What are Documents?
A Research Note
Version 1. Incomplete Draft

Dines Bjørner
Fredsvej 11, DK-2840 Holte, Danmark
E-Mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

Abstract. We domain analyse and suggest a description of a domain of documents. We emphasize that the model is one of several possible. Common to these models is that we model “all” we can say about documents – irrespective of whether it can also be “implemented”! The model(s) are not requirements prescriptions – but we can develop such from our domain description.

You may find that the model is overly detailed with respect to a number of “operations” and properties of documents. We find that these operations must be part of the very basis of a document domain in order to cope with documents such as they occur in, for example, public government, see Appendix sect. A, or in urban planning, see Appendix Sect. B.


Contents

1 Introduction
2 A Document Systems Description
   2.1 A System for Managing, Archiving and Handling Documents 4
   2.2 Principal Endurants 4
   2.3 Unique Identifiers 4
   2.4 Documents: A First View 5
      2.4.1 Document Identifiers 5
      2.4.2 Document Descriptors 5
      2.4.3 Document Annotations 5
      2.4.4 Document Contents: Text/Graphics 5
      2.4.5 Document Histories 6
      2.4.6 A Summary of Document Attributes 6
   2.5 Behaviours: An Informal, First View 7
   2.6 Channels, A First View 7
   2.7 An Informal Graphical System Rendition 8
   2.8 Behaviour Signatures 9
   2.9 Time 9
      2.9.1 Time and Time Intervals: Types and Functions 9
      2.9.2 A Time Behaviour and a Time Channel 10
      2.9.3 An Informal RSL Construct 10
   2.10 Behaviour “States” 10

© Dines Bjørner, 2017
Correspondence and offprint requests to: Dines Bjørner, Fredsvej 11, DK-2840 Holte, Denmark
2.11 Inter-Behaviour Messages
- 2.11.1 Management Messages with Respect to the Archive
- 2.11.2 Management Messages with Respect to Handlers
- 2.11.3 Document Access Rights
- 2.11.4 Archive Messages with Respect to Management
- 2.11.5 Archive Message with Respect to Documents
- 2.11.6 Handler Messages with Respect to Documents
- 2.11.7 Handler Messages with Respect to Management
- 2.11.8 A Summary of Behaviour Interactions

2.12 A General Discussion of Handler and Document Interactions

2.13 Channels: A Final View

2.14 An Informal Summary of Behaviours
- 2.14.1 The Create Behaviour: Left Fig. 3 on Page 15
- 2.14.2 The Edit Behaviour: Right Fig. 3 on Page 15
- 2.14.3 The Read Behaviour: Left Fig. 4 on Page 15
- 2.14.4 The Copy Behaviour: Right Fig. 4 on Page 15
- 2.14.5 The Grant Behaviour: Left Fig. 5 on Page 16
- 2.14.6 The Shred Behaviour: Right Fig. 5 on Page 16

2.15 The Behaviour Actions
- 2.15.1 Management Behaviour
  - Management Create Behaviour: Left Fig. 3 on Page 15
  - Management Copy Behaviour: Right Fig. 4 on Page 15
  - Management Grant Behaviour: Left Fig. 5 on Page 16
  - Management Shred Behaviour: Right Fig. 5 on Page 16
- 2.15.2 Archive Behaviour
  - The Archive Create Behaviour: Left Fig. 3 on Page 15
  - The Archive Copy Behaviour: Right Fig. 4 on Page 15
  - The Archive Shred Behaviour: Right Fig. 5 on Page 16
- 2.15.3 Handler Behaviours
  - The Handler Create Behaviour: Left Fig. 3 on Page 15
  - The Handler Edit Behaviour: Right Fig. 3 on Page 15
  - The Handler Read Behaviour: Left Fig. 4 on Page 15
  - The Handler Copy Behaviour: Right Fig. 4 on Page 15
  - The Handler Grant Behaviour: Left Fig. 5 on Page 16
- 2.15.4 Document Behaviours
  - The Document Edit Behaviour: Right Fig. 3 on Page 15
  - The Document Read Behaviour: Left Fig. 4 on Page 15
  - The Document Shred Behaviour: Right Fig. 5 on Page 16

