeCommit(τ)($\delta_j\sigma$:(pcom,com)) $\equiv$ pcom $\setminus$ $\{\tau\}$

Commit

192. When a subordinator receives the **"commit τ"** message then the subordinator moves the τ entry from the pre-commitment state component to the commitment state component.

value
192. Commit: $\Upsilon \to \Delta_j\Sigma \to \Delta_j\Sigma$
192. Commit(τ)($\delta_j\sigma$:(pcom,com)) $\equiv$
192. (pcom$\setminus\{\tau\}$,com $\cup$ $[\tau \mapsto (\text{p},''\texttt{unread}'')|\text{p:P1}\bullet\text{p} \in \text{ps}]$)

Get Window

193. When a subordinator receives a **"τ: get window *'p'*"** from some user and replies to that user: **"τ: here is path *'p'* window: *'w'*"**
    a) the subordinator marks the (τ,p) entry in the resulting com′ state component with a `"in use"` mark;
    b) this component's (τ,p) entry, before the get operations was marked `"not in use"`;
    c) all else is unchanged in the $\Delta_j\Sigma$ state and in the forest of domain frames,

d) except that now the argument window, w, is at position p in the resulting forest of domain frames.

value
193. GetW: $(\Upsilon\times P) \to \Delta_j\Sigma \to$ FoDF $\to \Delta_j\Sigma \times$ W
193. GetW(τ,w)($\delta_j\sigma$:(pcom,com))(fodf) **as** ($\delta_j\sigma'$:(pcom′,com′),w)
193. **pre** $\tau \in$ **dom** com
193b. $\wedge$ (p,"`not in use`") $\in$ com(τ)
193. $\wedge$ p $\in$ paths(fodf)
193. **post** p $\in$ paths(fodf′)
193d. $\wedge$ w = s_W(p)(fodf′)
193. $\wedge$ paths(fodf) = paths(fodf′)
193d. $\wedge \forall$ p′:P1•p′ $\in$ paths(fodf′)\{p}$\Rightarrow$sel_W(p)(fodf)=s_W(p)(fodf′)
193a. $\wedge$ (p,"`in use`") $\in$ com(τ)

Close/Put Window

194. When a subordinator receives a **"τ: put window 'w' at 'p'"** from some user and has replied **"τ: path 'p' window has been stored"** then the subordinator moves the τ entry 'p' from the commitment state component. If that was the last path in the "τ commitment" in com then the subordinator erases the τ entry (which is now empty) from com.

194. Close_Put: $(\Upsilon\times P\times W) \to \Delta_j\Sigma \to$ FoDF $\to \Delta_j\Sigma \times$ FoDF
194. Close_Put(τ,p,w)($\delta_j\sigma$:(pcom,com))(fodf) **as** ($\delta_j\sigma'$:(pcom′,com′),fodf′)
194. **pre** $\tau \notin$ **dom** pcom $\wedge$ $\tau \in$ **dom** com
194. $\wedge$ p $\in$ com(τ) $\wedge$ p $\in$ paths(fodf)
194. $\wedge$ (p,"`in use`") $\in$ com(τ)
194. **post** p $\in$ paths(fodf′)
194. $\wedge$ w = sel_W(p)(fodf′)
194. $\wedge$ paths(fodf) = paths(fodf′)
194. $\wedge \forall$ p′:P1•p′ $\in$ paths(fodf′)\{p}$\Rightarrow$sel_W(p)(fodf)=sel_W(p)(fodf′)
194. $\wedge$ com′(τ)=com\{τ}
194. $\wedge$ com′=[ ] $\Rightarrow$ [no special action]

8.5.3 Coordinator Process State, $\Delta_0\Sigma$ delta-0-state

delta-0-state

Window Catalogue

195. The coordinator process state, $\delta_0\sigma$ contains a catalogue which for every path of every forest of domain frames records the process index, di:DIdx, which holds the window designated by that path.

196. Let cat:CAT be the "typical" catalogue in use at any time by Δ_0, i.e., in $\delta_0\sigma$.

197. Let, cf. 167d on page 80, dis:DIdx-**set** be the set of indices all domain processes. Then the image, kwrng cat, of the catalogue is a subset of dis.

type
195. CAT = P $\overrightarrow{m}$ DIdx
value
196. cat:CAT
axiom
197. **rng** cat $\subseteq$ dis\{d0}

User Requests and Coordinator Buckets

198. A request:Request is a the set of paths to windows for which (i.e., the windows) a user is requesting read/write access, respectively the request at a designated forest of domain frames process.

199. A bucket:Buckets is a technical concept: it records, for some domain process indices which

200. Given a catalogue, cat, one can analyse(ps)(cat) a request, ps, into the set of all those buckets, one for each relevant forest of domain frames process, each of which records a the request for a distinct forest of domain frames process's windows.

type
198. Request = P1-**set**
199. Buckets = DIdx $\overrightarrow{m}$ Request
value
200. analyse: Request $\to$ CAT $\to$ Buckets
200. analyse(ps)(cat) $\equiv$ [dj$\mapsto${p|dj:DIdx,p:P1•p $\in$ ps $\wedge$ cat(p)=dj}]

The $\Delta_0\Omega$ Coordinator State

201. The $\Delta_0\Omega$ coordinator state consists of (so far[24]) of the catalogue state component.

type
201. $\Delta_0\Omega$ :: s_cat:CAT $\times$ ...

8.6 Processes

processes

processes

8.6.1 Window Process Ω_i: Initial Actions

202. A request for a set of Open Window and Put Window operations is a set of paths: ps.

203. **Window Process Ω_i: Initial Actions**

   a) Ω_i first selects an appropriate subset **'ps'** of paths of itw window schema.
   b) Then Ω_i sends **"request transaction *'ps'*"** [mkRTPS(ps)] to Δ_0.
   c) Now Ω_i awaits a reesponse from Δ_0.
   d) If the response from Δ_0 is **"request τ: *'ps'* cannot be accepted"** [ι130(f)i2 π69] then that's it.
   e) If the response from Δ_0 is **"request τ: path index map: *'p_to_wi'*"**,
      i. then that map becomes the basis for a session of get and put window commands.
      ii. The window schema for that get and put window session is unchanged.
      iii. The forest of window frames is derived directly from the index map.
      iv.

type
202. Request =P1-**set**
axiom
202. ∀ ps:Request•ps≠{}
value
203. Ω_i: wi:WIdx → $\Omega_i\Sigma$
203. → **in**,**out** dch[wi,d0]:(RTPS|RTPSCBA|RTPIM) **Unit**
203. Ω_i(wi)(ws,tm,fowf) ≡
203a. **let** ps = paths(mark(ws)) **in**
203b. dch[wi,d0] ! mkRTPS(ps) ;
203c. **let** re = dch[wi,d0]? **in**
203. **case** re **of**
203d. mkRTPSCBA(u,ps) → Ω_i(wi)(ws,tm,fowf),
203e. mkRTPIM(u,tm′) →
203(e)i. **let** tm″ = tm′,
203(e)ii. ws′ = ws,
203(e)iii. fofw′:FoFW • paths(fofw′)=**dom** tm′ **in**
203(e)iv. Get_Put_Session(ws′,tm′,fowf′)
203. **end end end end**

[24] further work may reveal the need for more – or changed – state components

8.6.2 The Coordinator Process, Δ_0

Δ_0: Initial Actions

204. **The Coordinator Process, Δ_0: Initial Actions:**

    a) Δ_0 receives the request, ps, from window frame process j.
    b) Δ_0 checks that the request paths are indeed paths of some domain frame of some domain frame process. If not, the request is "returned" **"path(s) ps′ are not defined"** where ps′ are those paths of ps which are not in any of the subordinator forests of domain frames. Δ_0 abandons any further handling of the request.
    c) If all paths are defined Δ_0 analyses ps into a number of sub-requests.
    d) Each sub-request, ps_{d_i}, is a subset of rq; each such ps_{d_i} "destined" for domain frame process Δ_i which possesses exactly those windows in rq designated by ps_{d_i} and only those.
    e) To help perform this "bucketing" Δ_0 has a catalogue which lists which domain frame processes stores the window designated by any one path.

```
value
204.  Δ0: Δ0Σ → in,out {wch[ wi,dj ]
204.        | wi:WIdx,dj:DIdx•wi ∈ wis∧dj ∈ dis\{d0}}:(RTPS|IRPPSND)  Unit
204.  Δ0(δ0σ :(cat,...)) ≡
204c.     []{let mkRTPS(ps) = wch[ wi,d0 ]? in
204b.       if ps ∼⊆ dom cat
204b.         then wch[ wi,d0 ] ! mkIRPPSND(ps/dom cat) ; Δ0(cat,...)
204e.         else let bs = analyse(ps), τ = unique_τ() in
205.                Δ0_PREPARE_PHASE(τ,wi,bs)(δ0σ)  end
204.       end end | wi:WIdx} ;
204.      Δ0(δ0σ)
```

After an initial request from some window frame process Ω_i, with its analysis of "requested paths", the coordinating domain frame process Δ_0 enters a prepare phase.

Δ_0: Prepare Commit Phase Actions

205. **The Coordinator Process, Δ_0, PREPARE PHASE:**

    a) Based on a successful analysis of request paths into "buckets", Δ_0 now sends a message to each of the subordinate domain frame processes identified in "bucket" bs. The message, directed at those subordinate domain frame processes, contains paths to windows in that process's forest of domain frames.

b) The coordinator now collects, into a set, the answers, either **"yes, able to handle τ"**: mkYATHT(utid) or **"no, unable to handle τ"**: mkNUTHT(utid) from each of these subordinate domain frame processes – as to whether they are able to accept or not accept transactions involving all of these path windows.

They might already be engaged in coordinated transactions with window frame processes, and some of these engagements may conflict, that is, target paths in the preparation message.

c) Having collected all the replies the coordinator inquires as to the composition of the set of replies:[25].

d) If **"yes, able to handle τ"** then the coordinator proceeds to the COMMIT PHASE.

e) If at least one inquired domain frame process replies **"no, unable to handle τ"** then the coordinator proceeds to the ABORT PHASE.

type
205b. RPLS = DIdx $\overrightarrow{m}$ YATHT|NUTHT
value
205. Δ_0_PREPARE_PHASE: (UTId×wi:WIdx×Bucket) → $\Delta_0\Omega$ →
207. **in**,**out** {dch[dj]|dj|DIfx•dj ∈ dis\{d0}}:(PCORTPS|YATHT|NUTHT) **Unit**
205. Ω_0_PREPARE_PHASE(τ,wi,bs)($\delta_0\omega$) ≡
205a. ‖{dch[dj] ! mkPCORTPS(τ,bs(dj))|dj:DIdx•dj ∈ **dom** bs} ;
205b. **let** repls = ∪{[dj↦dch[dj]?]|dj:DIdx•dj ∈ **dom** bs} **in**
205c. **case rng** repls **of**
205d. {mkYATHT(τ)} → Δ_0_COMMIT_PHASE(τ,wi,bs)($\delta_0\omega$),
205e. {mkNUTHT(τ)} ∪ rpls′ → Δ_0_ABORT_PHASE(τ,wi,rpls)($\delta_0\omega$)
205. **end end**

206. **The Coordinator Process, Δ_0, ABORT PHASE:**

a) The requesting window frame processor, (still wi) is sent the message **"request τ: *'ps'* cannot be accepted"** (mkRTPSCBA(τ,ps)) (due to one or more domain frame processes already committed to other window frame processes' request for overlapping path-designated windows); and, concurrently

b) each of the accepting window frame processes is sent a message **"abort τ"** (mkAT(τ)) to abort their τ transactions.

206. Δ_0_ABORT_PHASE: (TId×WIdx×RPLS) → $\Delta_0\Sigma$ →
206. **out** {wch[wi,d0]|wi:WIdx•wi ∈ wis}:RTPSCBA

[25] These reply sets contain either one element (either **"no, unable to handle τ"** or **"yes, able to handle τ"**) or two elements (both of these replies).

