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; printed September 6, 2010: 11:47

cover

Abstract

This divertimento – on the occasion of the 70th anniversary of Prof., Dr NNN – sketches some observations on relationships between window- and Web-based graphic user interfaces (GUI) and underlying Internet-based Linda and JavaSpaces-like spaces [19], that is, shared network-accessible repositories for arbitrary data structures. (The Preface (next) and Sect. 1.4 on page 11 presents the structure of the paper.)

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

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 [20] and JavaSpaces-like spaces [19], 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 [14, 15, 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 [14].
  - 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, [17, 32, 3, 34, 35, 28, 23]).
  - 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.
- With the current report
  - we carefully develop the notions of window and shared data spaces from basic concepts: painstakingly from
    - atomic values and types, via
    - curtain values and types, to
    - window values and types,

    in Sect. 2,
  - 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 Sect. 7,
  - the definition of a distributed system of
  - one domain process and
  - zero, one or more window processes.

  Whereas Sects. 2–6 are kep in a pure, functional style RSL [21, 22, 4], Sect. 7 extends this style with RSL's CSP [25].
- Our next tasks are to develop
  - a simple coordinated transaction processing system, in Sect. 8, and
  - a window design tool, in Sect. 9.

Contents

intro

1 Introduction

intro

1.1 Background

1.2 Intuition

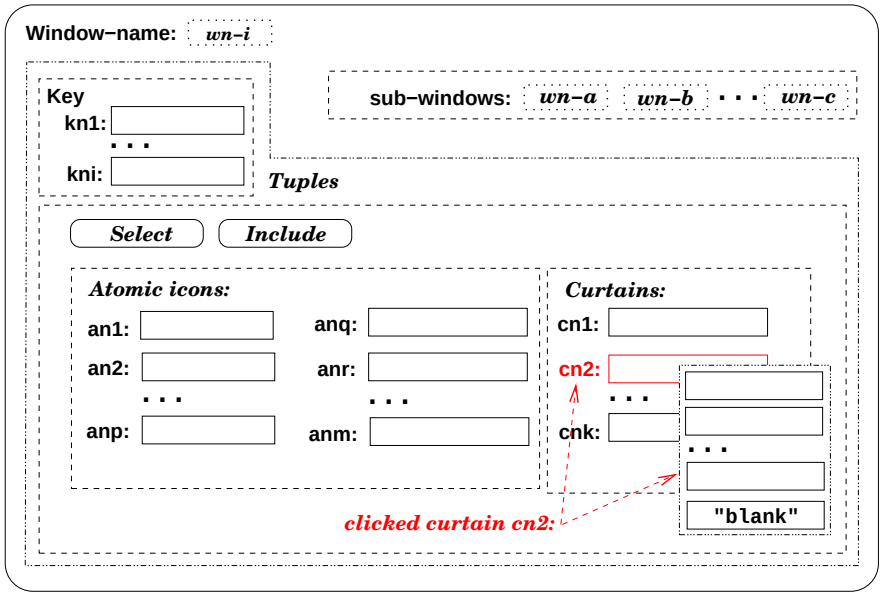

Figure 1: Schematic windows

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 6) then it represents a relation (over window values). Figure 1 on page 6 does not hint at this relation. Figure 5 on page 19 does hint at this relation.

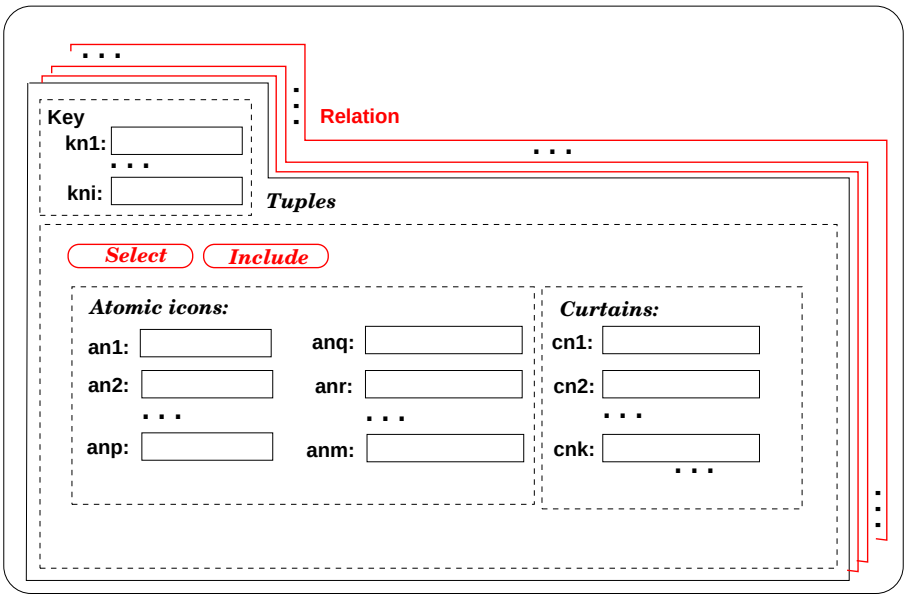

Figure 2: A window relation

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 6 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 6. In Fig. 1 on page 6 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.

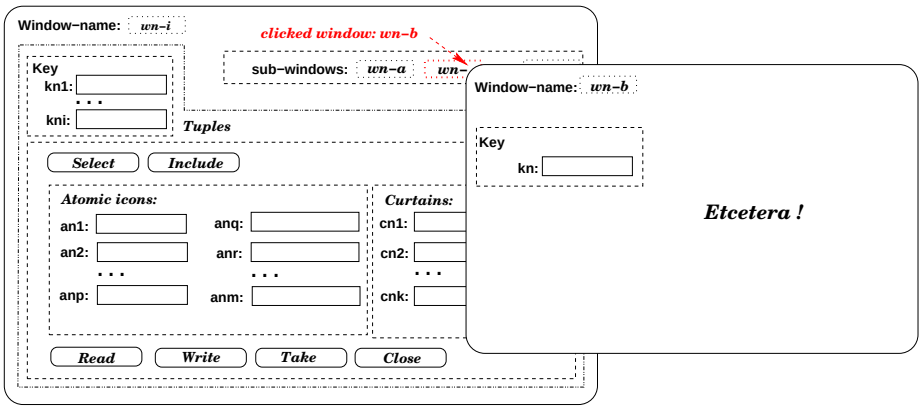

Figure 3: A sub-window

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`[1], `JavaSpaces`[2] and `XVSM`[3]-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)[4] we describe (narrate and formalise) a simple coordinated transaction processing system (commits, etc.).[5]

[1] Linda: [20]
[2] JavaSpaces: [19]
[3] XVSM: [14, 15, 29, 30]
[4] A section (Sect. 9) on a graphic user interface design system is contemplated and will appear some day!
[5] But as of September 6, 2010, this work has yet to be done.