2.16 Conclusion

3 Bibliography
3.1 Bibliographical Notes
3.2 Domain Modeling Experiments
3.3 References

A Documents in Public Governance
B Documents in Urban Planning
C RSL: The RAISE Specification Language – A Primer
  C.1 Type Expressions
    - C.1.1 Atomic Types
    - C.1.2 Composite Types
      - Concrete Composite Types
      - Sorts and Observer Functions
  C.2 Type Definitions
    - C.2.1 Concrete Types
    - C.2.2 Subtypes
    - C.2.3 Sorts — Abstract Types
  C.3 The RSL Predicate Calculus
  C.4 Propositional Expressions
    - C.4.1 Simple Predicate Expressions
    - C.4.2 Quantified Expressions
  C.5 Concrete RSL Types: Values and Operations
    - C.5.1 Arithmetic
    - C.5.2 Set Expressions
      - Set Enumerations
      - Set Comprehension
    - C.5.3 Cartesian Expressions
    - C.5.4 List Expressions
1. Introduction

We analyse a notion of documents. Documents such as they occur in daily life. What can we say about documents – regardless of whether we can actually provide compelling evidence for what we say! That is: we model documents, not as electronic entities — which they are becoming, more-and-more, but as if they were manifest entities. When we, for example, say that “this document was recently edited by such-and-such and the changes of that editing with respect to the text before is such-and-such”, then we can, of course, always claim so, even if it may be difficult or even impossible to verify the claim. It is a fact, although maybe not demonstrably so, that there was a version of any document before an edit of that document. It is a fact that some handler did the editing. It is a fact that the editing took place at (or in) exactly such-and-such a time (interval), etc. We model such facts.
2. A Document Systems Description

This research note unravels its analysis & description in stages.

2.1. A System for Managing, Archiving and Handling Documents

The title of this section: A System for Managing, Archiving and Handling Documents immediately reveals the major concepts: That we are dealing with a system that manages, archives and handles documents. So what do we mean by managing, archiving and handling documents? We give an ultra short survey. The survey relies on your prior knowledge of what you think documents are!

Management decides to direct handlers to work on documents. Management first directs the document archive to create documents. The document archive creates documents, as requested by management, and informs management of the unique document identifiers (by means of which handlers can handle these documents). Management then grants its designated handler(s) access rights to documents, these access rights enable handlers to edit, read and copy documents. The handlers' editing and reading of documents is accomplished by the handlers "working directly" with the documents (i.e., synchronising and communicating with document behaviours). The handlers' copying of documents is accomplished by the handlers requesting management, in collaboration with the archive behaviour, to do so.

2.2. Principal Endurants

By an endurant we shall understand “an entity that can be observed or conceived and described as a "complete thing" at no matter which given snapshot of time.” Were we to "freeze" time we would still be able to observe the entire endurant. This characterisation of what we mean by an ‘endurant’ is from [26, Manifest Domains: Analysis & Description].

We begin by identifying the principal endurants.

1 From document handling systems one can observe aggregates of handlers and documents. We shall refer to ‘aggregates of handlers’ by M, for management, and to ‘aggregates of documents’ by A, for archive.
2 From aggregates of handlers (i.e., M) we can observe sets of handlers (i.e., H).
3 From aggregates of documents (i.e., A) we can observe sets of documents (i.e., D).

2.3. Unique Identifiers

The notion of unique identifiers is treated, at length, in [26, Manifest Domains: Analysis & Description].

We use the logogram & between two terms, A & B, when we mean to express one meaning.

3 How these decisions come about is not shown in this research note – as it has nothing to do with the essence of document handling, but, perhaps, with ‘management’.

4 [26, Manifest Domains: Analysis & Description] explains how ‘parts’ are the discrete endurants with which we associate the full complement of properties: unique identifiers, mereology and attributes.
As reasoned in [26, Manifest Domains: Analysis & Description], the unique identifiers of endurant parts are indeed unique: No two parts, whether composite, as are the aggregates, or atomic, as are handlers and documents, can have the same unique identifiers.

2.4. Documents: A First View

A document is a written, drawn, presented, or memorialized representation of thought. The word originates from the Latin documentum, which denotes a “teaching” or “lesson”.9 We shall, for this research note, take a document in its written and/or drawn form. In this section we shall survey the concept a documents.

2.4.1. Document Identifiers

Documents have unique identifiers. If two or more documents have the same document identifier then they are the same, one (and not two or more) document(s).

2.4.2. Document Descriptors

With documents we associate document descriptors. We do not here stipulate what document descriptors are other than saying that when a document is created it is provided with a descriptor and this descriptor “remains” with the document and never changes value. In other words, it is a static attribute.10 We do, however, include, in document descriptors, that the document they describe was initially based on a set of zero, one or more documents – identified by their unique identifiers.