206. {dch[dj]!dj:DIdx•dj ∈ dis\{d0}}:AT **Unit**
206. Δ_0_ABORT_PHASE(τ,wi,rpls)($\delta_0\sigma$) ≡
206a. (wch[wi] ! mkRTPSCBA(τ,ps) ‖
206b. ‖{dch[dj] ! mkAT(τ) |dj:DIdx•dj ∈ **dom** rpls∧rpls(dj)=mkYATHT(τ)})

Δ_0: Commit Phase Actions

207. Δ_0 sends a **"commit τ"** [mkCT(τ)] message to all relevant subordinate domain processes.
208. Then Δ_0 awaits receiving an acknowledgement, **"commit τ acknowledgement"** [mkCTA(τ)], from each of these relevant subordinate domain processes.
209. For the time being we shall omit explaining what is to with the accumulated acknowledgements, acks.
210. Finally the coordinator end the τ transaction once having received all acknowledgements by sending the requesting forest of window frames process a **"request τ: path index map: *'p_to_wi'*"**, i.e., mkRTPIM(τ,pim) where pim maps requested paths into forest of domain frames process identifiers.

type
207. PIM = P $\overrightarrow{m}$ DIdx
value
207. Δ_0_COMMIT_PHASE: (UTId×wi:WIdx×Buckets) → $\Delta_0\Sigma$ →
207. **in**,**out** {dch[dj]|dj:DIfx • dj ∈ dis\{d0}}:CT
207. **in**,**out** {wch[wi,dj]:wi:WIdx,dj:DIdx • wi ∈ dis∧dj ∈ dis\{d0}}:RTPIM **Unit**
207. Δ_0_COMMIT_PHASE(τ,wi,bs)($\delta_0\sigma$) ≡
207. ‖{dch[dj] ! mkCT(τ) | dj:DIdx•dj ∈ **dom** bs} ;
208. **let** acks = ∪{[dj↦dch[dj]?]|dj:DIdx•dj ∈ **dom** bs} **in** dispose(acks) **end**
210. **let** pim = [p↦cat(p)|p:P1•p ∈ **dom** bs] **in**
210. wch[wi] ! mkRTPIM(τ,pim) **end**

8.6.3 Subordinate Domain Frame Processes, $\Delta_j, j \in \{1..m\}$

Coordinator–Subordinator Transactions: $\Delta_j, j \in \{1..m\}$

dj0 dj0

211. Each $\Delta_j, j \in \{1..m\}$ internally non-deterministically
    a) either engages with the coordinator Δ_0,
    b) or with a user, Ω_i,
    c) or itself performs window operations.

```
value
211.  Δj: dj:DIdx → ΔΩ → FoDF →
211.        in,out dch[ 0 ]
211.        in,out {wch[ wi,dj ]|wi:WIdx • wi ∈ dis}  Unit
211.  Δj(dj)(δσ)(fodf) ≡
211a.       Δj Δ0(dj)(δσ)(fodf)
211b.    ⫿ Δj Ωi(dj)(δσ)(fodf)
211c.    ⫿ Δj Own(dj)(δσ)(fodf)
```

dj1

Subordinator–Coordinator Transactions: $\Delta_j\Delta_0, j \in \{1..m\}$ dj1

212. The $\Delta_j\Delta_0, j \in \{1..m\}$ process has same state signature as the Δ_j process but only communicates with the coordinator.
    a) Each $\Delta_j\Delta_0$ initially expresses willingness to engage with the coordinator, Δ_0
    b) expecting a "first" engagement to be the receipt of the message: **"please, commit request** τ ***'ps'*****"** (mkPCORTPS(τ,ps)).
    c) $\Delta_j\Delta_0$ now inspects its current commitments.
    d) If there are conflicts then $\Delta_j\Delta_0$ sends the message **"no, unable to handle** τ**"** (mkNUTHT(τ)) and reverts to being Δ_i.
    e) If there are no conflicts
        i. then $\Delta_j\Delta_0$ sends the message **"yes, able to handle** τ**"** (mkYATHT(τ))
        ii. and $\Delta_j\Delta_0$ registers the bucket message in its pre-commitment stage component and awaits further commit messages:
            1. either the coordinator eventually sends an **"abort** τ**"** (mkAT(τ)) message and $\Delta_j\Delta_0$ removes the pre-commitment bucket whereupon $\Delta_j\Delta_0$ reverts to being Δ_j;
            2. or the coordinator eventually sends a **"commit** τ**"** (mkCT(τ)) message and $\Delta_j\Delta_0$ moves the pre-commitment bucket to the commitment state component and becomes Δ_j.

```
value
212.  Δj Δ0: dj:DIdx → ΔΩ → FoDF → in,out dch[ 0 ]  Unit
212.  Δj Δ0(dj)(δσ)(fodf) ≡
212a.     case dch[ 0 ]? of
212b.        mkPCORTPS(τ,bs)
212c.          → if ↑ com ∩ ps ≠ {}
212d.                then dch[ 0 ] ! mkNUTHT(τ)
212(e)i.               else (dch[ 0 ] ! mkYATHT(τ);
212(e)ii.                   Δj(dj)(reg_B(bs)(δσ))(fodf)) end,
212(e)ii1.    mkAT(τ) → Δj(dj)(rem_B(bs)(δσ))(fodf),
212(e)ii2.    mkCT(τ) → Δj(dj)(mov_B(bs)(δσ))(fodf)
212.      end
```

The Subordinator–User Transactions: $\Delta_j\Omega_i, j \in \{1..m\}, i \in \{1..n\}$ dj2 dj2

213. The $\Delta_j\Omega_i, j \in \{1..m\}, i \in \{1..n\}$ process only communicates with users.

a) Each subordinator expresses willingness to engage with any user. (Hence the external non-deterministic choice [] over the index range of users.)

i. Each subordinator, when interacting with users, is prepared to receive either of two messages: either the subordinator receives the message **"τ: get window 'p'"** [mkTGWP(τ,p)] – in which case:

1. either the p is *** and the subordinator retrieves the requested window and sends it in a a message **"τ: here is path 'p' window: 'w'"** [mkHIPPW(τ,p,w)] to to the user and reverts to being Δ_i.
2. or the p is not *** and the subordinator sends a message **"τ: cannot find window at 'p'"** [mkCFWAP(τ,p)] to the user and reverts to being Δ_i,

ii. Or the subordinator receives the message **"τ: put window 'w' at 'p'"** [mkTPWPW(τ,w,p)] – in which case:

1. either the p is not *** and the subordinator sends a message **"τ: cannot store window at 'p'"** [mkCSWAP(τ,p)] to to the user and reverts to being Δ_i.
2. or the p is *** and the subordinator sends a message **"τ: path 'p' window has been stored"** [mkPPWHBS(τ,p)] to to the user and reverts to being Δ_i with the updated window.

```
value
213.  Δj Ωi: dj:DIdx → Δj Σ → FoDF →
213.        in,out {wch[ wi,dj ]:wi:WIdx • wi ∈ dis}  Unit
213.  Δj Ωi(dj)(δσ)(fodf) ≡
213a.   [] {case wch[ wi,dj ] ? of
213(a)i.          mkTGWP(τ,p)
213(a)i1.            → if p ∈ paths(fodf)∧com(p)=wi
213(a)i1.                 then wch[ wi,dj ] ! mkHIPPW(τ,p,GetW(p,fodf))
213(a)i2.                 else wch[ wi,dj ] ! mkCFWAP(τ,p);
213(a)i2.              Δj(dj,com,pcom,fodf,{}) end
213(a)ii.         mkTPWPW(τ,w,p)
213(a)ii1.           → if p ∈ paths(fodf)∧com(p)=wi
213(a)ii1.                then (wch[ wi,dj ] ! mkPPWHBS(τ,p);
213(a)ii1.                      Δj(dj,com,pcom,fodf,{}))
213(a)ii2.                else (wch[ wi,dj ] ! mkCSWAP(τ,p);
213(a)ii2.                      Δj(dj)(δσ)(PutW(p,w)(fodf))) end
213.       end | wi:WIdx•wi ∈ wis }
```

dj3

The Subordinator "Own" Transactions: Δ_jOwn, $j \in \{1..m\}$ dj3

We are first reminded of the simple transaction system's forest of domain frames process (cf. Sect. 7.3.3 on page 59):

```
type
100.  Cmd == init | crea | rmdf | putw | getw | inut
value
101.  forest_of_domain_frames: Unit → in,out {ch[wi]|wi:WIdx•wi ∈ wis} Unit
101.  forest_of_domain_frames() ≡
103.    variable fofwv:FoDF := int_iniFoDF();
102.    while true do
104.      let cmd = init⌈crea⌈remv⌈put⌈get⌈inut in
101.      case cmd of
105.        init → fofwv := elab_FoDFinit(),
106a.       crea → fofwv := elab_FoDFCre(c vdf),
106b.       rmdf → fofwv := elab_FoDFRmv(c vdf),
106c.       putw → fofwv := elab_FoDFPutW(c vdf),
106d.       getw → fofwv := elab_FoDFGetW(c vdf),
107.        inut → interaction()
100.    end end end
```

```
107.  interaction: FoDF → FoDF
107.  interaction(fodf) ≡
107.    variable lfodf:FoDF
107a.    []{let req = ch[i]? in
107a.      case req of
107b.        mkDFGW(p) →
107c.          if pre:eval_DFGetW(mkDFGW(p))(fodf)
107d.            then ch[i]!eval_DFGetW(mkDFGW(p))(fodf)
107e.            else ch[i]!"error" end ; chaos,
107b.        put →
107c.          if pre:int_DFPutW(put)(c vdf)
107d.            then lfodf := int_DFPutW(put)(df);ch[i]!"ok"
107e.            else ch[i]!"error" ; chaos end
107a.      end | i:WIdx•i ∈ wis end} ; lfodf
```

- We need to modify formula Lines 100–107e slightly.

214.

215.

216.

217.

218.

219.

220.

221.

222.

```
value
214.  Δj Own: dj:DIdx → Δj Σ → FoDF →
214.        in,out {wch[ wi,dj ]|wi:WIdx•wi ∈ wis}  Unit
214.  Δj_Own(dj)(δσ)(fodf)
215.
216.
217.
218.
219.
220.
221.
222.
```

8.6.4 Window Frame Processes, Ω_i

wi

dj0

We are first reminded of the simple transaction system's forest of window frames process (cf. Sect. 7.3.4 on page 61):

```
type
108.  Cmd = opn | put | clk | wri | sel | inc | clo

value
109.  forest_of_window_frames: i:WIdx × FoWF
109.                          → in,out ch[ i ]  Unit

109.  forest_of_window_frames(wi,fowf) ≡
111.     variable fowfv:FoWF := fowf;
110.     while true do
108.        let cmd = opn⌈clo⌈put⌈clk⌈wri⌈sel⌈inc in
112.        case cmd of
113.           opn → fowfv := elab_WFOpnW(wi,fowf),
114.           put → fowfv := elab_WFPutW(wi,fowf),
115b.          clk → fowfv := elab_WFClkW(fowf),
115c.          wri → fowfv := elab_WFWriW(fowf),
115d.          sel → fowfv := elab_WFSelW(fowf),
115e.          inc → fowfv := elab_WFIncW(fowf),
115a.          clo → fowfv := elab_WFCloW(fowf)
108.     end end end
```

```
113.  elab_WFOpnW: WIdx → FoWF →
113.           in,out ch[ wi ] FoWF
113.  elab_WFOpnW(wi)(fowf) ≡
113.     if ∃ ow:OpenW • pre:int_OpnW(ow)(wf) in
113.        then let mkOpnW(p,wn,kv):OpenW •
113.                 pre:int_OpnW(mkOpnW(p,wn,kv))(fowf)
113a.          let w = ch[ wi ]!mkGW(p^⟨wn⟩) ; ch[ wi ]? in
113b.          if w=/error/ then chaos else skip end;
113c.          insert_W(p^⟨wn⟩,kv,w)(fowf) end end
114.        else wf end

114.  elab_WFPutW: WIdx → FoWF → in,out ch[ wi ] FoWF
114.  elab_WFPutW(wi)(fowf) ≡
114.     if ∃ putw•pre:int_WFPutW(puw)(fowf)
114.        then let mkWPW(p,wn):WFPutW •
114.                 pre:int_WFPutW(mkWPW(p,wn))(fowf) in
114a.          let w = s_W(p^⟨wn⟩,fowf) in
114b.          let result = ch[ wi ]!mkDFPW(p^⟨wn⟩,wn,w) ;
114b.              ch[ wi ]? in
114c.          if result = /error/ then chaos else skip end
114.           end end end
114.        else fowf end
```

We need to modify formula Lines 108–114c slightly.

223. **Window Frame Processes, Ω_i:**

a)

b)
c)
d)
e)
f)
g)

223. Ω_i: wi:WIdx → **in**,**out** {wch[wi,dj]|dj:DIdx•dj ∈ dis} **Unit**
223a.
223b.
223c.
223d.
223e.
223f.
223g.

9 Transaction Failure Techniques rollback

rollback

So far we have assumed ideal machine processing: no fall-outs of distributed storage, no data communication failures, etc.

We shall now consider a number of failure issues.

First we quote from Wikipedia: `http://en.wikipedia.org/wiki/Transaction_processing`[26].

9.1 WIKIPEDIA: Transaction Processing Issues

9.1.1 Roll-back

Transaction-processing systems ensure database integrity by recording intermediate states of the database as it is modified, then using these records to restore the database to a known state if a transaction cannot be committed. For example, copies of information on the database prior to its modification by a transaction are set aside by the system before the transaction can make any modifications (this is sometimes called a before image). If any part of the transaction fails before it is committed, these copies are used to restore the database to the state it was in before the transaction began.

9.1.2 Roll-forward

It is also possible to keep a separate journal of all modifications to a database (sometimes called after images); this is not required for rollback of failed transactions, but it is useful for updating the database in the event of a database failure, so some transaction-processing systems provide it. If the database fails entirely, it must be restored from the most recent back-up. The back-up will not reflect transactions committed since the back-up was made. However, once the database is restored, the journal of after images can be applied to the database (roll-forward) to bring the database up to date. Any transactions in progress at the time of the failure can then be rolled back. The result is a database in a consistent, known state that includes the results of all transactions committed up to the moment of failure.

9.1.3 Deadlocks

In some cases, two transactions may, in the course of their processing, attempt to access the same portion of a database at the same time, in a way that prevents them from proceeding. For example, transaction A may access portion X of the database, and transaction B may access portion Y of the database. If, at that point, transaction A then tries to access portion Y of the database while transaction B tries to access portion X, a deadlock occurs, and neither

[26] The almost two page quote from Wikipedia is meant as a working note to the author. I will, eventually get close to a library and study the seminal book of my old colleague at IBM Research, 1970-1973: Jim Gray [26].

transaction can move forward. Transaction-processing systems are designed to detect these deadlocks when they occur. Typically both transactions will be cancelled and rolled back, and then they will be started again in a different order, automatically, so that the deadlock doesn't occur again. Or sometimes, just one of the deadlocked transactions will be cancelled, rolled back, and automatically re-started after a short delay.

Deadlocks can also occur between three or more transactions. The more transactions involved, the more difficult they are to detect, to the point that transaction processing systems find there is a practical limit to the deadlocks they can detect.

9.1.4 Compensating transaction

In systems where commit and rollback mechanisms are not available or undesirable, a Compensating transaction is often used to undo failed transactions and restore the system to a previous state.