2 Windows

windows

windows

Windows are common to the user graphic comuting 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

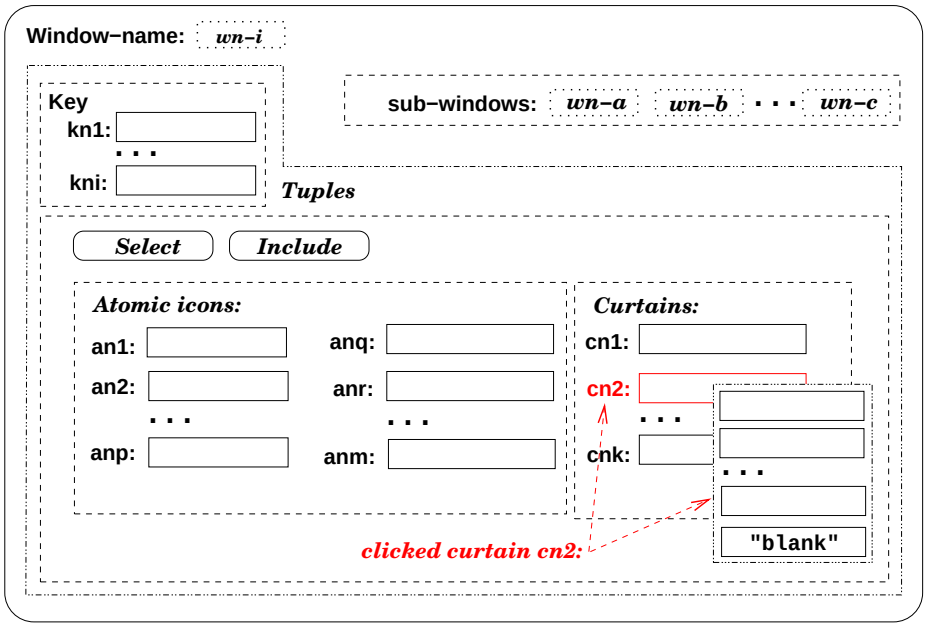

Figure 4: A Schematic Window

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".[6]

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. AType = {|″`int`″,″`rat`″,″`bool`″,″`text`″,″`nil`″|}
value
3. xtr_AType: AVAL → AType
3. xtr_AType(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_subt_ype.
   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"`.

[6] 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

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.[7]

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.

2.3.4 Curtain Supertypes

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.

[7] See Footnote 6 on page 13.

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

16. A field 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 $\overrightarrow{m}$ ATyp) ∪ (CNm $\overrightarrow{m}$ 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. KeyNn = ANm
22. KeyNms = ANm-**set**
23. Key, KVAL = ANm $\overrightarrow{m}$ AVAL
24. KTyp = ANm $\overrightarrow{m}$ ATyp
value
25. xtr_KTyp: KVAL → KTyp
25. xtr_KTyp(kv) ≡ [an↦xtr_type(kv(an))|an:ANm•an ∈ **dom** kv]

2.6.2 Relations and Relation Types

We remind the rerader of Fig. 2 repeated in Fig. 5 on the next page.

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

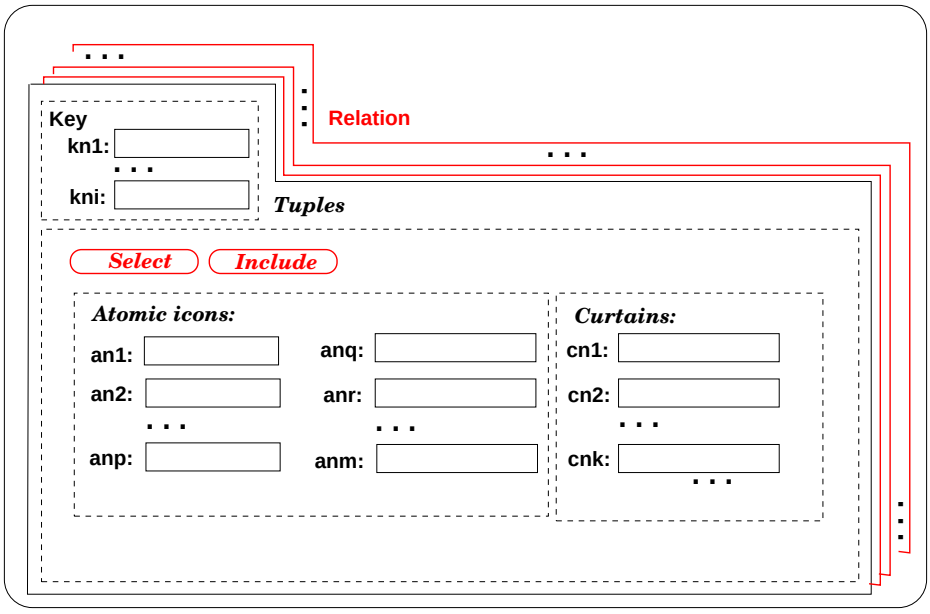

Figure 5: **Red** lines hint at a relation

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 Relation

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 reation 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,mkWType(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_FTyp(rel(kv)),wtyp)
35f. ∧ wn∉ wns

2.8 Null, Initial and Nil Windows

2.8.1 Null Windows

36. Let us recall the syntax of windows (Items 34 and 35 on the preceding page).
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**,Fields,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([ ],[ ],[ ],{}))

2.8.2 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)
```

2.8.3 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 ]
```

2.8.4 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
```

2.8.5 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 The 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)[8].

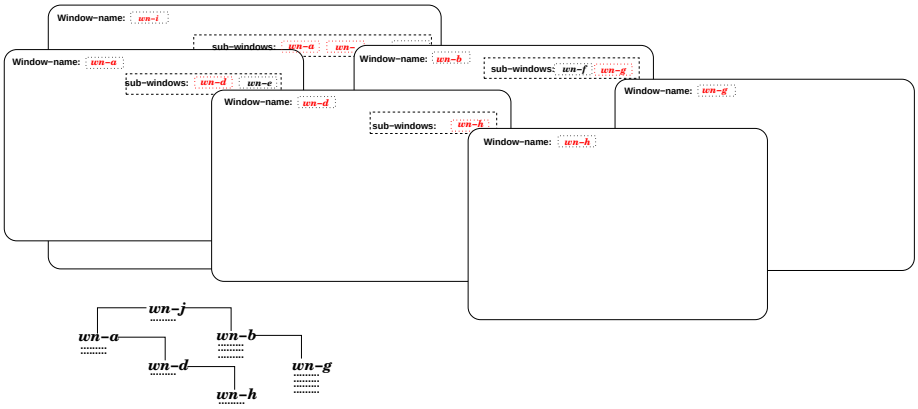

Figure 6: A cactus stack of windows

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 beween 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 Frame Syntax

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

[8] 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 $\overrightarrow{m}$ 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", or
- the `"close"` "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 wellf-rmed.

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 Paths of Window (and Domain) Frames

3.4.1 Syntax

A path is either a 0-path or it is a 1-path.

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

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

```
50.  P0 = WNm*
51.  P1 = {|p:P0•p≠⟨⟩|}
```

3.4.2 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 either window frames or to domain frames. The latter are introduced later!