2.4.3. Document Annotations

With documents we also associate document annotations. By a document annotation we mean a programmable attribute, that is, an attribute which can be ‘augmented’ by document handlers. We think of document annotations as “incremental”, that is, as “adding” notes “on top of” previous notes. Thus we shall model document annotations as a repository: notes are added, i.e., annotations are augmented, previous notes are not edited, and no notes are deleted. We suggest that notes be time-stamped. The notes (of annotations) may be such which record handlers work on documents. Examples could be: “15 June 2017: 10:43 am: This is version V.”, “This document was released on 15 June 2017: 10:43 am.”, “15 June 2017: 10:43 am: Section X.Y.Z of version III was deleted.”, “15 June 2017: 10:43 am: References to documents doc_i and doc_j are inserted on Pages p and q, respectively.” and “15 June 2017: 10:43 am: Final release.”

2.4.4. Document Contents: Text/Graphics

The main idea of a document, to us, is the written (i.e., text) and/or drawn (i.e., graphics) contents. We do not characterise any format for this contents. We may wish to insert, in the contents, references to locations in the contents of other documents. But, for now, we shall not go into such details. The main operations on documents, to us, are concerned with: their creation, editing, reading, copying and shredding. The editing and reading operations are mainly concerned with document annotations and text/graphics.

---

5 We shall not, in this research note, make use of the (one and only) management identifier.
6 We shall not, in this research note, make use of the (one and only) archive identifier.
7 Cf. Footnote 5: hence we shall not be using the uid_MI observer.
8 Cf. Footnote 6: hence we shall not be using the uid_AI observer.
10 You may think of a document descriptor as giving the document a title; perhaps one or more authors; perhaps a physical address (of, for example, these authors); an initial date; as expressing whether the document is a research, or a technical report, or other; who is issuing the document (a public institution, a private firm, an individual citizen, or other); etc.
2.4.5. **Document Histories**

So documents are created, edited, read, copied and shredded. These operations are initiated by the management (create), by the archive (create), and by handlers (edit, read, copy), and at specific times.

2.4.6. **A Summary of Document Attributes**

6 As separate attributes of documents we have document descriptors, document annotations, document contents and document histories.
7 Document annotations are lists of document notes.
8 Document histories are lists of time-stamped document operation designators.
9 A document operation designator is either a create, or an edit, or a read, or a copy, or a shred designator.
10 A create designator identifies
   a a handler and a time (at which the create request first arose), and presents
   b elements for constructing a document descriptor, one which
   i besides some further undefined information
   ii refers to a set of documents (i.e., embeds reference to their unique identifiers),
   c a (first) document note, and
   d an empty document contents.
11 An edit designator identifies a handler, a time, and specifies a pair of edit/undo functions.
12 A read designator identifies a handler.
13 A copy designator identifies a handler, a time, the document to be copied (by its unique identifier, and a document note to be inserted in both the master and the copy document.
14 A shred designator identifies a handler.
15 An edit function takes a triple of a document annotation, a document note and document contents and yields a pair of a document annotation and a document contents.
16 An undo function takes a pair of a document note and document contents and yields a triple of a document annotation, a document note and a document contents.
17 Proper pairs of (edit, undo) functions satisfy some inverse relation.

There is, of course, no need, in any document history, to identify the identifier of that document.

```plaintext
type
  6 DD, DA, DC, DH
value
  6 attr_DD: D → DD
  6 attr_DA: D → DA
  6 attr_DC: D → DC
  6 attr_DH: D → DH

axiom
  10bii  "empty_DC" ∈ DC

```
Axiom
17 \( \forall \text{mkEdit}(\_e,u) : \text{Edit} \) •
17 \( \forall (da,dn,dc) : (DA \times DN \times DC) \) •
17 \( u(e(da,dn,dc)) = (da,dn,dc) \)

2.5. Behaviours: An Informal, First View

In [26, Manifest Domains: Analysis & Description] we show that we can associate behaviours with parts, where parts are such discrete endurants for which we choose to model all its observable properties: unique identifiers, mereology and attributes, and where behaviours are sequences of actions, events and behaviours.

- The overall document handler system behaviour can be expressed in terms of the parallel composition of the behaviours
  18 of the system core behaviour,
  19 of the handler aggregate (the management) behaviour
  20 and the document aggregate (the archive) behaviour,

  with the (distributed) parallel composition of
  21 all the behaviours of handlers and,
  22 the (distributed) parallel composition of
  23 at any one time, zero, one or more behaviours of documents.