9.1.5 ACID criteria (Atomicity, Consistency, Isolation, Durability)

Transaction processing has these benefits:

```
* It allows sharing of computer resources among many users
* It shifts the time of job processing to when the computing
  resources are less busy
* It avoids idling the computing resources without
  minute-by-minute human interaction and supervision
* It is used on expensive classes of computers to help amortize
  the cost by keeping high rates of utilization of those expensive
  resources
* A transaction is an atomic unit of processing.
```

9.2 An Analysis of Δ_0, Δ_j and Ω_i Failures

In principle communication failures, in the form of the absence of expected messages, is a class of failures. We say: "in principle" because such failures are expected masked by the error-correcting data communication "layer" of the underlyig operating and data commun ication systems. We shall, however, bring an analysis of such failures, if for nothing else, as an example of failures. Subsequently we shall cover "more interresting" failures. In our analysis we make the followig assumptions:

- The domain processes — except for the subordinate processes' "own" operations — are fully computerised and without any human interfactions. Failures of the coordinator and the non-"own" aspects of subordinate processes are thus computer failures.

- The user processes are rather completely under human (i.e., interactive) control. That is, non-communication failures of user processes, to simplify matters, are thus human failures.

9.2.1 Communication Failures

Let us analyse what can go wrong during the processing of a transaction request, that is, from the first **"request transaction *'ps'*"** to the final "τ**: path *'p'* window has been stored"**.

224. After a user has issued **"request transaction *'ps'*"** to the coordinator, that user expects an answer from the coordinator:
    a) either **"path(s) *'ps'* are not defined"**
    b) or **"request τ: *'ps'* cannot be accepted"**
    c) or **"request τ: path index map: *'p_to_wi'*"**.

    Failure to receive a response, say within some reasonable time, is considered Failure [224]. Failure [224]

    We suggest to handle Failure [224] as follows ***

225. After the coordinator issues **"please, commit request τ *'ps'*"** to a set of subordinators some of these fail to respond.
    a) Some subordinators may respond: **"yes, able to handle τ"**
    b) Other subordinators may **"no, unable to handle τ"** – in which case the coordinator issues an **"abort τ"** and expects those subordinators which have (already) responded positively to respond with an **"abort τ acknowledgement"**.
    c) If no subordinators responds negatively but only a proper subset of those inquired subordinators respond positively, then we say that Failure [225c] has occurred. Failure [225c]

    We suggest to handle Failure [225c] as follows ***

    d) If some subordinators have responded positively and been sent an **"abort τ"** bu failed to reply with a **"abort τ acknowledgement"** then we say that Failure [225d] has occurred. Failure [225d]

    We suggest to handle Failure [225d] as follows ***

226. After a subordinator acknowledging **"please, commit request τ *'ps'*"** with either a **"no, unable to handle τ"** or a **"yes, able to handle τ"** the subordinator "hears no more", at least within a reasonable time. [The subordinator expects to receive, from the coordinator, either an **"abort τ"** or a **"commit τ"**.] We consider this to be Failure [226]. Failure [226]

    We suggest to handle Failure [226] as follows ***

227. After a user has sent a subordinator a "τ: **get window 'p'**" request it expects either "τ: **cannot find window at 'p'**" or a "τ: **here is path 'p' window: 'w'**" response within a reasonable time. If the user does not receive any response Failure [227] is adjudged to have occurred.

Failure [227]

We suggest to handle Failure [227] as follows ***

228. After a user has sent a "τ: **put window 'w' at 'p'**" request to a subordinator it expects either of a "τ: **cannot store window at 'p'**" or a "τ: **path 'p' window has been stored**" response. If the user does not receive any response Failure [228] is adjudged to have occurred.

Failure [228]

We suggest to handle Failure [228] as follows ***

229. After a user has received a "τ: **here is path 'p' window: 'w'**" response from a subordinator, that subordinator expects, after a reasonable time, say with a day or so, to receive, from the user, "τ: **put window 'w' at 'p'**" request. If the subordinator does not receive such a request Failure [229] is adjudged to have occurred.

Failure [229]

We suggest to handle Failure [229] as follows ***

9.2.2 Computer "Failures"

9.2.3 Human Failures

9.3 Redefinition of Some Functions

9.4 Δ_0: Coordination of Roll-backs/Roll-forwards

9.5 Δ_j: Effectuation of Roll-backs/Roll-forwards

9.6 Ω_i: Transparency of Roll-backs/Roll-forwards

9.7 Optimisation Issues

9.8 Discussion

rds: Wikikedia ces Wikipedia Atomicity analyse ps wi subordinators

10 An SQL-like Query Systems sql

sql

10.1 Clean SQL

Appendix B shows a description, narrative and formal, of a relational query language, `Clean SQL`. It is like `SQL` [33, 41], but relations of `Clean SQL` are sets nut multisets. Thus the *Oracle* implementation, which uses multisets, is not acceptable as an approximate implementation of `Clean SQL`. Instead the *FrontBase*[27] implementation is acceptable. In the Appendix B description relation tuples are sequences of atomic values and there is a notion of tuple types (which are there taken to be the same as relation types). In the window relations, see next, tuples are maps from atomic names to atomic values. Therefore we need a "translation" from atomic names into indices; and we need a "translation" from (stripped) path names into relation identifiers. But first, let us take a look at window values and relation (value)s.

10.2 Window Relations

10.2.1 Tuple and Window Values and Relation Values

We shall show how the 'data structures' of forests of domain and window frames "hide" a set of windows.

60. FoDF = WNm $\overrightarrow{m}$ (W × FoDF)

So does the forest of designated window frames:
Each window "hides" a relation in the form of the tuple and relation values, tval:TVAL,rval:RVAL:

36. W[28] = WNm × WTy × mkWV(ANm-**set**,TVAL,RVAL,WNm-**set**)
11. TVAL = ANm $\overrightarrow{m}$ AVAL
26. RVAL = KVAL $\overrightarrow{m}$ TVAL

230. We can convert a relation value into a set of tuple values.

value
230. convert_rel_tuples: RVAL → TVAL-**set**
230. convert_rel_tuples(rv) ≡ {kv ∪ rv(kv)|kv:KVAL•kv ∈ **dom** rv}

231. A tuple is a map from field names to field values, where field names are atomic names and field values are atomic values.

[27] http://www.frontbase.com/
[28] Here, and elsewhere, from now on, we omit the ′s that normally signify that the type need a well-formedness constraint.

232. A relation is a set of tuples all having the same definition set, with all tuples having, for each attribute name, a value of the same type.

233. From a window, (wn,wt,tv,rv,wns), we extract a relation,

type
231. TVAL = ANm $\overrightarrow{m}$ AVAL
232. REL = TVAL-**set**
axiom
232. ∀ rel:REL,tup,tup′:TVAL • tup ∈ rel ⇒
232. **dom** tup=**dom** tup′
232. ∧ ∀ an:ANm•an ∈ **dom** tup ⇒ xtr_ATyp(tup(an))=xtr_ATyp(tup′(an))
value
233. xtr_REL: W | (TVAL×RVAL) → REL
233. xtr_REL(_,_,tv,rv,_) ≡ {tv} ∪ conv_rel_tuples(rv)
233. xtr_REL(tv,rv) ≡ {tv} ∪ conv_rel_tuples(rv)

10.2.2 Relation Identifiers: Paths and Window Names

234. Relation identifiers form a type of unique tokens.

235. We can postulate a pair of bijective encoding functions

   a) which applies to pairs of paths (of window names) forests of domain frames (of which the path is indeed a path) and yield relation identifiers,

   b) respectively to relation identifiers and yield paths.

type
234. Rn See also Page 136
value
235a. P_to_Rn: P → FoDF $\xrightarrow{\sim}$ Rn
235b. Rn_to_P: Rn → P
axiom
235. ∀ fodf:FoDF,p:P,rn:Rn •
235. p ∈ paths(fodf)
235. ∧ Rn_to_P(P_to_Rn(p)(fodf))= p ∧ P_to_Rn(Rn_to_P(rn))(fodf)=rn

10.2.3 Tuple Attribute Names and Indices

236. We can postulate a function which applies to path names and a forest of domain frames and yields an index map from the attribute (i.w. field) names of the window of at that path in some forest of domain frames to indexes in a linear sequence of attribute values,

a) where these index maps are bijective and where indexes range from 1 to the number of window attributes.

value
236. P_to_AIdx: P → FoDF → (ANm $\overrightarrow{m}$ **Nat**)
axiom
236. P_to_AIdx(p)(fodf) **as** aidx
236. **pre** p ∈ paths(fodf)
236. **post rng** aidx = {1..**card dom** aidx}

10.2.4 A Relational Database

We refer again to Appendix B.

237. A relational database, rdb:RDB, maps relation names, rn:Rn, into pairs of relation types and relations.
    a) Relation types maps (atomic) attribute names into atomic types.
    b) The tuple values of any relation must be of the type of that relation.

type
237. RDB = Rn $\overrightarrow{m}$ (RelTyp × REL)
237a. RelTyp = ANm $\overrightarrow{m}$ ATyp
axiom
237b. ∀ (rt,rv):(RelTyp×REL) •
237b. ∀ t:TUP • tc ∈ rv ⇒
237b. ∀ an:ANm • an ∈ **dom** t ⇒
237b. xtr_ATyp(t(an)) = rt(an)

10.3 The Forest of Window Frames Relations

From a forest of window frames we can extract the set of all relations over that forest.

238. The extraction function of the set of all relations over a forest of domain frames, fodf, is defined as follows:
    a) for every path of fodf
    b) encode that path into its unique relation name, rn, and map that rn into a pair (rt,rv),
    c) where the relation type rt is the window type of,
    d) and the relation value rv is extracted from
    e) the window, w, located by p in fodf.

```
value
238.  xtr_RDB: FoDF → RDB
238.  xtr_RDB(fodf) ≡
238b.     [ (P_to_Rn(p)(fodf))↦(rt,rv)
238a.       | p:P•p ∈ paths(fodf) •
238e.         let (_,wt,(_,tv,rv),_) = s_W(p,fodf) in
238c.         rt = wt
238d.         ∧ rv = xtr_REL(tv,rv) end ]
```

10.4 Conversion to "Clean SQL" Relational Databases

From Appendix B we show its definition of a relational database.

```
type
1.    AVAL ==  mkIV(Int)|mkRV(Rat)|mkBV(Bool)|mkT(Text)|"nil"
2.    ATyp    = {|"int","rat","bool","text","nil"|}
240.  sTVAL   = {| vl:AVAL* • len vl≥1 |}
241.  sTTyp   = {| vtl:ATyp* • len vtl≥1 |}
242.  sREL′    =  sTVAL-set
242.  sREL     =  {| r:sREL′ • wf_sREL(r) |}
243.  Rn
244.  sRDB′    = Rn ~m~> (sTTyp×sREL)
244.  sRDB     = {| rdb:sRDB′ • wf_sRDB(rdb) |}
```

Most of the type definitions can be made to agree with those of this section. The only type definition that differs is the tuple type definition. Here we now make use of the P_to_NATs, Item 236a on the preceding page, function which maps paths into a compact set of indexes.

```
value
238.  xtr_sRDB: FoDF → sRDB
238.  xtr_sRDB(fodf) ≡
238b.     [ (P_to_Rn(p)(fodf))↦(rt,rv)
238a.       | p:P•p ∈ paths(fodf) •
238e.         let (_,wt,(_,tv,rv),_) = s_W(p,fodf) in
238c.         rt = convert_type(wt)(fodf)
238d.         ∧ rv = convert_value(xtr_REL(tv,rv))(fodf) end ]

238c.  convert_type: RelTyp → FoDF → sTTyp
238d.  convert_value: REL → FoDF → sREL
```

10.5 The Forests of Domain Frames Query Language

We have almost "connected" up to the description of Clean SQL (Appendix B). There remains only to show how queries in Clean SQL can be invoked from a forest of window frames or a forest of domain frames process. Based on a forest of domain (or window) frames, fodf:FoDF, and the above (Item 238 on page 112) xtr_sRDB(fodf) one obtains a Clean SQL relational database, srdb:sRDB, with which the query evaluation takes place. Please make note of different relation names: rn:Rn for those of the data base, and rid:Rid for the vurtual ones of the range expression.

type
241. sTTyp = {| vtl:ATyp* • **len** vtl≥1 |}
242. sREL = sTVAL-**set**
243. Rn
244. sRDB = Rn $\overrightarrow{m}$ (sTTyp×sREL)

246. Query = Targ* × (Rid $\overrightarrow{m}$ Range) × Wff
247. Targ == mkRn(ri:Rid) | mkRnIdx(ri:Rid,i:**Nat**)
248. Range == mkRnm(rn:Rn) | mkInfR(lr:Range,o:RelOp,rr:Range)

value
267. E_Query: Query → sRDB → sREL
267a. E_Query(tal,rm,wff)(rdb)

10.6 Discussion

- This discussion

11 A Window Design Tool gui-design

gui-design

11.1 Design Principles

11.2 Graphics

11.3 Syntax

11.4 Commands and Operations

11.4.1 Commands

11.4.2 Operations

11.5 Discussion

con

12 Conclusion con

12.1 Discussion

12.1.1 What Have We Achieved

- To appreciate what we have done is to first understand
  1. that there does not seem to be a general idea, let alone a precise description of what windows are;
  2. that there is a notion of data spaces such as provided by `Linda` [16], `JavaSpaces` [15] and `XVSM` [10, 11, 29, 30];
  3. and that these (1.–2.) are not professionally described.
- Finally one must appreciate the relevance of the question:
  4. *how are data in data spaces first initialised and regularly updated ?*
- This report provides the following answers:
  1. the report suggests a way to describe windows in a proper manner;
  2. the report suggests a way to describe data spaces in a proper manner;
  3. the report shows one such professional approach; and
  4. the report suggests that data initialisation and update occurs via windows.
- We do believe that we have achieved to precisely describe a version of `XVSM` [10, 11, 29, 30]: the query language, but in a far more general form than in [10, 11, 29, 30] and the so-called "application programmers interface[s]": `CAPI-1, CAPI-2, etc.`, again in a precise manner.