```
52a. paths: (WF|DF) → P0-set
52a. paths(_,f) ≡
52b.    let ps = {⟨⟩} ∪ {⟨wn⟩|wn:WNm•wn ∈ dom f}
52c.              ∪ {(p⌢⟨wn⟩)|p:P1,wn:Wnm•p ∈ ps∧wn ∈ s_WNms(f)} in
52a.    ps end
```

s_WNms is defined below.

proof obligation: the invocation of s_WNms(wfs) is well-defined

3.4.3 Selection Functions

53. From a window (or a domain[9]) frame one can select a sub-window-frame given a possibly empty path.

Select Window Frames

```
value
53. s_Frame: (P0 × WF ~→ WF) | (P0 × DF ~→ DF)
53. s_Frame(p,(w,fs)) ≡
53.    case p of
53.       ⟨⟩ → (w,fs),
53.       ⟨wn⟩⌢p′ → if wn ∈ dom fs
53.                    then s_Frame(p′,fs(wn))
53.                    else chaos end
53.    end
```

Select Window States

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

```
value
54. s_WΣ: P0 × WF ~→ WΣ
54. s_WΣ(p,wf) ≡ let (wσ,_) = s_Frame(p,wf) in wσ end
54. pre p ∈ paths(wf)
```

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

Select Windows

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

value
55. s_W: P0 × WF $\xrightarrow{\sim}$ W
55. s_W(p,wf) ≡ **let** ((_,w,_,_),_) = s_Frame(p,wf) **in** w **end**
55. **pre** p ∈ paths(wf)

Select Window Names

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

value
56. s_WNms: P0 × WF $\xrightarrow{\sim}$ WNm-**set**
56. s_WNms(p,wf) ≡ **let** (_,wfs) = s_Frame(p,wf) **in dom** wfs **end**
56. **pre** p ∈ paths(wf)

4 The Domain Frame System

domain-frames

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

57. Domain frames map window names into a pair of windows and a set of uniquely window named domain frames[10]

58. A "bottom-most" window has the "built-in" name `"wn_bottom_DF"`.

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

4.2 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 26.
59c. ∧ ∀ wn:WNm • wn ∈ **dom** dfs ⇒ wf_DF(dfs(wn))

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

4.3 Null Domain Frames

60. A null domain frame is a pair of a null window and an empty set of domain frames.

value
37. null_w:W = (wnm,[ ],mkWV([ ],[ ],[ ],{}))
60. null_d:DF = (null_w,[ ])

4.4 Paths of Domain Frames

This section is "isomorphic" to Sect. 3.4.

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.

4.4.1 Syntax

We recall the syntax of paths from Sect. 3.4.

50. A 0-path is a sequence of one or more window names.
51. A 1-path is a non-empty 0-path.

50. P0 = WNm*
51. P1 = {|p:P0•p≠⟨⟩|}

4.4.2 Domain Frames Define Paths

Parametrically identically to Item 52a on page 27's definition of paths over window frames we can define paths over domain frames. We have therefore, already in Sect. 3.4, types the paths and *selection* functions according.

63. The window selection function (Item 55 on page 29) is slightly different.

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

domain-ops

5 Domain Frame Operations domain-ops

Commands are syntactic structures. The meaning of commands are operations.

5.1 Commands

5.1.1 Narrative

64. The are the following domain space commands: initialize domain, create domain, remove domain, create window, get window, update window, delete window.
    a) The initialize domain need not present any arguments (other than that it is an inituialize domain command).
    b) The create domain command presents a path and a window name.
    c) The remove domain 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
64. Cmd = IniDF | CreDF | RmDF | UpdWi | GetWi
64a. IniDF == ″initialize″
64b. CreDF == mkCDF(s_p:P0,s_wn:WNm)
64c. RmDF == mkRDF(s_p:P0,s_wn:WNm)
64d. DFPutW == mkDFPW(s_p:P0,s_wn:WNm,s_w:W)
64e. DFGetW == mkDFGW(s_p:P0)

5.2 Operations

All operations apply in the context of a domain.

65. The get window operation possibly yields a window but does not change the domain.

66. All other domain operations either changes the domain or a undefined.

65. eval_DFGetW: DFGetW → DF $\xrightarrow{\sim}$ W
66. int_″X″: ″X″ → DF $\xrightarrow{\sim}$ DF

where "X" is one of IniDF, CreDF, RmDF or DFPutW.

5.2.1 The Initialize Domain Frame Operation

67. The initialize domain command takes oly the command argument but yields an empty domain.

67. int_IniDF: IniDF → DF
67. int_IniDF(″`initialize`″) ≡ (init_W(),[ ])

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 DomainSpace the empty domain is to be installed.

68. The create domain frame operation is partial; it produces a changed domain.
69. To express the changed domain, from df to df′ we use the concept of all paths, ps, ps′, of respectively df and df′. **Precondition:**
    a) The path, p, of the command must be a path of the domain frame; and
    b) the window name, wn, of the command must not be of a window of the window frames of the window frame 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 domain frames of the domain frame designated by p in df with an initial window, named wn, as part of the new domain[11].
    e) Then all paths common to ps and ps′, that is, all paths of ps, designate the same domain frames in df and df′,
    f) except that the window of the selected window frame has its subwindow names part augmented with wn (all other parts are left unchanged).

68. int_CreDF: CreDF → DF ~→ DF
68. int_CreDF(mkCDF(p,wn))(df) **as** df′
69. **let** ps = paths(df), ps′ = paths(df′) **in**
69a. **pre** p ∈ paths(df)
69b. ∧ {p⌢⟨wn⟩}∉ ps

[11] How that window is initialised we do n ot specify – other than through the nondeterminism of the init_W() operation.

69c. **post** $ps' = ps \cup \{p^\frown\langle wn\rangle\} \wedge \{p^\frown\langle wn\rangle\} \notin ps$
69e. $\wedge \forall\ p'{:}P \bullet p' \in ps \bullet s\_DF(p',df) = s\_DF(p',df')$
69d. $\wedge$ **let** $((wn',wt',wv'),dfb) = s\_DF(pp,df), ((wn'',wt'',wv''),dfa) = s\_DF(pp,df')$ **in**
69d. $wn' = wn'' \wedge wt' = wt'' \wedge dfa = dfb \cup [\, wn \mapsto (init\_W(),[\,])\,]$
69f. $\wedge$ **let** $mkWV(kn',tpl',rel',ws') = wv', mkWV(kn'',tpl'',rel'',ws'') = wv''$ **in**
69f. $kn' = kn'' \wedge tpl' = tpl'' \wedge rel' = rel'' \wedge ws'' = ws' \cup \{wn\}$
68. **end end end**

5.2.3 The Remove Domain Frame Operation

70. The remove operation is partial.

Precondition:

a) The window path $\{p^\frown\langle wn\rangle\}$ of the command must be a window path of df.

Postcondition:

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

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, $\{p^\frown\langle wn\rangle\}$, in ps′