- To express the latter

  23 we need introduce two “global” values: an indefinite set of handler identifiers and an indefinite set of document identifiers.

value
23 his:HI-set, dis:DI-set

18 sys(...)
19 || mgtm(...)
20 || arch(...)
21 || ||{hdlr,\_i\:HI\:i\:in\:his}
22 || ||{docu_i(dd)(da,dc,dh)\:i\:DI\:i\:in\:dis}

For now we leave undefined the arguments, (...) etc., of these behaviours. The arguments of the document behaviour, (dd)(da,dc,dh), are the static, respectively the three programmable (i.e., dynamic) attributes: document descriptor, document annotation, document contents and document history. The above expressions, Items 19–22, do not define anything, they can be said to be “snapshots” of a “behaviour state”. Initially there are no document behaviours, docu_i(dd)(da,dc,dh), Item 22. Document behaviours are “started” by the archive behaviour (on behalf of the management and the handler behaviours). Other than mentioning the system (core) behaviour we shall not model that behaviour further.

2.6. Channels, A First View

Channels are means for behaviours to synchronise and communicate values (such as unique identifiers, mereologies and attributes).
The management behaviour, \textit{mgtm}, need to (synchronise and) communicate with the archive behaviour, \textit{arch}, in order, for the management behaviour, to request the archive behaviour

- to \textbf{create} (ab initio or due to \textbf{copying})
- or \textbf{shred} document behaviours, \textit{docu}_j,

and for the archive behaviour

- to inform the management behaviour of the identity of the document(behaviour)s that it has created.

channel

24 \textit{mgtm}_{\textit{arch}}_{\textit{ch}}:\textit{MA}

The management behaviour, \textit{mgtm}, need to (synchronise and) communicate with all handler behaviours, \textit{hdlr}_i and they, in turn, to (synchronised) communicate with the handler management behaviour, \textit{mgtm}. The management behaviour need to do so in order

- to inform a handler behaviour that it is granted access rights to a specific document, subsequently these access rights may be modified, including revoked.

channel

25 \{\textit{mgtm}_{\textit{hdlr}}_{\textit{ch}}[i]:\textit{MH}|i:HI,i \in \textit{his}\}

The document archive behaviour, \textit{arch}, need (synchronise and) communicate with all document behaviours, \textit{docu}_j and they, in turn, to (synchronise and) communicate with the archive behaviour, \textit{arch}.

channel

26 \{\textit{arch}_{\textit{docu}}_{\textit{ch}}[j]:\textit{AD}|h:DI\}

Handler behaviours, \textit{hdlr}_i, need (synchronise and) communicate with all the document behaviours, \textit{docu}_j, with which it has operational allowance to do so\textsuperscript{11}, and document behaviours, \textit{docu}_j, need (synchronise and) communicate with potentially all handler behaviours, \textit{hdlr}_i, namely those handler behaviours, \textit{hdlr}_i with which they have (“earlier” synchronised and) communicated.

channel

27 \{\textit{hdlr}_{\textit{docu}}_{\textit{ch}}[i,j]:\textit{HD}|i:HI,j:DI|i \in \textit{his} \land j \in \textit{dis}\}

At present we leave undefined the type of messages that are communicated.

\textbf{type}

28 \textit{MA, MH, AD, HD}

\textbf{2.7. An Informal Graphical System Rendition}

Figure 1 on the next page is an informal rendition of the “state” of a number of behaviours: a single management behaviour, a single archive behaviour, a fixed number, \textit{n}_h, of one or more handler behaviours, and a variable, initially zero number of document behaviours, with a maximum of these being \textit{n}_d. The figure also indicates, again rather informally, the channels between these behaviours: one channel between the management and the archive behaviours; \textit{n}_h channels (\textit{n}_h is, again, informally indicated) between the management behaviour and the \textit{n}_h handler behaviours; \textit{n}_d channels (\textit{n}_d is, again, informally indicated) between the archive behaviour and the \textit{n}_d document behaviours; and \textit{n}_h \times \textit{n}_d channels (\textit{n}_h \times \textit{n}_d is, again, informally indicated) between the \textit{n}_h handler behaviours and the \textit{n}_d document behaviours.

\textsuperscript{11} The notion of operational allowance will be explained below.
2.8. **Behaviour Signatures**

29 The `mgtm` behaviour (synchronises and) communicates with the `arch` behaviour and with all of the handler behaviours, `hdlr_i`.

30 The `arch` behaviour (synchronises and) communicates with the `mgtm` behaviour and with all of the document behaviours, `docu_j`.

31 The signature of the generic handler behaviours, `hdlr_i` expresses that they [occasionally] receive “orders” from management, and otherwise [regularly] interacts with document behaviours.