12.1.2 What Have We Not Achieved

- We set out, in late April 2010, to describe `XVSM` [10, 11, 29, 30] as it can now be professionally described by several companies around the world, from Japan[29] via Europe[30] to the USofA[31].
- But we failed to complete our task. The available reports were such that we could not do it, from the distance there is between Vienna and Copenhagen. And we were not able to return to Vienna to complete the task there — that might have been possible.
- Instead we "crystallized" an essence of the `Linda, JavaSpaces` and `XVSM` concepts as they are presented in this report.

[29] http://www.csk.com/support_e/vdm/index.html
[30] http://www.altran-praxis.com/whitePapers.aspx, http://www.clearsy.com/ and many others.
[31] http://www.csl.sri.com/programs/formalmethods/

12.1.3 What Should We Do Next

- The report itself needs being reviewed, first by ourself,
  - printing it out, spreading it out on a large desk, carefully checking all formulas, identifying all proof oblgations, etc.;
  - analysing the definitions to possibly identify more general abstractions, both as concerns types — but probably more as concerns function definitions; and
  - "distilling" from the current transaction process descriptions such which capture the essence of transaction processing and proving properties of such "distillates".

12.2 Acknowledgements

The window formalisation dates back some 20 years and is recorded in [6, Examples 19.27–19.28, Pages 435–442]. The connection between window states and WindowSpaces arose as a result of trying to understand [11, 29, 30, 10, XVSM].

12.3 Bibliographical Notes

We have advocated using, in enumerated expressions and statements, as precise a subset of a national language, as here English, as possible, and "pairing" such narrative elements with correspondingly enumerated clauses of a formal specification language. For our formalisations we have used the RAISE formal Specification Language, RSL, [17, 18, 4, 5, 6].

But any of the model-oriented approaches and languages, as offered by Alloy [27], Event B [1], VDM [8, 9, 14] and Z [42], should work as well.

No single one of the above-mentioned formal specification languages, however, suffices. Often one has to carefully combine the above with elements of Petri Nets [38], CSP: Communicating Sequential Processes [21], MSC: Message Sequence Charts [22], Statecharts [20], and some temporal logic, for example either DC: Duration Calculus [43] or TLA+ [31].

Research into how such diverse textual and diagrammatic languages can be meaningfully and proof-theoretically combined is ongoing [2].

The recent book *Logics of Specification Languages* [12] covers ASM, Event B, CafeObj, CASL, Duration Calculus, RAISE, TLA+, VDM and Z; and the recent double journal issue on Formal Methods of Program Development [7] covers Alloy, ASM, Event B, Duration Calculus, RAISE, TLA+, VDM and Z.

References

[1] J.-R. Abrial. The B Book: Assigning Programs to Meanings *and* Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge, England, 1996 and 2009.

[2] K. Araki et al., editors. *IFM 1999–2009: Integrated Formal Methods*, volume 1945, 2335, 2999, 3771, 4591, 5423 (only some are listed) of *Lecture Notes in Computer Science*. Springer, 1999–2009.

[3] D. Beech, editor. *Concepts in User Interfaces: A (VDM) Reference Model for Command and Response Languages*, volume 234 of *Lecture Notes in Computer Science*. Springer, 1986.

[4] D. Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. .

[5] D. Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. Chapters 12–14 are primarily authored by Christian Krog Madsen.

[6] D. Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. .

[7] D. Bjørner. Editor: Special Double Issue on Formal Methods of Program Development. *International Journal of Software and Informatics*, 4(2–3), 2010. Contains papers on Alloy (D.Jackson et al.), ASM (M.Veanes et al.), Event B (D.Méry), RAISE (A.E.Haxthausen), TLA+ (S.Merz et al.), VDM-SL (J.Fitzgerald et al.) and Z (J.Woodcock et al.).

[8] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.

[9] D. Bjørner and C. B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[10] S. Craß. A Formal Model of the Extensible Virtual Shared Memory (XVSM) and its Implementation in Haskell – Design and Specification. M.sc., Technische Universität Wien, A-1040 Wien, Karlsplatz 13, Austria, Febrauary 5 2010.

[11] S. Craß, E. Kühn, and G. Salzer. Algebraic Foundation of a Data Model for an Extensible Space-based Collaboration Protocol. In B. C. Desai, editor, *IDEAS 2009*, pages 301–306, Cetraro, Calabria, Italy, September 16–18 2009.

[12] Dines Bjørner and Martin C. Henson, editor. *Logics of Specification Languages*. EATCS Series, Monograph in Theoretical Computer Science. Springer, Heidelberg, Germany, 2008.

[13] D. Duce, E. Fielding, and L. Marshall. Formal specification and graphic software. Technical report, Rutherford Appleton Laboratory, August 1984.

[14] J. Fitzgerald and P. G. Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, Cambridge, UK, Second edition, 2009.

[15] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Jini Technology Series from Sun Microsystems, Inc. Prentice Hall, June 1999. ISBN: 0-201-30955-6.

[16] D. Gelernter. *Mirrorworlds*. Oxford University Press, 1992.

[17] C. W. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.

[18] C. W. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[19] J. A. Goguen. An introduction to algebraic semiotics, with applications to user interface design. In C. Nehaniv, editor, *Computation for Metaphor, Analogy and Agents*, volume 1562 of *Springer Lecture Notes in Artificial Intelligence*, pages 242–291, 1999.

[20] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[21] C. Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: http://www.usingcsp.com/cspbook.pdf (2004).

[22] ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992, 1996, 1999.

[23] K. E. Iverson. *A Programming Language*. John Wiley and Sons, 1962.

[24] K. E. Iverson. Notation as a tool of thought. *Communications of the ACM*, 23(8):444–465, 1980.

[25] K. E. Iverson and A. D. Falkoff. *A Source Book In APL*. APL Press, 1981.

[26] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan-Kaufman, Redwood, CA, 1992.

[27] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.

[28] C. W. Johnson. Literate specification: Using design rationale to support formal methods in the development of human-machine interfaces. *Human-Computer Interaction*, 11(4):291–320, 1996.

[29] E. Kühn, R. Mordinyi, L. Keszthelyi, and C. Schreiber. Introducing the Concept of Customizable Structued Space for Agent Coordination in the Production of Automation Domain. In S. Decker, Sichman and Castelfranchi, editors, *8th Intl. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS 2009)*, volume 625–632 of *Proceedings of Autonomous Agents and Multi-Agent Systems*, Budapest, Hungary, May 10–15 2009. 8.

[30] E. Kühn, R. Mordinyi, L. Keszthelyi, C. Schreiber, S. Bessler, and S. Tomic. Aspect-oriented Space Containers for Efficient Publish/Subscribe Scenarios in Intelligent Transportation Systems. In T. D. and P. H. Meersmann, editors, *OTM 2009, Part I*, volume 5870 of *LNCS*, pages 432–448. Springer, 2009.

[31] L. Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.

[32] B. Lampson and D. Lomet. A New Presumed Commit Optimization for Two Phase Commit. In *19th VLDB Conference*, 1993.

[33] P. M. Lewis, A. Bernstein, and M. Kifer. *Databases and Transaction Processing: An Application-Oriented Approach*. Addison Wesley, 2002.

[34] L. Marshall. *A Formal Description Method for User Interfaces*. PhD thesis, University of Manchester, Oct. 1986.

[35] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. *ACM SIGOPS Operating Systems Review*, 19(2):40–52, April 1985.

[36] Philip A. Bernstein and Vassos Hadzilacos and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[37] Y. Raz. The Dynamic Two Phase Commitment (D2PC) Protocol. In *Database Theory ICDT '95*, volume 893 of *Lecture Notes in Computer Scienc*, pages 162–176. Springer, 1995. ISBN 978-3-540-58907-5.

[38] W. Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner, 1st edition, 15 June 2010. 248 pages; ISBN 978-3-8348-1290-2.

[39] B. A. Sufrin. Formal methods and the design of effective user interfaces. In M. D. Harrison and A. F. Monk, editors, *People and Computers: Designing for Usability*. Cambridge University Press, 1986.

[40] V. P. Team. Man machine interface: Final specification. Report VIP.T.E.8.3, VIP, Praxis Systems, Bath, England, December 1988.

[41] J. D. Ullman, H. Garcia-Molina, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, Upper Saddle River, NJ, USA, 2001.

[42] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

[43] C. C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real–time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer–Verlag, 2004.

A Definition of Window Frame Process Functions

appWOps

appWOps

A.1 Summary of Window Operations

A.1.1 Sect. 6 Window Operations: Syntax and Signatures

type
79. WFCmd = WFOpnW|WFCloW|WFPutW|WFClkW|WFWrW|WFSelTpl|WFIncTpl
79a. WFOpnW == mkWFOpnW(s_p:P0,s_wn:WNm,s_kv:KeyVAL)
79b. WFCloW == mkWFCloW(s_p:P0,s_wn:WNm)
79c. WFPutW == mkWFPW(s_p:P0,s_wn:WNm)
79d. WFClkW == mkWFClkW(s_p:P0,s_wn:WNm,s_f:FPos)
79d. FPos = ANm | CNm | CNmIx
79e. WFWrW == mkWFWrW(s_p:P0,s_wn:WNm,v:FVAL)
79f. WFSelTpl == mkWFSel(s_p:P0,w_wn:WNm,s_kv:KeyVAL)
79g. WFIncTpl == mkWFInc(s_p:P0,w_wn:WNm)

A.1.2 Sect. 8.5.1 Window Process State Function Signatures Pages 85–87

value
175. Cre_DFW: $\Omega\Sigma \xrightarrow{\sim} \Omega\Sigma$
176. Opn_W: $P \rightarrow \Omega\Sigma \xrightarrow{\sim} \Omega\Sigma$
177. Clo_W: $P \rightarrow \Omega\Sigma \xrightarrow{\sim} \Omega\Sigma$
178. Wri_W: $P \rightarrow FNm \times FVAL \rightarrow \Omega\Sigma \xrightarrow{\sim} \Omega\Sigma$
179. Del_DFW: $\Omega\Sigma \xrightarrow{\sim} \Omega\Sigma$

A.2 "Before" and Now Window Operations

A.2.1 Cre_FoWF: Create Forest of Designated Window Frames

239. The create forest of designated window frames operation is explained as follows.

a) The transaction map, tm, component of the window frame process state contains all the designated paths that (are to be) paths of the designated window frame to be created.[32]
   - The transaction map is tm: UTId $\overrightarrow{m}$ (DP $\overrightarrow{m}$ DIdx).
   - **rng** tm is a value of type (DP $\overrightarrow{m}$ DIdx)**-set**.
   - Taking the ∪ over such a set gives us a map of type (DP $\overrightarrow{m}$ DIdx).
   - Finally taking the **dom**ain of such a map gives us a set of DPs.

b) That designated window frame are characterised by exactly all and only these paths and then that all the windows at the nodes of these paths are unmarked windows.

   The windows at paths which end with a marked window name are those that the user can open (i.e., ge from a domain frame process identified in the transaction map), write to and close (i.e., put back into the domain frame process from whither it came).

c)

d)

value
175. Cre_FoFW: ΩΣ$\xrightarrow{\sim}$ΩΣ
239. Cre_FoFW(_,tm) **as** (fofw,tm)
239a. **let** ps = **dom**(∪(**rng**(tm))) **in**
239b.
239c.
239d.
239. **end**

[32] The almost APL-like [23, 24, 25] one-liner (formula Line 239a.) achieves "assembling" all these paths.

A.2.2 Open Window

Sect. 6.2.1 Open and Insert Windows

```
value
80.  int_WFOpnW: WFOpnW → FoWF → FoDF ~→ FoWF
80.  int_WFOpnW(mkWFOpnW(p,wn,kv))(fowf)(fodf) as fowf′
80.    let ps=paths(fowf),ps′=paths(fowf′) in
80a.    pre p ∈ ps ∧ p^⟨wn⟩ ∉ ps
80b.    post ps′ = ps ∪ {p^⟨wn⟩}
80c.      ∧ let w = eval_DFGetW(mkGW(p^⟨wn⟩,fodf)) in
80d.          fowf′ = WF_InsertW(p^⟨wn⟩,kv,w)(fowf) end
80.    end

81.  WF_Insert_W: P0 × KVAL × W → FoWF ~→ FoWF
81.  WF_Insert_W(p^⟨wn⟩,kv,w)(fowf) as fowf′
81.      let ps=paths(fowf),ps′=paths(fowf′) in
81a.      pre  p^⟨wn⟩ ∉ ps
81b.      post ps′ = ps ∪ {p^⟨wn⟩}
81c.        ∧ ∀ p:P • p ∈ ps ⇒ s_WF(p,fowf)=s_WF(p,fowf′)
81.         ∧ let (wn′,wtyp,mkWV(kn,tpl,rel)) = w in
81f.          let w′ = (wn,wtyp,mkWV(kn,sel_flds(kv,rel,wtyp),rel)) in
81e.           wn=wn′ ∧ dom kv = kn
81d.           w′ = s_W(p^⟨wn⟩,fowf′)
81g.         ∧ s_DF(p^⟨wn⟩,fowf′) = [ ]
81d.      end end end
```

Opn_W: The Window Frame Process

```
value
176.  Opn_W: P→ΩΣ~→ΩΣ
```

A.2.3 Close and Put Windows

Sect. 6.2.2 Close Window (Frame)

value
82. int_WFCloW: WFCloW → FoWF $\xrightarrow{\sim}$ FoWF
82. int_WFCloW(mkWFCloW(p,wn))(fowf) **as** wf′
82. **let** ps = paths(fowf), ps′ = paths(fowf′) **in**
82a. **pre** p⌢⟨wn⟩ ∈ ps
82b. **post** ps′ = ps \ rm_paths(ps)(p⌢⟨wn⟩) ∧ {p⌢⟨wn⟩} ∉ ps
82c. ∧ ∀ p′:P•p′ ∈ ps′•s_WF(p′,fowf)=s_WF(p′,fowf′)
82. **end**