ii. removed as a result of removing the domain frame designated by $\{p^\frown\langle wn\rangle\}$ in df.

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

e) except that the window of the selected window frame has its sub-window names part reduced by wn (all other parts are left unchanged).

70. int_RmDF: RmDF → DF ~→ DF
70. int_RmDF(mkRDF(p,wn))(df) **as** df′
70. **let** $ps = paths(df), ps' = paths(df')$ **in**
70a. **pre** $p^\frown\langle wn\rangle \in ps$
70b. **post** $ps' = ps \setminus rm\_paths(ps)(p^\frown\langle wn\rangle)$
70d. $\wedge \forall\ p'{:}P \bullet p' \in ps' \bullet s\_D(p',df) = s\_D(p',df')$
70e. $\wedge$ **let** $((wn',wt',wv'),dfb) = s\_DF(p,df), ((wn'',wt'',wv''),dfa) = s\_DF(p,df')$ **in**
70e. $wn' = wn'' \wedge wt' = wt'' \wedge dfa = dfb \cup [\, wn \mapsto (init\_W(),[\,])\,]$
70e. $\wedge$ **let** $mkWV(kn,tpl,rel,ws) = wv, mkWV(kn',tpl',rel',ws') = wv'$ **in**
70e. $kn = kn' \wedge tpl = tpl' \wedge rel = rel' \wedge ws' = ws \setminus \{wn\}$
70. **end end end**

70(c)i. is_prefix: P1 × P1 → **Bool**
70(c)i. $is\_prefix(p,p') \equiv$ **len** $p <$ **len** $p' \wedge \forall\ i{:}\mathbf{Nat} \bullet i \in$ **inds** $p \Rightarrow p(i) = p'(i)$

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

Identity of (int_RmD(mkRD(p,wn))∘int_CreD(mkCD(p,wn)))(df)

71. Given an domain, d, any path p in d and any window name wn not at p in d.
72. 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
71. ∀ df:DF, p:P0, wn:WNm • p ∈ paths(df) ∧ wn ∉ s_WNms(p,df)
72. ⇒ int_RmD(mkRD(p,wn))(int_CreD(mkCD(p,wn))(df)) = df

5.2.4 The Put Window Operation

73. The int_DFPutW(mkDFPW(p,wna,wa:(wnb,wta,wva)))(df) 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 d and d′.
    b) The window path, p, of the command must be a window path of d;
    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 d;
    d) the argument window must be well-formed; and
    e) the argument window type must conform to the type of the window in df 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 domains 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 thery were in the domain frame version of the (before) window and wrt. the argument relation.
    j) The sub-domain frame window names are unchanged.

73. int_DFPutW: DFPutW → DF ~→ DF
73. int_DFPutW(mkDFPW(p,wna,wa:(wnb,wta,wva)))(df) **as** df′
73a. **let** ps = paths(df), ps′ = paths(df′),
73. mkWV(kn,tpla,rela,wsa) = wva **in**
73b. **pre** p ∈ ps ∧ p⌢⟨wna⟩ ∈ ps
73c. ∧ wna = wnb ∧ wna ∈ s_WNms(p,df)
73d. ∧ wf_W(wa)
73e. ∧ **let** ((wnc,vtd,_),_) = s_DF(p⌢⟨wna⟩,d) **in**
73e. wna = wnc ∧ wta = wtd **end**
73e. **post** ps′ = ps
73f. ∧ ∀ p′:P•p′ ∈ ps′\{p⌢⟨wn⟩} •s_DF(p′,d)=s_DF(p′,d′)
73. ∧ **let** ((wn,wt,wv),df″) = s_DF(p⌢⟨wna⟩,df),
73. ((wn′,wt′,wv′),df‴) = s_DF(p⌢⟨wn⟩,df′) **in**
73g. wna=wn=wn′ ∧ wt=wt′ ∧ df″=df‴
73. ∧ **let** mkWV(kn,tpl,rel,ws) = wv,
73. mkWV(kn′,tpl′,rel′,ws′) = wv′ **in**
73h. tpl′ = tpla
73i. ∧ rel′ = rel † [fn↦rela(fn)|fn:FNm•fn ∈ kn]
73j. ∧ ws′ = ws = wsa
73. **end end end**

5.2.5 The Get Window Operation

74. The get window does not change the domain – otherwise the operation result seems obviois !.
75. The path of the command must be in the domain of the operation.

74. eval_DFGetW: DFGetWi → DF ~→ W
74. eval_DFGetW(mkDFGW(p,wn)(df)) ≡ s_W(p,df,wn)
75. **pre** p ∈ paths(df)

5.3 Discussion

5.3.1 Mon. 30 Aug., 2010

Some "loose" remarks – expressed at a time (Mon. 30 Aug., 2010) when I had not yet had time to caregully study my own narratives and formalisations:

- 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 everal 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

76. 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,domain frames) pair. The window name designate the window. That window is to be the result of the open window operation.

    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.

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

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

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

    f) The select command "retrieves" a tuple from the (not displayed) relation ('which always "underlies" a window, whether in the window frame or in the domain frame sub-sustems). The tuple, in the relation, which is selected from the relation is the one which, in a sense, 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
76. Cmd = OpnW|CloW|WFPutW|ClkW|WrW|SelTpl|IncTpl
76a. OpnW == mkOpnW(s_p:P0,s_wn:WNm,s_kv:KeyVAL)
76b. CloW == mkCloW(s_p:P0,s_wn:WNm)
76c. WFPutW == mkWFPW(s_p:P0,s_wn:WNm)

76d. ClkW == mkClkW(s_p:P0,s_wn:WNm,s_f:FPos)
76d. FPos = ANm | CNm | CNmIx
76e. WrW == mkWrW(s_p:P0,s_wn:WNm,v:FVAL)
76f. SelTpl == mkSel(s_p:P0,w_wn:WNm,s_kv:KeyVAL)
76g. IncTpl == mkInc(s_p:P0,w_wn:WNm)

6.2 Operations

77. Window domain commands are interpreted in the context of of a window frame and a domain frame and potentially changes both.

78. Window commands are interpreted in the context of of a graphic user interface and potentially changes it.

To express the changed wf (into wf′) resulting from the operations we express some predicates over the sets of paths, ps, ps′, of wf and wf′, 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_OpnW(mkOpnW(p,wn,kv))(wf)(df), is intended to open a "fresh" window frame, (window,domain frame), at *location* p under the name of wn and with an initially empty domain frame. That window, window, is obtained from the domain frame, df, at location p^⟨wn⟩. The intention is now for the user to *click* (see Sect. 6.2.3 on page 42) on that window, to *write* (see Sect. 6.2.4 on page 43) into fields of that window (which thereby is updated – to become window′), to, most likely *put* (see Sect. 6.2.5 on page 44) that window back into the (global) domain frame (df), and, finally to *close* (see Sect. 6.2.2 on page 41) the window′.

79. The open window operation is partial.

    Precondition:

    a) The path p^⟨wn⟩ must be a path of the window frame wf.
       By a theorem, to be stated later, the that path is then also a path pf the domain frame df.

    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
79.  int_OpnW: OpnW → WF → DF ~→ WF
79.  int_OpnW(mkOpnW(p,wn,kv))(wf)(df) as wf′
79.    let ps=paths(wf),ps′=paths(wf′) in
79a.    pre p ∈ ps ∧ p^⟨wn⟩ ∉ ps
79b.    post ps′ = ps ∪ {p^⟨wn⟩}
79c.      ∧ let w = eval_DFGetW(mkGW(p^⟨wn⟩,df)) in
79d.          wf′ = insertW(p^⟨wn⟩,kv,w)(wf) end
79.    end
```