32 The signature of the generic document behaviours, `docu_j` expresses that they [occasionally] receive “orders” from the `arch` behaviour and that they [regularly] interacts with handler behaviours.

\[
\text{value}
\]

29 `mgtm`: ... → \text{in,out} \text{mgtm}\_arch\_ch, \{\text{mgtm\_hdlr\_ch}[i]|i:Hi\#i \in \text{his}\} \text{ Unit}

30 `arch`: ... → \text{in,out} \text{mgtm}\_arch\_ch, \{\text{arch\_docu\_ch}[j]|j:Di\#j \in \text{dis}\} \text{ Unit}

31 `hdlr_i`: ... → \text{in} \text{mgtm\_hdlr\_ch}[i], \text{in,out} \{\text{hdlr\_docu\_ch}[i,j]|j:Di\#j \in \text{dis}\} \text{ Unit}

32 `docu_j`: ... → \text{in} \text{mgtm\_arch\_ch}, \text{in,out} \{\text{hdlr\_docu\_ch}[i,j]|i:Hi\#i \in \text{his}\} \text{ Unit}

2.9. **Time**

2.9.1. **Time and Time Intervals: Types and Functions**

33 We postulate a notion of time, one that covers both a calendar date (from before Christ up till now and beyond). But we do not specify any concrete type (i.e., format such as: YY:MM:DD, HH:MM:SS).

34 And we postulate a notion of (signed) time interval — between two times (say: ±YY:MM:DD:HH:MM:SS).

35 Then we postulate some operations on time: Adding a time interval to a time obtaining a time; subtracting one time from another time obtaining a time interval, multiplying a time interval with a natural number; etc.

36 And we postulate some relations between times and between time intervals.

\[
\text{type}
\]

33 `TIME`

34 `TIME\_INTERVAL`

\[
\text{value}
\]

35 `add`: `TIME\_INTERVAL × TIME` → `TIME`

36 `sub`: `TIME × TIME` → `TIME\_INTERVAL`

37 `mpy`: `TIME\_INTERVAL × Nat` → `TIME\_INTERVAL`

38 `<,≤,=,≠,≥,>`: `((TIME×TIME)×(TIME\_INTERVAL×TIME\_INTERVAL))` → `Bool`
2.9.2. A Time Behaviour and a Time Channel

37 We postulate an “ongoing” time behaviour: it either keeps being a time behaviour with unchanged time, \( t \), or – internally non-deterministically – chooses being a time behaviour with a time interval incremented time, \( t+\Delta t \), or – internally non-deterministically – chooses to [first] offer its time on a [global] channel, \( \text{time}_{\text{ch}} \), then resumes being a time behaviour with unchanged time., \( t \).

38 The time interval increment, \( \Delta t \), is likewise internally non-deterministically chosen. We would assume that the increment is “infinitesimally small”, but there is no need to specify so.

39 We also postulate a channel, \( \text{time}_{\text{ch}} \), on which the time behaviour offers time values to whoever so requests.

\[
\text{value} \\
37 \text{time}: \text{TIME} \rightarrow \text{time}_{\text{ch}} \text{ TIME Unit} \\
37 \text{time}(t) \equiv (\text{time}(t) \mid \text{time}(t+\Delta t) \mid \text{time}_{\text{ch}}(t) ; \text{time}(t)) \\
38 \Delta t: \text{TIME INTERVAL} ... \\
\text{channel} \\
39 \text{time}_{\text{ch}}: \text{TIME}
\]

2.9.3. An Informal RSL Construct

The formal-looking specifications of this report appear in the style of the RAISE [37] Specification Language, RSL [36]. We shall be making use of an informal language construct:

• \text{wait} \( \Delta t \).

\text{wait} is a keyword; \( \Delta t \) designates a time interval. A typical use of the wait construct is:

• ... ptA ; \text{wait} \( \Delta t \); ptB ; ...

If at specification text point ptA we may assert that time is \( t \), then at specification text point ptB we can assert that time is \( t+\Delta t \).

2.10. Behaviour “States”

We recall that the endurant parts, Management, Archive, Handlers, and Documents, have properties in the form of unique identifiers, mereologies and attributes. We shall not, in this research note, deal with possible mereologies of these endurants. In this section we shall discuss the endurant attributes of mgmt (management), arch (archive), hdlrs (handlers), and docs (documents). Together the values of these properties, notably the attributes, constitute states – and, since we associate behaviours with these endurants, we can refer to these states also a behaviour states. Some attributes are static, i.e., their value never changes. Other attributes are dynamic. Document handling systems are rather conceptual, i.e., abstract in nature. The dynamic attributes, therefore, in this modeling “exercise”, are constrained to just the programmable attributes. Programmable attributes are those whose value is set by “their” behaviour. For a behaviour \( \beta \) we shall show the static attributes as one set of parameters and the programmable attributes as another set of parameters.