Sect. 6.2.5 Put Window (Frame)

value
86. int_WFPutW: WFPutW → FoWF → FoDF → (FoDF × FoWF)
86. int_WFPutW(mkWFPW(p,wn))(fowf)(fodf) **as** (fodf′,fowf′)
86. **let** ps = paths(fodf) **in**
86a. **pre** {p⌢⟨wn⟩} ∈ ps
86b. **post** fowf′ = fowf
86c. ∧ **let** w = s_W(p⌢⟨wn⟩,wf) **in**
86d. **let** fodf″ = int_DFPutW(mkDPW(p,wn,w))(fodf) **in**
86e. fodf′ = fodf″
86. **end end end**

Clo_W: The Window Frame Process

value
177. Clo_W: P→ΩΣ$\xrightarrow{\sim}$ΩΣ

A.2.4 Click and Write Windows

Sect. 6.2.3: Click Windows

```
value
83.  int_WFClkW: WFClkW → FoWF ~→ FoWF
83.  int_WFClkW(mkWFClkW(p,wn,fp))(fowf) as fowf′
83.    let ps = paths(fowf), ps′ = paths(fowf′),
83a.   pre  {p⌢⟨wn⟩} ∈ ps
83.         ∧ let (wσ:(w,fv,c),wfpwn) = s_Frame(p⌢⟨wn⟩,fowf),
83.           let mkWV(wnb,wtyp,flds,frel,wns) = w in
83b.          appropriate_FPos(fp,flds,wns)
83c.   post ps′ = ps \ rm_paths(ps)(p⌢⟨wn⟩) ∧ p⌢⟨wn⟩∉ ps
83d.         ∀ p′:P•p′ ∈ ps′⇒s_Frame(p′,fowf)=s_Frame(p′,fowf′)
83.        ∧ let (wσ′:(w′,fv′,c′),wfpwn′) = s_Frame(p⌢⟨wn⟩,fowf′) in
83.          let mkWV(wna′,wtyp′,flds′,frel′,wns′) = w′ in
83e.          wnb=wna∧wtyp=wtyp′∧flds=flds′∧frel=frel′∧wns=wns′
83f.       ∧ appropriate_FPos(fp,flds′,wns′)
83g.       ∧ c′=fp
83h.       ∧ fv′ = select_value(flds′)(fp)
83.    end end end end end

83b.,83f.   appropriate_FPos: FPos × Fields × WNm-set → Bool
83b.,83f.   appropriate_FPos(fp,flds,wns) ≡
83b.,83f.     case fp of
83b.,83f.       mkCNmIx(cnm,x)
83b.,83f.         → appropriate_FPos(mkCNm(cnm),flds,wns)
83b.,83f.           ∧ x ∈ inds flds(mkCNm(cnm)),
83b.,83f.       _ → fp ∈ dom flds ∪ wns
83b.,83f.     end

83d.  select_value: Fields → FPos ~→ FV
83d.  select_value(flds)(fp) ≡
83d.    case fp of
83d.      mkCNmIx(cnm,x) → (flds(mkCNm(cnm)))(x),
83d.      _ → flds(fp)
83d.    end
```

Sect. 6.2.4: Write Windows

```
value
84.  int_WFWrW: WFWrW → FoWF ~→ FoWF
84.  int_WFWrW(mkWFWrW(p,wn,fv))(fowf) as fowf′
84.    let ps = paths(fowf), ps′ = paths(fowf′),
84a.   pre  p⌢⟨wn⟩ ∈ ps
84.         ∧ let (wσ:(w,fv,c),wfpwn) = s_Frame(p⌢⟨wn⟩,fowf),
```

84. **let** mkWV(wnb,wtyp,flds,frel,wns) = w **in**
84b. appropriate_FPos(fp,fields,{})
84c. ∧ sub_type(xtr_typ(fv),wtyp(c)) **[check!]**
84d. **post** ps′ = ps
84e. ∧ ∀ p′:P•p′ ∈ ps′\{p⌢⟨wn⟩}⇒s_Frame(p′,fowf)=s_Frame(p′,fowf′)
84. ∧ **let** (wσ′:(w′,fv′,c′),wfpwn′) = s_Frame(p⌢⟨wn⟩,fowf′) **in**
84. **let** mkWV(wna′,wtyp′,flds′,frel′,wns′) = w′ **in**
84f. wnb=wna ∧ wtyp=wtyp′ ∧ **dom** flds=**dom** flds′ ∧ frel=frel′ ∧
84g. ∧ flds′ = update_field(flds,c,fv)
84h. ∧ c′=c
84. **end end end end end**

Wri_W: The Window Frame Process

value
178. Wri_W: FNm×FVAL→ΩΣ~→ΩΣ

A.2.5 Del_DFW: Delete Designated Window Frame

value
179. Del_DFW: $\Omega\Sigma \xrightarrow{\sim} \Omega\Sigma$

B Clean SQL

appSQL

appSQL

B.1 Semantic Types

B.1.1 Types

We use several of the value and type concepts of earlier. A few need be recast.

240. A sequential tuple value is a sequence of atomic values.
241. A sequential tuple types is a sequence of atomic types.
242. A sequential relation is a set of sequential tuples all of the same non-zero length.
243. Relation names are further undefined quantities.
244. A relational data base maps relation names into pairs of tuple types and relations such that types fits relations.

type
1. AVAL == mkIV(**Int**)|mkRV(**Rat**)|mkBV(**Bool**)|mkT(**Text**)|″nil″
2. ATyp = {|″int″,″rat″,″bool″,″text″,″nil″|}
240. sTVAL = {| vl:AVAL* • **len** vl≥1 |}
241. sTTyp = {| vtl:ATyp* • **len** vtl≥1 |}
242. sREL′ = sTVAL-**set**
242. sREL = {| r:sREL′ • wf_sREL(r) |}
243. Rn
244. sRDB′ = Rn $\overrightarrow{m}$ (sTTyp×sREL)
244. sRDB = {| rdb:sRDB′ • wf_sRDB(rdb) |}

B.1.2 Semantic Well-formedness

value
242. wf_sREL: sREL′ → **Bool**
242. wf_sREL(r) ≡ ∀ t,t′:sTVAL•{t,t′}⊆r⇒0<**len** t=**len** t′>0

244. wf_sRDB: sRDB′ → **Bool**
244. wf_sRDB(rdb) ≡
244. ∀ (tt,rel):(sTTyp×sREL) • (tt,rel)∈ **rng** rdb⇒wf_sR(tt,rel)

244. wf_sR: sTTyp × sREL → **Bool**
244. wf_sR(tt,rel) ≡
244. ∀ t:sTVAL • t ∈ rel ⇒
244. **len** t = **len** tt ∧ ∀ i:**Nat** • i ∈ **inds** t ⇒ tt(i)=xtr_ATyp(t(i))

B.2 Syntactics

The motivation for the query language of SQL-like RDMSs is found in the usual set comprehension expression:

{ (a,b,...,c) | a:A,b:B,...,c:C • p(a,b,...,c) }

Either the individual *a*s, *b*s, ..., *c*s of the above set comprehension are element values of tuples or subsequences of these are tuple values of relations. That is, the individual *A*s, *B*s, ..., *C*s of the above set comprehension are attribute names, or subsequences of these are relation names. The predicates p are just that. The SQL-like query language of the RDMS being illustrated in this section now follows the above schematised set comprehension.

245. The token types Rid and Tid model various kinds of (free) identifiers. Rids stand for relations (defined in range expressions) and Tids stand for tuples (defined in quantified expressions, see below).

246. A query consists of three parts:
    a) a target list, corresponding to the (a,b,...,c) expression of the set comprehension,
    b) a binding of variable identifiers to relations and
    c) the (well-formed formula, Wff) corresponding to the p(a,b,...,c) predicate of the set comprehension.

247. The target list consists of either relation names or of tuple element index qualified relation names.

248. A range expression either names a relation or is an infix expression denoting the union, the intersection or the complement of of the value of two range expressions.

249. A well-formed formula is either a quantified, or an infix, or a negated, or an atomic well-formed formula.
    a) A quantified well-formed formula indicates the quantifier ($\forall, \exists$), identifies, by t:Tid, the name of an arbitrary, the quantified tuple in a named relation (rn:Rn), and states the subsidiary well-formed formula (in which t is expected to occur free and range over the tuples of the named relation, rn:Rn).
    b) An infix well-formed formula expresses the conjunction (AND) or the disjunction (OR) of two subsidiary well-formed formulas.
    c) A negated well-formed formula expresses the negation of a (the subsidiary) well-formed formula.
    d) An atomic well-formed formula expresses an arithmetic relation (`less than`, `less than or equal`, `equal`, `not equal`, `larger than or equal`, or `larger than`) between two term values.

250. A term expression is either a simple atomic value or stands for an indexed tuple element value.

251. An indexed tuple element value names a range expression relation and gives an index (into tuples of that range relation).

252. The name of a range expression relations is either a range relation identifier or a range tuple identifier.

type
245. Rid, Tid
246. Query′ = Targ* × (Rid $\overrightarrow{m}$ Range) × Wff
246. Query = {| q:Query′ • wf_Query(q) |}
247. Targ == mkRn(ri:Rid) | mkRnIdx(ri:Rid,i:**Nat**)
248. Range == mkRnm(rn:Rn) | mkInfR(lr:Range,o:RelOp,rr:Range)
248. RelOp == UNION | INTER | COMPL
249. Wff = QPre | IPre | NPre | APre
249a. QPre == mkQ(q:Quan,ti:Tid,rn:Rnm,cond:Wff)
249a. Quan == ALL | EXISTS
249b. IPre == mkI(lp:Wff,ao:BOp,rp:Wff)
249b. BOp == AND | OR
249c. NPre == mkN(pr:Wff)
249d. APre == mkA(lt:Term,ar:ARel,rt:Term)
249d. ARel == LESSEQ | LESS | EQUAL | NOTEQ | LARG | LARGEQ
250. Term = AVAL | Elem
251. Elem == mkE(vt:RTid,i:**Nat**)
252. RTid == mkR(ri:Rid) | mkT(ti:Tid)

B.3 Semantics

B.3.1 Semantic Well-formedness

253. The function attributes yields a set of attributes of named relations.

254. For a query to be evaluated its precondition for evaluation must hold:

    a) the target list must be well-formed,
    b) the range expressions must be well-formed and
    c) the predicate (wff) must be well-formed.

255. The well-formedness of these syntactic quantities depends on a context.

    a) For target list well-formedness the context is a dictionary, Δ, which maps identifiers of tuples into index sets.
    b) For the range expression well-formedness the context is the relational database.

c) And for the predicate well-formedness the context is both the dictionary mentioned above and the database.

256. The function dict creates from the range expressions and the database a dictionary. It does so using the auxiliary function attrs. For every relation name, rn, of the range expression map, rm, attrs extracts the attribute names for that relation.

257. The function attrs should be reasonably self-explanatory.

258. A target list is well-formed if all of its target expressions are well-formed in the same dictionary context.

    a) A target expression is either a simple relation identifier which must then be defined in the dictionary,

    b) or it is a pair of a relation identifier and an attribute name where the formed must be defined in the dictionary and the latter must be in the definition set of attribute names for that relation identifier.

259. The range expression map (from relation identifiers to range expressions) is well-formed if all of the range expressions are well-formed.

    a) A range expression is either just the name of a relation which must then be defined in the database,

    b) or it is an infix range expression both of whose range expressions must be well-formed.

260. The well-formedness of a predicate expression wff depends on the kind of expression it is.

    a) If wff is a quantified expression then its relation name, rn, must be defined in the database and the contained wff, pr, must be well-formed in a context which keeps the database but updates the dictionary to map the quantified tuple variable to the set of attribute names of the relation rn.

    b) If wff is an infix predicate expression then both predicate expression operands must be well-formed.

    c) If wff is a negated predicate expression then that predicate expression operand must be well-formed.

    d) Finally, if wff is an atomic expression of two terms (and an arithmetic relation operator) then the two terms must be well-formed.