80. 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 (79a.).
    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 – as also expressed in the postcondition of the calling function (79b.).
    c) Paths common to wf and wf′, that is, in ps, designate the same window frames in both wf and wf′.
    d) The window at location p^⟨wn⟩ in wf′ –
    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 wf′ is empty.

```
80.  insert_W: P0 × KVAL × W → WF ~→ WF
80.  insertW(p^⟨wn⟩,kv,w)(wf) as wf′
80.    let ps=paths(wf),ps′=paths(wf′) in
80a.    pre  p^⟨wn⟩ ∉ ps
80b.    post ps′ = ps ∪ {p^⟨wn⟩}
80c.      ∧ ∀ p:P • p ∈ ps ⇒ s_WF(p,wf)=s_WF(p,wf′)
80.       ∧ let (wn′,wtyp,mkWV(kn,tpl,rel)) = w in
80f.        let w′ = (wn,wtyp,mkWV(kn,sel_flds(kv,rel,wtyp),rel)) in
80e.        wn=wn′ ∧ dom kv = kn
80d.        w′ = s_W(p^⟨wn⟩,wf′)
80g.      ∧ s_DF(p^⟨wn⟩,wf′) = [ ]
80d.    end end end
```

6.2.2 Close Window Frame

We saw, in Sect. 5.2.3, Items 70–70e (Page 34), how the *remove domain frame* operation can delete an entire 'cactus stack' of domain frames[12]. 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.

81. The close window operation, int_CloW(mkCloW(p,wn))(wf), is partial.

    Precondition:

    a) The path p⌢⟨wn⟩ must be a path of the window frame df (and hence also, by a theorem, of the domain frame df).

    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 70(c)i on page 34.) which are a proper prefix of paths, {p⌢⟨wn⟩}, in ps′

       (Cf. Item 70(c)ii on page 34.) removed as a result of removing the window frame designated by {p⌢⟨wn⟩} in wf.

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

The reader is encouraged to compare the *remove domain frame* and the *close window frame* operation: Sect. 5.2.3 on page 34 versus this section.

```
value
81.  int_CloW: CloW → WF ~→ WF
81.  int_CloW(mkCloW(p,wn))(wf) as wf′
81.    let ps = paths(wf), ps′ = paths(wf′) in
81a.   pre p⌢⟨wn⟩ ∈ ps
81b.   post ps′ = ps \ rm_paths(ps)(p⌢⟨wn⟩) ∧ {p⌢⟨wn⟩} ∉ ps
81c.        ∧ ∀ p′:P•p′ ∈ ps′•s_WF(p′,wf)=s_WF(p′,wf′)
81.    end
```

[12] 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_ClkW(mkClkW(p,wn,fp))(wf)(df) 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.

82. The click on window operation is partial.

Precondition:

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

b) The field position, fp, of the argument must be a proper position within the window selected by $\mathsf{p}^\frown\langle\mathsf{wn}\rangle$.

Postcondition:

c) The paths of the window frame are unchanged.

d) All paths common to ps (except path $\mathsf{p}^\frown\langle\mathsf{wn}\rangle$) and ps′, that is, all paths of ps′ (except path $\mathsf{p}^\frown\langle\mathsf{wn}\rangle$) designate the same window frames in wf and wf′.

e) Path $\mathsf{p}^\frown\langle\mathsf{wn}\rangle$ designates window mkWV(wnb,wtyp,flds,frel,wns) in wf and window mkWV(wna,wtyp′,flds′,frel′,wns′) in wf′ where, pairwise, wnb (before, in wf) and wna (after, in textsfwf), wtyp and wtyp′, flds and flds′, flre 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

82. int_ClkW: ClkW → WF $\xrightarrow{\sim}$ WF
82. int_ClkW(mkClkW(p,wn,fp))(wf)(df) **as** wf′
82. **let** ps = paths(wf), ps′ = paths(wf′),
82a. **pre** {p⌢⟨wn⟩} ∈ ps
82. ∧ **let** (wσ:(w,fv,c),wfpwn) = s_Frame(p⌢⟨wn⟩,wf),
82. **let** mkWV(wnb,wtyp,flds,frel,wns) = w **in**
82b. appropriate_FPos(fp,flds,wns)
82c. **post** ps′ = ps \ rm_paths(ps)(p⌢⟨wn⟩) ∧ p⌢⟨wn⟩∉ ps
82d. ∀ p′:P•p′ ∈ ps′⇒s_Frame(p′,wf)=s_Frame(p′,wf′)
82. ∧ **let** (wσ′:(w′,fv′,c′),wfpwn′) = s_Frame(p⌢⟨wn⟩,wf′) **in**
82. **let** mkWV(wna′,wtyp′,flds′,frel′,wns′) = w′ **in**

82e. wnb=wna∧wtyp=wtyp′∧flds=flds′∧frel=frel′∧wns=wns′
82f. ∧ appropriate_FPos(fp,flds′,wns′)
82g. ∧ c′=fp
82h. ∧ fv′ = select_value(flds′)(fp)
82. **end end end end end**

82b.,82f. appropriate_FPos: FPos × Fields × WNm-**set** → **Bool**
82b.,82f. appropriate_FPos(fp,flds,wns) ≡
82b.,82f. **case** fp **of**
82b.,82f. mkCNmIx(cnm,x)
82b.,82f. → appropriate_FPos(mkCNm(cnm),flds,wns)
82b.,82f. ∧ x ∈ **inds** flds(mkCNm(cnm)),
82b.,82f. _ → fp ∈ **dom** flds ∪ wns
82b.,82f. **end**

82d. select_value: Fields → FPos $\xrightarrow{\sim}$ FV
82d. select_value(flds)(fp) ≡
82d. **case** fp **of**
82d. mkCNmIx(cnm,x) → (flds(mkCNm(cnm)))(x),
82d. _ → flds(fp)
82d. **end**

6.2.4 Write Window

83. The write window operation, int_WrW(mkWrW(p,wn))(wf)(df), is partial.

Precondition:

a) The path p⌢⟨wn⟩ must be a path of the window frame wf.
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
e)
f)
g)
h)
i)

value
83. int_WrW: WrW → WF $\xrightarrow{\sim}$ WF
83. int_WrW(mkWrW(p,wn,fv))(wf) **as** wf′