\[
\text{value} \quad \beta: \text{Static} \rightarrow \text{Program} \rightarrow ... \text{ Unit}
\]

40 For the management endurant/behaviour we focus on one programmable attribute. The management behaviour needs keep track of all the handlers it is charged with, and for each of these which zero, one or more documents they have been granted access to (cf. Sect. 2.11.3 on Page 12). Initially that management directory lists a number of handlers, by their identifiers, but with no granted documents.

\[\text{We refer to Sect. 3.4 of [26], and in particular its subsection 3.4.4.}\]
41 For the archive behaviour we similarly focus on one programmable attribute. The archive behaviour needs keep track of all the documents it has used (i.e., created), those that are available (and not yet used), and of those it has shredded. Initially all these three archive directory sets are empty.

42 For the handler behaviour we similarly focus on one programmable attribute. The handler behaviour needs keep track of all the documents it has been charged with and its access rights to these.

43 Document attributes we mentioned above, cf. Items 6–9.

| type |  
| MDIR = HI \rightarrow (DI \rightarrow \text{ANm-set}) |  
| ADIR = \text{avail:DI-set} \times \text{used:DI-set} \times \text{gone:DI-set} |  
| HDIR = DI \rightarrow \text{ANm-set} |  
| SDATR = DD, PDATR = DA \times DC \times DH |  

axiom

∀ (avail,used,gone):ADIR • avail \cap used = {} \land gone \subseteq used

We can now “complete” the behaviour signatures. We omit, for now, static attributes.

| value |  
| mgmt: MDIR \rightarrow \text{in,out mgmt_arch, ch, \{mgmt_hdlr_ch[i]|i:Hl\i \in his\}} Unit |  
| arch: ADIR \rightarrow \text{in,out mgmt_arch, ch, \{arch_docu_ch[j]|j:DI\j \in dis\}} Unit |  
| hdr_i: HDIR \rightarrow \text{in mgmt_hdlr_ch[i], in,out \{hdr_docu_ch[i,j]|j:DI\j \in dis\}} Unit |  
| docu_j: SDATR \rightarrow PDATR \rightarrow \text{in mgmt_arch, ch, in,out \{hdr_docu_ch[i,j]|i:Hl\i \in his\}} Unit |  

2.11. Inter-Behaviour Messages

Documents are not “fixed, innate” entities. They embody a “history”, they have a “past”. Somehow or other they “carry a trace of all the ”things” that have happened/occurred to them. And, to us, these things are the manipulations that management, via the archive and handlers perform on documents.

2.11.1. Management Messages with Respect to the Archive

44 Management create documents. It does so by requesting the archive behaviour to allocate a document identifier and initialize the document “state” and start a document behaviour, with initial information, cf. Item 10 on Page 6:

a the identity of the initial handler of the document to be created,
b the time at which the request is being made,
c a document descriptor which embodies a (finite) set of zero or more (used) document identifiers (dis),
d a document annotation note dn, and
e an initial, i.e., “empty” contents, "empty_DC".

| type |  
| Crea :: (HI \times TIME) \times (DI-set \times Info) \times DN \times \{|"empty_DC"|\} [cf. formula Item 10, Page 6] |  

45 The management behaviour passes on to the archive behaviour, requests that it accepts from handlers behaviours, for the copying of document:

45 Copy :: DI \times HI \times TIME \times DN [cf. Item 55 on Page 13]

46 Management schreds documents by informing the archive behaviour to do so.

| type |  
| Shred :: TIME \times DI |
2.11.2. Management Messages with Respect to Handlers

Upon receiving, from the archive behaviour, the “feedback” the identifier of the created document (behaviour):

\[
\text{type}
\]
\[
\text{CreateReply :: NewDocID(di:DI)}
\]

the management behaviour decides to grant access rights, \(acrs:ACRS\)\(^{13}\), to a document handler, \(hi:HI\).

\[
\text{type}
\]
\[
\text{Gran :: HI \times TIME \times DI \times ACRS}
\]

2.11.3. Document Access Rights

Implicit in the above is a notion of document access rights.

By document access rights we mean a set of action names. By an action name we mean such tokens that indicate either of the document handler operations indicated above.

\[
\text{type}
\]
\[
\text{ACRS = ANm-set}
\]
\[
\text{ANm = \{"edit","read","copy"\}}
\]

2.11.4. Archive Messages with Respect to Management

To create a document management provides the archive with some initial information. The archive behaviour selects a document identifier that has not been used before.