261. The well-formedness of a term depends on the kind of expression it is.

    a) A simple term value is always well-formed.

    b) A term reference of a relation or a tuple identifier and a tuple element index is well-formed if

i. the relation or a tuple identifier is defined in the dictionary,
ii. and the index is in the definition set of that identifier (in the dictionary).

```
value
253. attributes: Rnm → sRDB → Nat-set
253. attributes(rn)(rdb) ≡ let (tt,_) = rdb(rn) in inds tt end

254. pre_E_Query: Query → sRDB → Bool
254. pre_E_Query(tal,rm,wff) ≡
254a.      wf_Targl(tal)(dict(rm,rdb))
254b.    ∧ wf_Ranges(rm)(rdb)
254c.    ∧ wf_Wff(wff)(rdb)(dict(rm,rdb))

type
255a. Δ = (Rid|Tid) ~m~> Nat-set

value
256. dict: (Rid ~m~> Range) × sRDB → Δ
256. dict(rm,rdb) ≡ [ ri↦attrs(rm(ri),rdb)|ri:Rid•ri ∈ dom rm ]

257. attrs: Range × RDB → Nat-set
257. attrs(range,rdb) ≡
257.   case range of
257.     mkRnm(rn)→attributes(rnm)(rdb),
257.     mkInfR(lr,_,_)→attrs(lr,rdb)
257.   end

258. wf_Targl: Targ* → Δ → Bool
258. wf_Targl(tal)(δ) ≡ ∀ t:Targ • t ∈ elems tal ⇒ wf_Targ(t)

258. wf_Targ: Targ → Δ → Bool
258. wf_Targ(t)(δ) ≡
258.   case t of
258a.     mkRn(ri) → ri ∈ dom δ,
258b.     mkRnAn(ri,i) → ri ∈ dom δ ∧ i ∈ δ(ri)
258.   end

259. wf_Ranges: (Rid ~m~> Range) → sRDB → Bool
259. wf_Ranges(rm)(rdb) ≡
259.   ∀ range:Range • range ∈ rng rm ⇒ wf_Range(rng)(rdb)

259. wf_Range: Range → sRDB → Bool
259. wf_Range(range)(rdb) ≡
259.   case range of
```

```
259a.      mkRnm(rn) →
259a.        rn ∈ dom rdb,
259b.      mkInfR(lr,_,rr) →
259b.        wf_Ranges(lr)(rdb)∧wf_Ranges(rr)(rdb)
259b.      ∧ attrs(lr)(rdb)=attrs(rr)(rdb)
259.   end

260. wf_Wff: Wff → RDB → Δ → Bool
260. wf_Wff(wff)(rdb)(δ) ≡
260.   case wff of
260a.     mkQ(_,ti,rn,pr) →
260a.       rn ∈ dom rdb
260a.       ∧ wf_Wff(pr)(δ†[ti↦attrs(rn)(rdb)]),
260b.     mkI(lp,_,rp) →
260b.       wf_Wff(lp)(rdb)(δ) ∧ wf_Wff(rp)(rdb)(δ),
260c.     mkN(pr) →
260c.       wf_Wff(pr)(rdb)(δ),
260d.     mkA(lt,ar,rt) →
260d.       wfTerm(lt)(δ)∧wf_Term(rt)(δ)
260.   end

261. wf_Term: Term → Δ → Bool
261. wf_Term(trm)(δ) ≡
261.   case trm of
261a.     mkV(_) → true,
261(b)i.    mkE(mkR(ri),i) → ri ∈ dom δ ∧ i ∈ δ(ri),
261(b)ii.   mkE(mkT(ti),i) → ti ∈ dom δ ∧ i ∈ δ(ti)
261.   end
```

B.3.2 Auxiliary Functions

262. V_Rs stands for virtual relations.

263. VRs stands for sets of virtual tuples.

264. G: For each combination of (a1,b1j),...,(an,bnj) in
[a1↦{b11,...,b1m1},a2↦{b21,...,b2m2},...,an↦{bn1,... bnmn}]
G delivers the map [a1↦b1j,...,an↦bnj] in the set G(rm) of such maps.

265. C: For a pair of a list ⟨a′,a″,...,a‴⟩ and a map m:[a1↦b1j,...,an↦bnj] C delivers a tuple ⟨ m(a′),m(a″),...,m(a‴)⟩.

266. Conc take a list of tuples and produces a tuple.

```
type
262. V_Rs = Rid ~m~> sREL
```

263. VRs = (Rid $\overrightarrow{m}$ sTVAL)**-set**

value
264. G: V_Rs → VRs
264. G(vrs) ≡
264. **if** vrs=[ ] **then** {[ ]}
264. **else** { [v↦t] ∪ m | v:Rid,t:sTVAL •
264. v ∈ **dom** vrs∧t ∈ vrs(v)∧m ∈ G(vrs\{v})} **end**

265. C: Targ* × VRs → sTVAL*
265. C(tal)(vrs) ≡
265. ⟨ **case** tal(i) **of**
265. mkRn(rn) → vrs(rn),
265. mkRnAn(rn,i) → ⟨(vrs(rn))(i)⟩ **end** | i **in** [1..**len** tal] ⟩

266. Conc: sTVAL* → sTVAL
266. Conc(tupl) ≡ **if** tupl=⟨⟩ **then** ⟨⟩ **else hd** tupl ⌢ Conc(**tl** tupl) **end**

B.3.3 Evaluation Functions

267. The evaluation of a query, E_Query, results in a relation.
    a) Query evaluation takes place in the context of the state of the database,
    b) and makes use of an auxiliary evaluation function E_Pred, and the auxiliary functions G, C and Conc.
        - G is applied to the range definitions and yields a set of mappings from range identifiers to tuples.
        - For each such mapping, m, E_Pred is applied to the wff and the database. If true, then the mapping m is made into a tuple of tuples by means of the target list and using function C.
        - Finally the tuple of tuples is "straightened out" into a simple tuple using function Conc.
        - And this is done for all mappings m, hence generating a set of simple tuples, i.e., a relation.

268. Evaluation, E_Pred, of a predicate proceeds according to the form of the predicate.
    a) The predicate wff:mkQ(ALL,ti,rn,pr) holds if the predicate pr holds for ti being bound to every tuple, tup, in the rn-named relation in database rdb.
    b) The predicate, wff:mkQ(EXISTS,ti,rn,pr) holds if the predicate pr holds for ti being bound to some tuple, tup, in the rn-named relation.
    c) The predicate wff:mkI(lp,AND,rp) holds if both the predicates lp and rp holds.

d) The predicate wff:mkI(lp,OR,rp) holds if either of the predicates lp or rp hold.

e) The predicate wff:mkN(pr) holds if pr does not hold.

f) The atomic expression wff:mkA(lt,ao,rt) holds if the values of the terms lt and rt stand in the relation designated by the arithmetic relation operator ao.

269. Evaluation, E_Range, of a range expressions seems pretty obvious.

270. Evaluation, E_Term, of a term, t, proceeds according to the form of the term t:

a) The term t:mkV(v) evaluates to that value v.

b) The term t:mkE(vt,i) evaluates to the i indexed tuple value designated by vt.

c) The term t:mkT(ti) evaluates to the i indexed tuple value designated by ti.

```
value
267. E_Query: Query → sRDB → sREL
267a. E_Query(tal,rm,wff)(rdb) ≡
267.   let v_rs = [ v↦E_Range(rm(v))(rdb)|v:Vid•v ∈ dom rm ] in
267b.   {Conc(C(tal,m))|m:VRs•m ∈ G(v_rs)∧E_Pred(wff)(m)(rdb)}
267.   end

268. E_Pred: Wff → VRs → sRDB → Bool
268. E_Pred(wff)(vrs)(rdb) ≡
268.   case wff of
268.     mkQ(q,ti,rn,pr) →
268.       let (_,rel) = rdb(rn) in
268.       case q of
268a.         ALL → ∀ tup:sTVAL • tup ∈ rel
268a.                 ∧ E_Pred(pr)(vrs † [ti↦tup])(rdb),
268b.         EXISTS → ∃ tup:sTVAL • tup ∈ rel
268b.                 ∧ E_Pred(pr)(vrs † [ti↦tup])(rdb)
268.       end end
268c.     mkI(lp,bo,rp) →
268c.       let lb = E_Pred(lp)(vrs)(rdb), rb = E_Pred(rp)(vrs)(rdb) in
268d.       case bo of AND → lb∧rb, → OR → lb∨rb end,
268e.     mkN(pr) → ∼E_Pred(pr)(vrs)(rdb),
268f.     mkA(lt,ao,rt) →
268f.       let lv = E_Term(lt)vrs, rv = E_Term(rt)vrs in
268f.       case ao of
268f.         LESSEQ → lv≤rv, LESS → lv<rv, EQUAL → lv=rv,
268f.         NOTEQ → lv≠rv, LARG → lv>rv, LARGEQ → lb≥rv
```

```
268.   end end end end

269. E_Range: Range → sRDB → sREL
269. E_Range(re)(rdb) ≡
269.   case re of
269.     mkRnm(rn) → rdb(rn),
269.     mkInfR(lr,o,rr) → let lrel = E_Range(lr)(rdb),
269.                           rrel = E_Range(rr)(rdb) in
269.                       case o of
269.                         UNION → lrel ∪ rrel,
269.                         INTER → lrel ∩ rrel,
269.                         COMPL → lrel \ rrel
269.   end                 end end

270. E_Term: Term → VRs → AVAL
270. E_Term(t)(vrs) ≡
270.   case t of
270a.     mkV(v) → v,
270b.     mkE(vt,i) → (vrs(vt))(i),
270c.     mkT(ti) → (vrs(ti))(i),
270.   end
```

C An RSL Primer

This is an ultra-short introduction to the RAISE Specification Language, RSL. Examples follow and expand on the examples of earlier sections.

C.1 Types

The reader is kindly asked to study first the decomposition of this section into its sub-parts and sub-sub-parts.

C.1.1 Type Expressions

Type expressions are expressions whose values are types, that is, possibly infinite sets of values (of "that" type).

Atomic Types Atomic types have (atomic) values. That is, values which we consider to have no proper constituent (sub-)values, i.e., cannot, to us, be meaningfully "taken apart".

RSL has a number of *built-in* atomic types. There are the Booleans, integers, natural numbers, reals, characters, and texts.

Basic Types

type
[1] **Bool**
[2] **Int**
[3] **Nat**
[4] **Real**
[5] **Char**
[6] **Text**

1. The Boolean type of truth values **false** and **true**.
2. The integer type on integers ..., –2, –1, 0, 1, 2, ...
3. The natural number type of positive integer values 0, 1, 2, ...
4. The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period ("."), followed by a natural number (the fraction).
5. The character type of character values ″a″, ″b″, ...
6. The text type of character string values ″aa″, ″aaa″, ..., ″abc″, ...

Composite Types

Composite types have composite values. That is, values which we consider to have proper constituent (sub-)values, i.e., can, to us, be meaningfully "taken apart".

From these one can form type expressions: finite sets, infinite sets, Cartesian products, lists, maps, etc.

Let A, B and C be any type names or type expressions, then:

Composite Type Expressions

[7] A-set
[8] A-infset
[9] A × B × ... × C
[10] A*
[11] A^ω
[12] A $\xrightarrow[m]{}$ B
[13] A → B
[14] A $\xrightarrow{\sim}$ B
[15] (A)
[16] A | B | ... | C
[17] mk_id(sel_a:A,...,sel_b:B)
[18] sel_a:A ... sel_b:B

7. The set type of finite cardinality set values.
8. The set type of infinite and finite cardinality set values.
9. The Cartesian type of Cartesian values.
10. The list type of finite length list values.
11. The list type of infinite and finite length list values.
12. The map type of finite definition set map values.
13. The function type of total function values.
14. The function type of partial function values.
15. In (A) A is constrained to be:
    - either a Cartesian B × C × ... × D, in which case it is identical to type expression kind 9,
    - or not to be the name of a built-in type (cf., 1–6) or of a type, in which case the parentheses serve as simple delimiters, e.g., (A $\xrightarrow[m]{}$ B), or (A*)-set, or (A-set)list, or (A|B) $\xrightarrow[m]{}$ (C|D|(E $\xrightarrow[m]{}$ F)), etc.
16. The postulated disjoint union of types A, B, ..., and C.
17. The record type of mk_id-named record values mk_id(av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.
18. The record type of unnamed record values (av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

C.1.2 Type Definitions

Concrete Types Types can be concrete in which case the structure of the type is specified by type expressions:

Type Definition

type
A = Type_expr

Variety of Type Definitions

[1] Type_name = Type_expr /∗ without |s or subtypes ∗/
[2] Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[3] Type_name ==
mk_id_1(s_a1:Type_name_a1,...,s_ai:Type_name_ai) |

... |
mk_id_n(s_z1:Type_name_z1,...,s_zk:Type_name_zk)
[4] Type_name :: sel_a:Type_name_a ... sel_z:Type_name_z
[5] Type_name = {| v:Type_name′ • $\mathcal{P}$(v) |}

where a form of [2–3] is provided by combining the types:

Record Types

Type_name = A | B | ... | Z
A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
...
Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)

Types A, B, ..., Z are disjoint, i.e., shares no values, provided all mk_id_k are distinct and due to the use of the disjoint record type constructor ==.

axiom
∀ a1:A_1, a2:A_2, ..., ai:Ai •
s_a1(mk_id_1(a1,a2,...,ai))=a1 ∧ s_a2(mk_id_1(a1,a2,...,ai))=a2 ∧
... ∧ s_ai(mk_id_1(a1,a2,...,ai))=ai ∧
∀ a:A • **let** mk_id_1(a1′,a2′,...,ai′) = a **in**
a1′ = s_a1(a) ∧ a2′ = s_a2(a) ∧ ... ∧ ai′ = s_ai(a) **end**

Subtypes

In RSL, each type represents a set of values. Such a set can be delimited by means of predicates. The set of values b which have type B and which satisfy the predicate $\mathcal{P}$, constitute the subtype A:

Subtypes

type
A = {| b:B • $\mathcal{P}$(b) |}

Sorts — Abstract Types

Types can be (abstract) sorts in which case their structure is not specified:

Sorts

type
A, B, ..., C

C.2 Concrete RSL Types: Values and Operations

C.2.1 Arithmetic

Arithmetic

type
  Nat, **Int**, **Real**
value
  +,−,∗: **Nat**×**Nat**→**Nat** | **Int**×**Int**→**Int** | **Real**×**Real**→**Real**
  /: **Nat**×**Nat**$\xrightarrow{\sim}$**Nat** | **Int**×**Int**$\xrightarrow{\sim}$**Int** | **Real**×**Real**$\xrightarrow{\sim}$**Real**
  <,≤,=,≠,≥,> (**Nat**|**Int**|**Real**) × (**Nat**|**Int**|**Real**) → **Bool**

C.2.2 Set Expressions

Set Enumerations Let the below a's denote values of type A, then the below designate simple set enumerations:

Set Enumerations

{{}, {a}, {e$_1$,e$_2$,...,e$_n$}, ...} ⊆ A-**set**
{{}, {a}, {e$_1$,e$_2$,...,e$_n$}, ..., {e$_1$,e$_2$,...}} ⊆ A-**infset**

Set Comprehension

The expression, last line below, to the right of the ≡, expresses set comprehension. The expression "builds" the set of values satisfying the given predicate. It is abstract in the sense that it does not do so by following a concrete algorithm.