83. **let** ps = paths(wf), ps′ = paths(wf′),
83a. **pre** p⌢⟨wn⟩ ∈ ps
83. ∧ **let** (wσ:(w,fv,c),wfpwn) = s_Frame(p⌢⟨wn⟩,wf),
83. **let** mkWV(wnb,wtyp,flds,frel,wns) = w **in**
83b. appropriate_FPos(fp,fields,{})
83c. ∧ sub_type(xtr_typ(fv),wtyp(c)) **[check!]**
83d. **post** ps′ = ps
83e. ∧ ∀ p′:P•p′ ∈ ps′\{p⌢⟨wn⟩}⇒s_Frame(p′,wf)=s_Frame(p′,wf′)
83. ∧ **let** (wσ′:(w′,fv′,c′),wfpwn′) = s_Frame(p⌢⟨wn⟩,wf′) **in**
83. **let** mkWV(wna′,wtyp′,flds′,frel′,wns′) = w′ **in**
83f. wnb=wna ∧ wtyp=wtyp′ ∧ frel=frel′ ∧ wns=wns′
83g. ∧ flds′ = update_field(flds,c,fv)
83h. ∧ c′=fp
83i. ∧ fv′ = select_value(flds′)(fp) **assert:** fv′ = fv
83. **end end end end end**

83g. update_field: Fields × Cursor × FVAL → Fields
83g. update_field(flds,c,fv) **as** flds′
83g. **pre**: appropriate_FPos(c,flds,{})
83g. **post** ∀ fn:FNm•fn ∈ **dom** flds ⇒
83g. **case** (c,fp) **of**
83g. (mkCNmIx(cnm,x),mkCNmIx(cnm,x)) → select_value(flds)(c)=fv,
83g. _ → flds′(c)=fv
83g. **end**

6.2.5 Put Window

84. The put window operation is partial.

Precondition:

a) The path p⌢⟨wn⟩ must be a path of the window frame df (and hence also, by a theorem, of the domain frame df).

Postcondition:

b) The window frame is unchanged.

c) Let w be the window

d) being put in the domain frame sub-system which thereby "changes" state to df″.

e) The (only) effect of this window frame operation is that the domain frame has changed to df′ = df″.

value
84. int_WFPutW: WFPutW → WF → DF → (DF × WF)

84. int_WFPutW(mkWPW(p,wn))(wf)(df) **as** (df′,wf′)
84. **let** ps = paths(df) **in**
84a. **pre** {p⌢⟨wn⟩} ∈ ps
84b. **post** wf′ = wf
84c. ∧ **let** w = s_W(p⌢⟨wn⟩,wf) **in**
84d. **let** df″ = int_DFPutW(mkDPW(p⌢⟨wn⟩,wn,w))(df) **in**
84e. df′ = df″
84. **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_PutW(mkPW(p,wn))(wf')(df')
    - originated from that position, specifically:
      let (df',wf') = int_OpnW(mkOpnW(p,wn,kv))(wf)(df) **in** ... **end**
    - That is:
      p=p′, wn=wn′
  - **A Specific:**
    - **let** (df',wf') = int_OpnW(mkOpnW(p,wn,kv))(wf)(df) **in**
    - **let** (df'',wf'') = int_OpnW(mkOpnW(p,wn,kv))(wf')(df') **in**
    - **let** (df''',wf''') = int_PutW(mkPW(p,wn))(wf'')(df'') **in**
    - **let** (df'''',wf'''') = int_CloW(mkCloW(p,wn))(wf''')(df''') **in**
    - wf''=wf'∧df''=df'∧wf'''=wf''∧df'''= df''∧wf''''=wf∧df''''= df'''
    - **end end end end**

6.2.6 Select Tuple

85. The mkSel(s_p:P0,w_wn:WNm,s_kv:KeyVAL) operation is partial.

Precondition:

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

Postcondition:

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

c) For all paths other than to the window effect the 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
85. int_SelTpl: SelTpl → WF → WF
85. int_SelTpl(mkSel(p,wn))(wf) **as** wf′
85. **let** ps = paths(wf), ps′ = paths(wf′), pn=p⌢⟨wn⟩ **in**
85a. **pre** pn ∈ ps
85b. **post** ps = ps′
85c. ∧ ∀ p:P•p ∈ ps\{pn} ⇒ s_WF(p,wf)=w_WF(p.wf′)
85. ∧ **let** w = s_W(pn,wf), w′ = s_W(pn,wf) **in**
85. **let** (wn′,rt,mkWV(kn,tpl,rel,wns))=w,
85. (wn″,rt′,mkWV(kn′,tpl′,rel′,wns′))=w′
85. kv = [fn↦tpl(fn)|fn:FNm•fn ∈ kn],**in**
85d. wn′=wn″=wn ∧ rt=rt′ ∧ kn=kn′ ∧ rel=rel′ ∧ wns=wns′
85e. ∧ tpl′ = **if** kv ∈ **dom** rv
85e. **then** kv ∪ rv(kv)
85e. **else** kv ∪ nil_FVAL(rt**dom** kn)
85. **end end end end**

6.2.7 Include Tuple

86. The mkInc(s_p:P0,w_wn:WNm) operation is partial.

Precondition:

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

Postcondition:

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

c) For all paths other than to the window effect 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
86. int_IncTpl: IncTpl → WF → WF
86. int_IncTpl(mkInc(p,wn))(wf) **as** wf′

86. **let** ps = paths(wf), ps′ = paths(wf′), pn=p⌢⟨wn⟩ **in**
86a. **pre** pn ∈ ps
86b. **post** ps = ps′
86c. ∧ ∀ p:P•p ∈ ps\{pn} ⇒ s_WF(p,wf)=w_WF(p.wf′)
86. ∧ **let** w = s_W(pn,wf), w′ = s_W(pn,wf) **in**
86. **let** (wn′,rt,mkWV(kn,tpl,rel,wns))=w,
86. (wn″,rt′,mkWV(kn′,tpl′,rel′,wns′))=w′,
86. kv′ = [fn↦tpl(fn)|fn ∈ kn] **in**
86d. wn′=wn″=wn ∧ rt=rt′ ∧kn=kn′ ∧ rel=rel′ ∧ wns=wns′
86e. ∧ rel′ = rel † [kv′↦tpl**dom** kv′]
86. **end end end**

6.3 Discussion

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

transactions

7 A Simple Transaction System transactions

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

7.1 An Analysis

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

7.1.1 Domain Frame Elaboration (Function) Signatures

67. int_IniDF: IniDF → DF
68. int_CreDF: CreDF → DF $\xrightarrow{\sim}$ DF
70. int_RmDF: RmDF → DF $\xrightarrow{\sim}$ DF
73. int_DFPutW: DFPutW → DF $\xrightarrow{\sim}$ DF
74. eval_DFGetW: DFGW → DF $\xrightarrow{\sim}$ W

7.1.2 Window Frame Elaboration Function Signatures

79. int_OpnW: OpnW → WF → DF $\xrightarrow{\sim}$ WF
81. int_CloW: CloW → WF $\xrightarrow{\sim}$ WF
82. int_ClkW: ClkW → WF $\xrightarrow{\sim}$ WF
83. int_WrW: WrW → WF $\xrightarrow{\sim}$ WF
84. int_WFPutW: WFPutW → WF → DF → (DF × WF)
85. int_SelTpl: SelTpl → WF → WF
86. int_IncTpl: IncTpl → WF → WF

7.1.3 Window Frame to Domain Frame Invocations

Of the above referenced operations only two involve

79d. eval_DFGetW(mkGetW(p⌢⟨wn⟩,df)) yielding w:W
84d. int_DFPutW(mkDFPW(p⌢⟨wn⟩,wn,w))(df) transferring w:W

7.1.4 Changes

- The domain process shall have a local variable, df, replacing the need for an interpretation function argument (df).