The archive behaviour informs the management behaviour of the identifier of the created document.

\[
\text{type}
\]
\[
\text{NewDocID :: DI}
\]

2.11.5. Archive Message with Respect to Documents

To shred a document the archive behaviour must access the designated document in order to stop it. No “message”, other than a symbolic “stop”, need be communicated to the document behaviour.

\[
\text{type}
\]
\[
\text{Shred :: \{"stop"\}}
\]

2.11.6. Handler Messages with Respect to Documents

Handlers, generically referred to by \(hldr_i\), may perform the following operations on documents: edit, read and copy. (Management, via the archive behaviour, creates and shreds documents.)

To perform an edit action handler \(hldr_i\) must provide the following:

- the document identity – in the form of a \((i:HI,j:DI)\) channel \(hldr_{docu}\) index value,
- the handler identity, \(i\),
- the time of the edit request,
- and a pair of functions: one which performs the editing and one which un-does it!

\(^{13}\) For the concept of access rights see Sect. 2.11.3.
2.11.7. Handler Messages with Respect to Management

55 To perform a **copy** action, a handler, $\text{hdlr}_i$, must provide the following information to the management behaviour, $\text{mgmt}$:

- the document identity,
- the handler identity – in the form of an hi:HI channel $\text{hdlr}_i \cdot \text{docu}_i \cdot \text{ch}$ index value,
- the time of the copy request, and
- a document note (to be affixed both the master and the copy documents).

55 **Copy** :: DI $\times$ HI $\times$ TIME $\times$ DN [cf. Item 45 on Page 11]

How the handler, the management, the archive and the “named other” handlers then enact the copying, etc., will be outlined later.

2.11.8. A Summary of Behaviour Interactions

Figure 2 summarises the sources, out, resp. !, and the targets, in, resp. ?, of the messages covered in the previous sections.

2.12. A General Discussion of Handler and Document Interactions

We think of documents being manifest. Either a document is in paper form, or it is in electronic form. In paper form we think of a document as being in only one – and exactly one – physical location. In electronic form a document is also in only one – and exactly one – physical location. No two handlers can access the same document at the same time or in overlapping time intervals. If your conventional thinking makes you think that two or more handlers can, for example, read the same document “at the same time”, then, in
fact, they are reading either a master and a copy of that master, or they are reading two copies of a common master.

2.13. **Channels: A Final View**

We can now summarize the types of the various channel messages first referred to in Items 24, 25, 26 and 27.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 MH = Grant (Item 44c on Page 11)</td>
<td>Copy (Item 55 on the preceding page)</td>
</tr>
<tr>
<td>26 AD = Shred (Item 52 on Page 12)</td>
<td></td>
</tr>
<tr>
<td>27 HD = Edit (Item 53 on Page 12)</td>
<td>Read (Item 54 on the previous page)</td>
</tr>
</tbody>
</table>

2.14. **An Informal Summary of Behaviours**

2.14.1. **The Create Behaviour: Left Fig. 3 on the next page**

1. Handler behaviour $i$, at its own volition, initiates an edit action on document $j$ (where $i$ has editing rights for document $j$). Handler $i$, optionally, provides document $j$ with an annotation note. While editing document $j$ handler $i$ also “selects” an appropriate pair of edit/undo functions for document $j$.
2. Document behaviour $j$ accepts the editing request, enacts the editing, optionally appends the (annotation) note, and, with handler $i$, completes the editing, after some time interval $t_i$.
3. Handler behaviour $i$ completes its edit action.

2.14.2. **The Read Behaviour: Right Fig. 3 on the facing page**

1. Handler behaviour $i$, at its own volition, initiates a read action on document $j$ (where $i$ has reading rights for document $j$). Handler $i$, optionally, provides document $j$ with an annotation note.
The dotted line means:
Initialising the document.

The dotted line means:
Obtaining the document “data”!

2 Document behaviour $j$ accepts the reading request, enacts the reading by providing the handler, $i$, with the document contents, and optionally appends the (annotation) note, and, with handler $i$, completes the reading, after some time interval $t_i$.

3 Handler behaviour $i$ completes its read action.

2.14.4. The Copy Behaviour: Right Fig. 4

1 Handler behaviour $i$, at its own volition, initiates a copy action on document $j$ (where $i$ has copying rights for document $j$). Handler $i$, optionally, provides master document $j$ as well as the copied document (yet to be identified) with respective (annotation) notes.

2 The management behaviour offers to accept the handler message. As for the create action, the management behaviour offers a combined copy and create document message to the archive behaviour.

3 The archive behaviour selects an available document identifier (here shown as $k$), henceforth marking $k$ as used.

4 The archive behaviour then obtains, from the master document $j$ its document descriptor, $dd_j$, its document annotations, $da_j$, its document contents, $dc_j$, and its document history, $dh_j$.