Set Comprehension

type
  A, B
  P = A → **Bool**
  Q = A $\xrightarrow{\sim}$ B
value
  comprehend: A-**infset** × P × Q → B-**infset**
  comprehend(s,P,Q) ≡ { Q(a) | a:A • a ∈ s ∧ P(a)}

C.2.3 Cartesian Expressions

Cartesian Enumerations Let e range over values of Cartesian types involving A, B, ..., C, then the below expressions are simple Cartesian enumerations:

Cartesian Enumerations

type
A, B, ..., C
A × B × ... × C
value
(e1,e2,...,en)

C.2.4 List Expressions

List Enumerations Let a range over values of type A, then the below expressions are simple list enumerations:

List Enumerations

{⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ...} ⊆ A*
{⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ..., ⟨e1,e2,...,en,... ⟩, ...} ⊆ A^ω

⟨ a_*i* .. a_*j* ⟩

The last line above assumes a_i and a_j to be integer-valued expressions. It then expresses the set of integers from the value of e_i to and including the value of e_j. If the latter is smaller than the former, then the list is empty.

List Comprehension

The last line below expresses list comprehension.

List Comprehension

type
A, B, P = A → **Bool**, Q = A $\xrightarrow{\sim}$ B
value
comprehend(l,P,Q) ≡
⟨ Q(l(i)) | i **in** ⟨1..**len** l⟩ • P(l(i))⟩

C.2.5 Map Expressions

Map Enumerations Let (possibly indexed) u and v range over values of type $T1$ and $T2$, respectively, then the below expressions are simple map enumerations:

Map Enumerations

type
T1, T2
M = T1 $\overrightarrow{m}$ T2
value
u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
{[ ], [u↦v], ..., [u1↦v1,u2↦v2,...,un↦vn],...} ⊆ M

Map Comprehension

The last line below expresses map comprehension:

Map Comprehension

type
U, V, X, Y
M = U $\overrightarrow{m}$ V
F = U $\xrightarrow{\sim}$ X
G = V $\xrightarrow{\sim}$ Y
P = U → **Bool**
value
comprehend: M×F×G×P → (X $\overrightarrow{m}$ Y)
comprehend(m,F,G,P) ≡
[F(u) ↦ G(m(u)) | u:U • u ∈ **dom** m ∧ P(u)]

C.2.6 Set Operations

Set Operator Signatures

Set Operations

value
19 ∈: A × A-**infset** → **Bool**
20 ∉: A × A-**infset** → **Bool**
21 ∪: A-**infset** × A-**infset** → A-**infset**
22 ∪: (A-**infset**)-**infset** → A-**infset**
23 ∩: A-**infset** × A-**infset** → A-**infset**
24 ∩: (A-**infset**)-**infset** → A-**infset**
25 \: A-**infset** × A-**infset** → A-**infset**
26 ⊂: A-**infset** × A-**infset** → **Bool**
27 ⊆: A-**infset** × A-**infset** → **Bool**
28 =: A-**infset** × A-**infset** → **Bool**
29 ≠: A-**infset** × A-**infset** → **Bool**
30 **card**: A-**infset** $\xrightarrow{\sim}$ **Nat**

Set Examples

Set Examples

examples
a ∈ {a,b,c}
a ∉ {}, a ∉ {b,c}
{a,b,c} ∪ {a,b,d,e} = {a,b,c,d,e}
∪{{a},{a,b},{a,d}} = {a,b,d}
{a,b,c} ∩ {c,d,e} = {c}
∩{{a},{a,b},{a,d}} = {a}
{a,b,c} \ {c,d} = {a,b}
{a,b} ⊂ {a,b,c}
{a,b,c} ⊆ {a,b,c}
{a,b,c} = {a,b,c}
{a,b,c} ≠ {a,b}
card {} = 0, **card** {a,b,c} = 3

Informal Explication

19. ∈: The membership operator expresses that an element is a member of a set.
20. ∉: The nonmembership operator expresses that an element is not a member of a set.
21. ∪: The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.
22. ∪: The distributed prefix union operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
23. ∩: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
24. ∩: The prefix distributed intersection operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
25. \: The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.
26. ⊆: The proper subset operator expresses that all members of the left operand set are also in the right operand set.
27. ⊂: The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.

28. =: The equal operator expresses that the two operand sets are identical.
29. ≠: The nonequal operator expresses that the two operand sets are *not* identical.
30. **card**: The cardinality operator gives the number of elements in a finite set.

Set Operator Definitions

The operations can be defined as follows (≡ is the definition symbol):

Set Operation Definitions

```
value
  s′ ∪ s″ ≡ { a | a:A • a ∈ s′ ∨ a ∈ s″ }
  s′ ∩ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∈ s″ }
  s′ \ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∉ s″ }
  s′ ⊆ s″ ≡ ∀ a:A • a ∈ s′ ⇒ a ∈ s″
  s′ ⊂ s″ ≡ s′ ⊆ s″ ∧ ∃ a:A • a ∈ s″ ∧ a ∉ s′
  s′ = s″ ≡ ∀ a:A • a ∈ s′ ≡ a ∈ s″ ≡ s⊆s′ ∧ s′⊆s
  s′ ≠ s″ ≡ s′ ∩ s″ ≠ {}
  card s ≡
    if s = {} then 0 else
    let a:A • a ∈ s in 1 + card (s \ {a}) end end
    pre s /∗ is a finite set ∗/
  card s ≡ chaos /∗ tests for infinity of s ∗/
```

C.2.7 Cartesian Operations

Cartesian Operations

```
type
  A, B, C
  g0: G0 = A × B × C
  g1: G1 = ( A × B × C )
  g2: G2 = ( A × B ) × C
  g3: G3 = A × ( B × C )

value
  va:A, vb:B, vc:C, vd:D
  (va,vb,vc):G0,
  (va,vb,vc):G1
  ((va,vb),vc):G2
  (va3,(vb3,vc3)):G3

decomposition expressions
  let (a1,b1,c1) = g0,
      (a1′,b1′,c1′) = g1 in .. end
  let ((a2,b2),c2) = g2 in .. end
  let (a3,(b3,c3)) = g3 in .. end
```

C.2.8 List Operations

List Operator Signatures

List Operations

value
hd: $A^\omega \xrightarrow{\sim} A$
tl: $A^\omega \xrightarrow{\sim} A^\omega$
len: $A^\omega \xrightarrow{\sim}$ **Nat**
inds: $A^\omega \to$ **Nat-infset**
elems: $A^\omega \to$ **A-infset**
.(.): $A^\omega \times$ **Nat** $\xrightarrow{\sim} A$
$\frown$: $A^* \times A^\omega \to A^\omega$
=: $A^\omega \times A^\omega \to$ **Bool**
$\neq$: $A^\omega \times A^\omega \to$ **Bool**

List Operation Examples

List Examples

examples
hd⟨a1,a2,...,am⟩=a1
tl⟨a1,a2,...,am⟩=⟨a2,...,am⟩
len⟨a1,a2,...,am⟩=m
inds⟨a1,a2,...,am⟩={1,2,...,m}
elems⟨a1,a2,...,am⟩={a1,a2,...,am}
⟨a1,a2,...,am⟩(i)=ai
⟨a,b,c⟩$\frown$⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
⟨a,b,c⟩=⟨a,b,c⟩
⟨a,b,c⟩ ≠ ⟨a,b,d⟩

Informal Explication

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i, gives the ith element of the list.

- $\hat{}$: Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- =: The equal operator expresses that the two operand lists are identical.
- ≠: The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

List Operator "Definitions"

```
value
  is_finite_list: A^ω → Bool

  len q ≡
    case is_finite_list(q) of
      true → if q = ⟨⟩ then 0 else 1 + len tl q end,
      false → chaos end

  inds q ≡
    case is_finite_list(q) of
      true → { i | i:Nat • 1 ≤ i ≤ len q },
      false → { i | i:Nat • i≠0 } end

  elems q ≡ { q(i) | i:Nat • i ∈ inds q }


  q(i) ≡
    case (q,i) of
      (⟨⟩,1) → chaos,
      (_,1) → let a:A,q′:Q • q=⟨a⟩^q′ in a end
      _ → q(i−1)
    end

  fq ^ iq ≡
      ⟨ if 1 ≤ i ≤ len fq then fq(i) else iq(i − len fq) end
       | i:Nat • if len iq≠chaos then i ≤ len fq+len end ⟩
    pre is_finite_list(fq)

  iq′ = iq″ ≡
    inds iq′ = inds iq″ ∧ ∀ i:Nat • i ∈ inds iq′ ⇒ iq′(i) = iq″(i)

  iq′ ≠ iq″ ≡ ∼(iq′ = iq″)
```

C.2.9 Map Operations

Map Operator Signatures and Map Operation Examples

value

m(a): M → A $\xrightarrow{\sim}$ B, m(a) = b

dom: M → A-**infset** [domain of map]
dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}

rng: M → B-**infset** [range of map]
rng [a1↦b1,a2↦b2,...,an↦bn] = {b1,b2,...,bn}

†: M × M → M [override extension]
[a↦b,a′↦b′,a″↦b″] † [a′↦b″,a″↦b′] = [a↦b,a′↦b″,a″↦b′]

∪: M × M → M [merge ∪]
[a↦b,a′↦b′,a″↦b″] ∪ [a‴↦b‴] = [a↦b,a′↦b′,a″↦b″,a‴↦b‴]

\: M × A-**infset** → M [restriction by]
[a↦b,a′↦b′,a″↦b″]\{a} = [a′↦b′,a″↦b″]

/: M × A-**infset** → M [restriction to]
[a↦b,a′↦b′,a″↦b″]/{a′,a″} = [a′↦b′,a″↦b″]

=,≠: M × M → **Bool**

°: (A $\xrightarrow[m]{}$ B) × (B $\xrightarrow[m]{}$ C) → (A $\xrightarrow[m]{}$ C) [composition]
[a↦b,a′↦b′] ° [b↦c,b′↦c′,b″↦c″] = [a↦c,a′↦c′]

Map Operation Explication

- $m(a)$: Application gives the element that a maps to in the map m.
- **dom**: Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.
- †: Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some "pairings" of the right operand map.
- ∪: Merge. When applied to two operand maps, it gives a merge of these maps.

- \: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements that are not in the right operand set.
- /: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.
- =: The equal operator expresses that the two operand maps are identical.
- ≠: The nonequal operator expresses that the two operand maps are *not* identical.
- °: Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1, to the range elements of the right operand map, m_2, such that if a is in the definition set of m_1 and maps into b, and if b is in the definition set of m_2 and maps into c, then a, in the composition, maps into c.

Map Operation "Redefinitions"

The map operations can also be defined as follows:

```
value
  rng m ≡ { m(a) | a:A • a ∈ dom m }

  m1 † m2 ≡
    [ a↦b | a:A,b:B •
      a ∈ dom m1 \ dom m2 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=m2(a) ]

  m1 ∪ m2 ≡ [ a↦b | a:A,b:B •
              a ∈ dom m1 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=m2(a) ]


  m \ s ≡ [ a↦m(a) | a:A • a ∈ dom m \ s ]
  m / s ≡ [ a↦m(a) | a:A • a ∈ dom m ∩ s ]

  m1 = m2 ≡
    dom m1 = dom m2 ∧ ∀ a:A • a ∈ dom m1 ⇒ m1(a) = m2(a)
  m1 ≠ m2 ≡ ~(m1 = m2)

  m°n ≡
    [ a↦c | a:A,c:C • a ∈ dom m ∧ c = n(m(a)) ]
    pre rng m ⊆ dom n
```

C.3 The RSL Predicate Calculus

C.3.1 Propositional Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values (**true** or **false** [or **chaos**]). Then:

Propositional Expressions

false, **true**
a, b, ..., c ∼a, a∧b, a∨b, a⇒b, a=b, a≠b

are propositional expressions having Boolean values. ∼, ∧, ∨, ⇒, = and ≠ are Boolean connectives (i.e., operators). They can be read as: *not*, *and*, *or*, *if then* (or *implies*), *equal* and *not equal*.

C.3.2 Simple Predicate Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values, let x, y, ..., z (or term expressions) designate non-Boolean values and let i, j, ..., k designate number values, then:

Simple Predicate Expressions

false, **true**
a, b, ..., c
∼a, a∧b, a∨b, a⇒b, a=b, a≠b
x=y, x≠y,
i<j, i≤j, i≥j, i≠j, i≥j, i>j

are simple predicate expressions.

C.3.3 Quantified Expressions

Let X, Y, ..., C be type names or type expressions, and let $\mathcal{P}(x)$, $\mathcal{Q}(y)$ and $\mathcal{R}(z)$ designate predicate expressions in which x, y and z are free. Then:

Quantified Expressions

∀ x:X • $\mathcal{P}(x)$
∃ y:Y • $\mathcal{Q}(y)$
∃ ! z:Z • $\mathcal{R}(z)$

are quantified expressions — also being predicate expressions.

They are "read" as: For all x (values in type X) the predicate $\mathcal{P}(x)$ holds; there exists (at least) one y (value in type Y) such that the predicate $\mathcal{Q}(y)$ holds; and there exists a unique z (value in type Z) such that the predicate $\mathcal{R}(z)$ holds.

C.4 λ-Calculus + Functions

C.4.1 The λ-Calculus Syntax

λ-Calculus Syntax

type /∗ A BNF Syntax: ∗/
⟨L⟩ ::= ⟨V⟩ | ⟨F⟩ | ⟨A⟩ | (⟨A⟩)
⟨V⟩ ::= /∗ variables, i.e. identifiers ∗/
⟨F⟩ ::= λ⟨V⟩ • ⟨L⟩
⟨A⟩ ::= (⟨L⟩⟨L⟩)
value /∗ Examples ∗/
⟨L⟩: e, f, a, ...
⟨V⟩: x, ...
⟨F⟩: λ x • e, ...
⟨A⟩: f a, (f a), f(a), (f)(a), ...