- Each window process shall have a local variable, wf, replacing the need for an interpretation function argument (wf).
- The int_OpnW(mkOpnW(p,wn,kv))(wf)(df) function is interpreted in the window frame processes.
  - The function invokes the eval_DFGetW(mkGW(p,df)) function which is evaluated in the domain process.
  - That process yields, i.e., communicates, a window w.
  - Hence two communications shall be offered:
    * A request from a window frame process to the domain process to perform eval_DFGetW(mkGW(p,df));
    * followed by the domain process "returning" (hopefully) the window w (else that an "error" has occurred).
- The int_WFPutW(mkWPW(p,wn))(wf)(df) function is interpreted in the window frame processes.
  - The function invokes the int_DFPutW(mkDPW(p,wn,w))(df) function which is interpreted in the domain process,
  - which yields a side-effect on that domain process.
  - Hence two communications shall be offered:
    * The request from a window frame process to the domain process to perform the int_DFPutW(mkDPW(p,wn,w))(df).
    * The reply from the domain process that the request was honoured, that is, "ok", or that something went wrong, that is, "error".

7.2 The System

87. Channels are indexed by an otherwise undefined quantity.
88. 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.
89. A domain_frame process ...
90. A window_frame process ...

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

88. system: WIdx-**set** → **Unit**

88. system(cxs) ≡ domain_frame(...) ‖ (‖{window_frame(wi,...)|wi:WIdx•wi ∈ wis})
89. domain_frame: DF → **in**,**out** ... **Unit**
90. window_frame: i:WIdx × WF → **in**,**out** ... **Unit**

7.2.1 Channels

87. Channels are indexed by an otherwise undefined quantity.

91. Channels express willingness to accept and offer messages of, for the moment, further unspecified nature.

92. The is an array of channels.

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

7.2.2 The System Process

93. There is an initial association, wfs:WFS, window frame indexes into initial, not necessarily identical window frames.

94. The domain frame signaure can now be completed.

type
93. WFS = WIdx $\overrightarrow{m}$ WF
value
93. wfs:WFS
94. system: WFS → **in**,**out** {ch[wi]|wi:WIdx•wi ∈ wis} **Unit**
94. system(wfs) ≡ domain_frame() ‖ (‖{window_frame(wi,wfs(wi))|wi:WIdx•wi ∈ wis})

7.2.3 The Domain Frame Process

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

95. 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 inpur/output), stands for "willingness to accept either of two inputs from any window process.

96. The domain frame process takes no argument and
97. "cycles forever".
98. The elaboration function parameter, df, is provided by the **c**ontents of an assignable domain frame valued variable, vdf – which is initialised to an initial domain frame (cf. 67 on page 33).
99. In each step (well, cycle) of the domain process a non-deterministic internal choice is made (is taken) as to which kind of operation to perform.
100. If the choice is to re-initialise the domain frame then that is done.
101. If the choice is
    a) to create a domain frame, or
    b) to remove a domain frame, or
    c) to put a window (into the domain frame), or
    d) to obtain (get) a window from the domain frame,

    then a corresponding command is internally non-deterministically chosen such that thise command satisfies the **pre** condition for the corresponding interpretation function — and the domain frame variable is set to the result of the corresponding operation.
102. If the choice is to be willing to accept a request from a window frame process
    a) then the domain process external non-deterministically ([]) choose to accept from any window frame process, i:WIdx,
    b) either a get window or a put window request;
    c) if the request satisfies the precondition of the corresponding domain frame elaboration function,
    d) then that operation is performed and its result communicated back to the requesting window process,
    e) else an `"error"` message is returned.

```
type
95.  Cmd == init | crea | rmdf | putw | getw | inut
value
96.  domain_frame: Unit → in,out {ch[ wi ]|wi:WIdx•wi ∈ wis}  Unit
96.  domain_frame() ≡
98.    variable vdf:DF := int_iniDF();
97.    while true do
99.      let cmd = init⌈⌉crea⌈⌉remv⌈⌉put⌈⌉get⌈⌉inut in
96.      case cmd of
```

```
100.          init → vdf := elab_DFinit(),
101a.         crea → vdf := elab_DFCre(c vdf),
101b.         rmdf → vdf := elab_DFRmv(c vdf),
101c.         putw → vdf := elab_DFPutW(c vdf),
101d.         getw → vdf := elab_DFGetW(c vdf),
102.          inut → interaction()
95.     end end end

100.  elab_DFinit: Unit → DF
100.  elab_DFinit() ≡ int_iniDF()

101a.  elab_DFCre: Unit → DF → DF
101a.  elab_DFCre(mkCDF(p,wn))(df) ≡
101a.     if ∃ cdf:CreDF • pre:int_CreDF(cdf)(df)
101a.        then let cdf:CreDF • pre:int_CreDF(cdf)(df) in
101a.                int_CreDF(mkCDF(p,wn))(df) end
101a.        else df end

101b.  elab_DFRmv: DF → DF
101b.  elab_DFRmv(df) ≡
101b.     if ∃ rdf:RmDF • pre:int_RmDF(rdf)(df) in
101b.        then let rdf:RmDF • pre:int_RmDF(rdf)(df) in
101b.                int_RmDF(cdf)(df) end
101b.        else df end

101c.  elab_DFPutW: DF → DF
101c.  elab_DFPutW(df) ≡
101c.     if ∃ put:DFPutW • pre:int_DFPutW(put)(df) in
101c.        then let put:DFPutW • pre:int_DFPutW(put)(df) in
101c.                int_DFPutW(put)(df) end
101c.        else df end

101d.  elab_DFGetDF: DF → DF
101d.  elab_DFGetDF(df) ≡
101d.     if ∃ get:DFGetW • pre:eval_DFGetW(get)(df) in
101d.        then let get:DFGetW • pre:eval_DFGetW(get)(df) in
101d.                eval_DFGetW(get)(df) end
101d.        else df end

102.  interaction: DF → DF
102.  interaction(df) ≡
102.     variable ldf:DF
102a.     []{let req = ch[i]? in
102a.        case req of
102b.           mkDFGW(p) →
102c.              if pre:eval_DFGetW(mkDFGW(p))(df)
102d.                 then ch[i]!eval_DFGetW(mkDFGW(p))(df)
102e.                 else ch[i]!"error" end ;
102d.              ldf:=df
```

```
102b.          put →
102c.            if pre:int_DFPutW(put)(c vdf)
102d.               then ldf := int_DFPutW(put)(df);ch[i]!"ok"
102e.               else ch[i]!"error" ; ldf := df end
102a.        end | i:WIdx•i ∈ wis end}; ldf
```

7.2.4 The Window Frame Processes

The window frame process, at its own volition, "alternates" between honouring either of the five window seven frame operations – two of which requires interaction with the domain process.