C.4.2 Free and Bound Variables

Free and Bound Variables Let x, y be variable names and e, f be λ-expressions.

- ⟨V⟩: Variable x is free in x.
- ⟨F⟩: x is free in $\lambda y \bullet e$ if $x \neq y$ and x is free in e.
- ⟨A⟩: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

C.4.3 Substitution

In RSL, the following rules for substitution apply:

Substitution

- **subst**([N/x]x) ≡ N;
- **subst**([N/x]a) ≡ a,

  for all variables a≠ x;
- **subst**([N/x](P Q)) ≡ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x](λx•P)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ λy• **subst**([N/x]P),

  if x≠y and y is not free in N or x is not free in P;

- **subst**([N/x](λy•P)) ≡ λz•**subst**([N/z]**subst**([z/y]P)),

  if y≠x and y is free in N and x is free in P
  (where z is not free in (N P)).

C.4.4 α-Renaming and β-Reduction

α and β Conversions

- α-renaming: λx•M

  If x, y are distinct variables then replacing x by y in λx•M results in λy•**subst**([y/x]M). We can rename the formal parameter of a λ-function expression provided that no free variables of its body M thereby become bound.

- β-reduction: (λx•M)(N)

  All free occurrences of x in M are replaced by the expression N provided that no free variables of N thereby become bound in the result. (λx•M)(N) ≡ **subst**([N/x]M)

C.4.5 Function Signatures

For sorts we may want to postulate some functions:

Sorts and Function Signatures

```
type
   A, B, ..., C
value
   ωB: A → B
   ...
   ωC: A → C
```

These functions cannot be defined. Once a domain is presented in which sort A and sorts or types B, ... and C occurs these observer functions can be demonstrated.

C.4.6 Function Definitions

Functions can be defined explicitly:

```
type                                   f(a_expr) ≡ b_expr
   A, B
value
   f: A → B [ a total function ]       g: A ~→ B [ a partial function ]
```

g(a_expr) ≡ b_expr
pre P(a_expr)

P: A → **Bool**

a_expr, b_expr are A, respectively B valued expressions of any of the kinds illustrated in earlier and later sections of this primer.
Or functions can be defined implicitly:

value
f: A→B
f(a_expr) **as** b
post P(a_expr,b)
P: A×B→**Bool**

g: A$\xrightarrow{\sim}$B
g(a_expr) **as** b
pre P′(a_expr)
post P(a_expr,b)
P′: A→**Bool**

where *b* is just an identifier.
The symbol $\xrightarrow{\sim}$ indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function.

Finally functions, f, g, ..., can be defined in terms of axioms over function identifiers, f, g, ..., and over identiers of function arguments and results.

type
A, B, C, D, ...
value
f: A → B
g: C → D
...
axiom
∀ a:A, b:B, c:C, d:D, ...
$\mathcal{P}_1$(f,a,b) ∧ ... ∧ $\mathcal{P}_m$(f,a,b)
...
$\mathcal{Q}_1$(g,c,d) ∧ ... ∧ $\mathcal{Q}_n$(g,c,d)

where $\mathcal{P}_1, \ldots, \mathcal{P}_m$ and $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$ designate suitable predicate expressions.

C.5 Other Applicative Expressions

C.5.1 Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

Let Expressions

let a = $\mathcal{E}_d$ **in** $\mathcal{E}_b$(a) **end**

is an "expanded" form of:

(λa.$\mathcal{E}_b$(a))($\mathcal{E}_d$)

C.5.2 Recursive let Expressions

Recursive **let** expressions are written as:

Recursive let Expressions

let f = λa•E(f,a) **in** B(f,a) **end**
let f = (λg•λa•E(g,a))(f) **in** B(f.a) **end**
let f = F(f) **in** E(f,a) **end where** F ≡ λg•λa•E(g,a)
let f = **YF** **in** B(f,a) **end where** **YF** = F(**YF**)

We read f = **YF** as "*f is a fix point of F*".

C.5.3 Non-deterministic let Clause

The non-deterministic **let** clause:

let a:A • $\mathcal{P}$(a) **in** $\mathcal{B}$(a) **end**

expresses the non-deterministic selection of a value a of type A which satisfies a predicate $\mathcal{P}$(a) for evaluation in the body $\mathcal{B}$(a). If no a:A • P(a) the clause evaluates to **chaos**.

C.5.4 Pattern and "Wild Card" let Expressions

Patterns and *wild cards* can be used:

Patterns

let {a} ∪ s = set **in** ... **end**
let {a,_} ∪ s = set **in** ... **end**

let (a,b,...,c) = cart **in** ... **end**
let (a,_,...,c) = cart **in** ... **end**

let ⟨a⟩^ℓ = list **in** ... **end**
let ⟨a,_,b⟩^ℓ = list **in** ... **end**

let [a↦b] ∪ m = map **in** ... **end**
let [a↦b,_] ∪ m = map **in** ... **end**

C.5.5 Conditionals

Various kinds of conditional expressions are offered by RSL:

Conditionals

if b_expr **then** c_expr **else** a_expr
end

if b_expr **then** c_expr **end** ≡ /∗ same as: ∗/
if b_expr **then** c_expr **else skip end**

if b_expr_1 **then** c_expr_1
elsif b_expr_2 **then** c_expr_2
elsif b_expr_3 **then** c_expr_3
...
elsif b_expr_n **then** c_expr_n **end**

case expr **of**
choice_pattern_1 → expr_1,
choice_pattern_2 → expr_2,
...
choice_pattern_n_or_wild_card → expr_n **end**

C.5.6 Operator/Operand Expressions

Operator/Operand Expressions

⟨Expr⟩ ::=
⟨Prefix_Op⟩ ⟨Expr⟩
| ⟨Expr⟩ ⟨Infix_Op⟩ ⟨Expr⟩
| ⟨Expr⟩ ⟨Suffix_Op⟩
| ...
⟨Prefix_Op⟩ ::=
− | ∼ | ∪ | ∩ | **card** | **len** | **inds** | **elems** | **hd** | **tl** | **dom** | **rng**
⟨Infix_Op⟩ ::=
= | ≠ | ≡ | + | − | ∗ | ↑ | / | < | ≤ | ≥ | > | ∧ | ∨ | ⇒
| ∈ | ∉ | ∪ | ∩ | \ | ⊂ | ⊆ | ⊇ | ⊃ | ⌢ | † | ∘
⟨Suffix_Op⟩ ::= !

C.6 Imperative Constructs

C.6.1 Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract, sorts and applicative constructs which, through stages of refinements, are turned into concrete types and imperative constructs.

Imperative constructs are thus inevitable in RSL.

```
  Unit
value
  stmt: Unit → Unit
  stmt()
```

- The **Unit** clause, in a sense, denotes "an underlying state"
  - which we, for simplicity, can consider as
  - a mapping from identifiers of declared variables into their values.
- Statements accept no arguments and, usually, operate on the state
  - through "reading" the value(s) of declared variables and
  - through "writing", i.e., assigning values to such declared variables.
- Statement execution thus changes the state (of declared variables).
- **Unit** → **Unit** designates a function from states to states.
- Statements, stmt, denote state-to-state changing functions.
- Affixing () as an "only" arguments to a function "means" that () is an argument of type **Unit**.

C.6.2 Variables and Assignment

Variables and Assignment

0. **variable** v:Type := expression
1. v := expr

C.6.3 Statement Sequences and skip

Sequencing is expressed using the ';' operator. **skip** is the empty statement having no value or side-effect.

2. **skip**
3. stm_1;stm_2;...;stm_n

C.6.4 Imperative Conditionals

4. **if** expr **then** stm_c **else** stm_a **end**
5. **case** e **of**: p_1→S_1(p_1),...,p_n→S_n(p_n) **end**

C.6.5 Iterative Conditionals

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

C.6.6 Iterative Sequencing

8. **for** e **in** list_expr • P(b) **do** S(b) **end**

C.7 Process Constructs

C.7.1 Process Channels

Let A, B and D stand for two types of (channel) messages and i:KIdx for channel array indexes, then:

Process Channels

```
channel
  c,c′:A
channel
  {k[i]|i:KIdx}:B
  {ch[i]i:KIdx}:B
```

declare a channel, c, and a set (an array) of channels, k[i], capable of communicating values of the designated types (A and B).

C.7.2 Process Definitions

A process definition is a function definition. The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, via which channels they wish to engage in input and output events.

Processes P and Q are to interact, and to do so "ad infinitum". Processes R and S are to interact, and to do so "once", and then yielding B, respectively D values.

```
value
  P: Unit → in c out k[i] Unit
  Q: i:KIdx → out c in k[i] Unit
  P() ≡ ... c ? ... k[i] ! e ... ; P()
  Q(i) ≡ ... k[i] ? ... c ! e ... ; Q(i)

  R: Unit → out c in k[i] B
  S: i:KIdx → out c in k[i] D
  R() ≡ ... c′ ? ... ch[i] ! e ... ; B_Val_Expr
  S(i) ≡ ... ch[i] ? ... c ! e ...; D_Val_Expr
```

C.7.3 Process Composition

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels. Let $\mathcal{P}$ and $\mathcal{Q}$ stand for process expressions, and let $\mathcal{P}_i$ stand for an indexed process expression, then:

$\mathcal{P} \parallel \mathcal{Q}$	Parallel composition
$\mathcal{P} \,[]\, \mathcal{Q}$	Nondeterministic external choice (either/or)
$\mathcal{P} \sqcap \mathcal{Q}$	Nondeterministic internal choice (either/or)
$\mathcal{P} \,\#\, \mathcal{Q}$	Interlock parallel composition
$\mathcal{O}$ { $\mathcal{P}_i$ | i:Idx }	Distributed composition, $\mathcal{O} = \parallel, [], \sqcap, \#$

express the parallel ($\parallel$) of two processes, or the nondeterministic choice between two processes: either external ([]) or internal ($\sqcap$). The interlock (#) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

C.7.4 Input/Output Events

Let c and k[i] designate channels of type A and e expression values of type A, then:

[1] c?, k[i]?	input A value
[2] c!e, k[i]!e	output A value

value

[3] P: ... → **out** c ..., P(...) ≡ ... c!e ...	offer an A value,
[4] Q: ... → **in** c ..., Q(...) ≡ ... c? ...	accept an A value
[5] S: ... → ..., S(...) = P(...)‖Q(...)	synchronise and communicate

[5] expresses the willingness of a process to engage in an event that [1,3] "reads" an input, respectively [2,4] "writes" an output. If process P reaches the c!e "program point before" process Q 'reaches program point' c? then process P "waits" on Q — and vice versa. Once both processes have reached these respective program points they "synchronise while communicating the message vale e.

The process function definitions (i.e., their bodies) express possible [output/input] events.

C.8 Simple RSL Specifications

Besides the above constructs RSL also possesses module-oriented scheme, class and object constructs. We shall not cover these here. An RSL specification is then simply a sequence of one or more clusters of zero, one or more sort and/or type definitions, zero, one or more variable declarations, zero, one or more channel declarations, zero, one or more value definitions (including functions) and zero, one or more and axioms. We can illustrate these specification components schematically:

Simple RSL Specifications

```
type
  A, B, C, D, E, F, G
  Hf = A-set, Hi = A-infset
  J = B×C×...×D
  Kf = E*, Ki = E^ω
  L = F ~m~ G
  Mt = J → Kf, Mp = J ~→ Ki
  N == alpha | beta | ... | omega
  O == μHf(as:Hf)
       | μKf(el:Kf) | ...
  P = Hf | Kf | L | ...
variable
  vhf:Hf := ⟨⟩
channel
  chf:F, chg:G, {chb[i]|i:A}:B
value
  va:A, vb:B, ..., ve:E
  f1: A → B, f2: C ~→ D
  f1(a) ≡ E_f1(a)
  f2: E → in|out chf F
  f2(e) ≡ E_f2(e)
  f3: Unit → in chf out chg Unit
  ...
axiom
  P_i(f1,va),
  P_j(f2,vb),
  ...
  P_k(f3,ve)
```

The ordering of these clauses is immaterial. Intuitively the meaning of these definitions and declarations are the following.

The **type** clause introduces a number of user-defined type names; the type names are visible anywhere in the specification; and either denote sorts or concrete types.

The **variable** clause declares some variable names; a variable name denote some value of decalred type; the variable names are visible anywhere in the specification: assigned to ('written') or values 'read'.

The **channel** clause declares some channel names; either simple channels or arrays of channels of some type; the channel names are visible anywhere in the specification.

The **value** clause bind (constant) values to value names. These value names are visible anywhere in the specification. The specification

```
type
  A
value
  a:A
```

non-deterministically binds a to a value of type A. Thuis includes, for example

```
type
  A, B
value
  f: A → B
```

which non-deterministically binds f to a function value of type A→B.

The **axiom** clause is usually expressed as several "comma (,) separated" predicates:

$$\mathcal{P}_i(\overline{A_i}, \overline{f_i}, \overline{v_i}), \mathcal{P}_j(\overline{A_j}, \overline{f_j}, \overline{v_j}), \ldots, \mathcal{P}_k(\overline{A_k}, \overline{f_k}, \overline{v_k})$$

where $(\overline{A_k}, f_\ell, \overline{v\ell})$ is an abbreviation for A_{ℓ_1}, A_{ℓ_2}, ..., A_t, f_{ℓ_1}, f_{ℓ_2}, ..., f_{ℓ_f}, v_{ℓ_1}, v_{ℓ_2}, ..., v_{ℓ_v}. The indexed sort or type names, A and the indexed function

names, d, are defined elsewhere in the specification. The index value names, v are usually names of bound 'variables' of universally or existentially quantified predicates of the indexed ("comma"-separated) $\mathcal{P}$.

Full Index

Type Index

Function Index

Channel Index

Variable Index

Symbol Index

Last page