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

104. The window frame process takes no argument and

105. "cycles forever".

106. The elaboration function parameter, wf, is provided by the **c**ontents of an assignable window frame valued variable, vwf – which is initialised to a "pre-set" initial window frame (cf. 93 on page 50).

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

108. If the choice is 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 window process communicates a put window command request (mkGW(p⌢⟨wn⟩)) to the domain process and awaits a response.
    b) If the domain process could not find the window at the communicated path position then an "error" result is received and **chaos** ensues.
    c) If the domain process does find the (a) window, w, at the communicated path position then it is returend and that window, w, is *inserted* at the chosen path position – by means of the auxiliary function: int_InsertW(p⌢⟨wn⟩,kv,w)(wf).

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

    a) then a corresponding command is internally non-deterministically chosen such that this command satisfies the **pre** condition for the put window operation;
    b) the window process communicates a mkGW(p) 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 nothin – the window frame state is unchanged.

110. 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 internally non-deterministically chosen such that thise command satisfies the **pre** condition for the corresponding interpretation function —

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

```
type
103.  Cmd = opn | put | clk | wri | sel | inc | clo
value
104.  window_frame: i:ChIdx × WF → in,out ch[i] Unit
104.  window_frame(i,wf) ≡
106.    variable vwf:WF := wf;
105.    while true do the
103.      let cmd = opn⌈clo⌈put⌈clk⌈wri⌈sel⌈inc in
107.      case cmd of
108.        opn → vwf := elab_WFOpnW(wf),
109.        put → vwf := elab_WFPutW(wf),
110b.         clk → vwf := elab_WFClkW(wf),
110c.         wri → vwf := elab_WFWriW(wf),
110d.         sel → vwf := elab_WFSelW(wf),
110e.         inc → vwf := elab_WFIncW(wf),
110a.         clo → vwf := elab_WFCloW(wf)
103.    end end end
108.  elab_WFOpnW: WF → WF
108.  elab_WFOpnW(wf) ≡
108.    if ∃ ow:OpenW • pre:int_OpnW(ow)(wf) in
108.      then let mkOpnW(p,wn,kv):OpenW•pre:int_OpnW(mkOpnW(p,wn,kv))(wf)
108a.           let w = ch[i]!mkGW(p^⟨wn⟩) ; ch[i]? in
108b.           if w=″error″ then chaos else skip end;
108c.           insert_W(p^⟨wn⟩,kv,w)(wf) end end
109.      else wf end
```

```
109.  elab_WFPutW: WF → WF
109.  elab_WFPutW(wf) ≡
109.    if ∃ mkWPW(p,wn):WFPutW•pre:int_WFPutW(mkWPW(p,wn))(wf)
109.      then let mkWPW(p,wn):WFPutW •
109.                 pre:int_WFPutW(mkWPW(p,wn))(wf) in
109a.           let w = s_W(p^⟨wn⟩,wf) in
109b.           let result = ch[i]!mkDPW(p^⟨wn⟩,wn,w) ; ch[i]? in
109c.           if result = "error" then chaos else skip end
109.          end end end
109.      else wf end
110b. elab_WFClkW: WF → WF
110b. elab_WFClkW(wf) ≡
110b.   if ∃ cl:ClkW • pre:int_ClkW(cl)(wf)
110b.     then let cl:ClkW • pre:int_ClkW(cl)(wf) in
110b.          int_ ClkW(cl)(wf) end
110b.     else wf end
110c. elab_WFWriW: WF → WF
110c. elab_WFWriW(wf) ≡
110c.   if ∃ wr:WrW • pre:int_WrW(wr)(wf)
110c.     then let wr:WrW • pre:int_WrW(wr)(wf) in
110c.          int_WrW(wr)(wf) end
110c.     else wf end
110d. elab_WFSelW: WF → WF
110d. elab_WFSelW(wf) ≡
110d.   if ∃ sl:Sel • pre:int_Sel(sl)(c vwf)
110d.     then let sl:Sel • pre:int_Sel(sl)(c vwf) in
110d.          int_ClkW(sl)(c vwf) end
110d.     else wf end
110e. elab_WFIncW: WF → WF
110e. elab_WFIncW(wf) ≡
110e.   if ∃ ic:Inc • pre:int_Inc(ic)(wf)
110e.     then let ic:Inc • pre:int_Inc(ic)(wf) in
110e.          int_ClkW(ic)(wf) end
110e.     else wf end
110a. elab_WFCloW: WF → WF
110a. elab_WFCloW(wf) ≡
110a.   if ∃ cl:CloW • pre:int_CloW(clo)(wf)
110a.     then let cl:CloW • pre:int_CloW(clo)(wf) in
110a.          int_CloW(clo)(wf) end
110a.     else wf end
```

7.3 Discussion

There are (quite) a number of problems with the definition of the system process as composed from one domain_frame and n window_frame processes.

8 A Coordinated Transactions Processing System

tp-system

tp-system

8.1 Co-ordination Principle

8.2 The Coordination Primitives

8.3 The Model

8.3.1 The Revised Domain Frame Process

8.3.2 The Revised Window Frame Processes

8.4 Discussion

9 A Window Design Tool gui-design

gui-design

9.1 Design Principles

9.2 Graphics

9.3 Syntax

9.4 Commands and Operations

9.4.1 Commands

9.4.2 Operations

9.5 Discussion

10 Conclusion

con

con

10.1 Discussion

10.1.1 What Have We Achieved

10.1.2 What Have We Not Achieved

10.1.3 What Should We Do Next

10.2 Acknowledgements

The window formalisation dates back some 20 years and is recorded in [6, Example 19.28, Pages 438–442]. The connection between window states and WindowSpaces arose as a result of trying to understand [15, 29, 30, 14, XVSM].

10.3 Bibliographical Notes

Besides using as precise a subset of a national language, as here English, as possible, and, in enumerated expressions and statements, "pairing" such narrative elements with corresponding enumerated clauses of a formal specification language. For our formalisations ee have used the RAISE formal Specification Language, RSL, [21, 22, 5]. But any of the model-oriented approaches and languages offered by Alloy [27], Event B [1], VDM [12, 13, 18] and Z [36], 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 [33], CSP: Communicating Sequential Processes [25], MSC: Message Sequence Charts [26], Statecharts [24], and some temporal logic, for example either DC: Duration Calculus [37] 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* [16] 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 [11] 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. See [7, 9].

[5] D. Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling; Vol. 2: Specification of Systems and Languages; ol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

[6] D. Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. See [8, 10].

[7] D. Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Qinghua University Press, 2008.

[8] D. Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Qinghua University Press, 2008.

[9] D. Bjørner. ***Chinese:*** *Software Engineering, Vol. 1: Abstraction and Modelling*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.

[10] D. Bjørner. ***Chinese:*** *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.

[11] 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.).

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

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

[14] 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.

[15] 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.

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

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

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

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

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

[25] 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).

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

[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, *8^{th} 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] L. Marshall. *A Formal Description Method for User Interfaces*. PhD thesis, University of Manchester, Oct. 1986.

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

[34] 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.

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

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

[37] 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.

Index

Last page