5 The archice behaviour informs the management behaviour of the identifier, $k$, of the (new) document copy.

6 while assembling the attributes for that (new) document copy: its document descriptor, $dd_k$, its document annotations, $da_k$, its document contents, $dc_k$, and its document history, $dh_k$, from these “similar” attributes of the master document $j$,

7 while then “spawning off” document behaviour $docu_k$ – here shown by the “dash–dotted” rounded edge square.

8 The management behaviour accepts the identifier, $k$, of the (new) document copy, recording the identities of the handlers and their access rights to $k$,

9 while informing these handlers (informally indicated by a “dangling” dash-dotted line) of their grants,
2.14.5. The Grant Behaviour: Left Fig. 5

This behaviour has its
1. Item [1] correspond, in essence, to Item [9] of the copy behaviour – see just above – and

2.14.6. The Shred Behaviour: Right Fig. 5

1. The management, at its own volition, selects a document, $j$, to be shredded. It so informs the archive behaviour.
2. The archive behaviour records that document $j$ is to be no longer in use, but shredded, and informs document $j$’s behaviour.
3. The document $j$ behaviour accepts the shred message and stops (indicated by the dotted rounded edge box).

2.15. The Behaviour Actions

To properly structure the definitions of the four kinds of (management, archive, handler and document) behaviours we single each of these out “across” the six behaviour traces informally described in Sects. 2.14.1–2.14.6. The idea is that if behaviour $\beta$ is involved in $\tau$ traces, $\tau_1, \tau_2, \ldots, \tau_\tau$, then behaviour $\beta$ shall be defined in terms of $\tau$ non-deterministic alternative behaviours named $\beta_{\tau_1}, \beta_{\tau_2}, \ldots, \beta_{\tau_\tau}$.

2.15.1. Management Behaviour

The management behaviour is involved in the following action traces:

| a | create                                                                 | Fig. 3 on the previous page Left |
| b | copy                                                                  | Fig. 4 on the preceding page Right |
| c | grant                                                                 | Fig. 5 Left                                 |
| d | shred                                                                  | Fig. 5 Right                                 |

value

62 $\text{mgmt} : \text{MDIR} \to \text{in}, \text{out} \text{mgmt}_\text{arch}_\text{ch} , \{ \text{mgmt}_\text{hdr}_\text{ch}[\text{hi}] | \text{hi} : \text{Hl} \in \text{his} \} \text{ Unit}$

62 $\text{mgmt}(\text{mdir}) \equiv$

62a $\text{mgmt}_\text{create}(\text{mdir})$

62b $\text{mgmt}_\text{copy}(\text{mdir})$

62c $\text{mgmt}_\text{grant}(\text{mdir})$

62d $\text{mgmt}_\text{shred}(\text{mdir})$
Management Create Behaviour: Left Fig. 3 on Page 15

The **management create** behaviour initiates a create document behaviour (i.e., a request to the archive behaviour), and then awaits its response.

\[
\text{value} \\
\text{mgmt}_{\text{create}}: \text{MDIR} \rightarrow \text{in}, \text{out} \quad \text{mgmt}_{\text{arch}} \quad \text{mgmt}_{\text{hdlr}}[\text{hi}] | \text{hi}: \text{HI} | \text{hi} \in \text{his} \quad \text{Unit}
\]

\[\text{mgmt}_{\text{create}}(\text{mdir}) \equiv \]

\[\text{let } \text{hi} = \text{mgmt}_{\text{create}}\text{initiation}(\text{mdir}) ; \quad \text{Left Fig. 3 on Page 15} \]

\[\text{mgmt}_{\text{create}}\text{awaits response}(\text{mdir})(\text{hi}) \quad \text{end} \quad \text{Left Fig. 3 on Page 15} \]

The **management create initiation** behaviour selects a handler on behalf of which it requests the document creation, assembles the elements of the create message:

- by embedding a set of zero or more document references, \(\text{dis}\), with some information, \(\text{info}\), into a document descriptor, adding
- a document note, \(\text{dn}\), and
- and initial, that is, empty document contents, "\text{empty DC}",

offers such a create document message to the archive behaviour, and

yields the identifier of the chosen handler.

\[
\text{value} \\
\text{mgmt}_{\text{create}}\text{initiation}: \text{MDIR} \rightarrow \text{in}, \text{out} \quad \text{mgmt}_{\text{arch}} \quad \text{mgmt}_{\text{hdlr}}[\text{hi}] | \text{hi}: \text{HI} | \text{hi} \in \text{his} \quad \text{HI}
\]

\[\text{mgmt}_{\text{create}}\text{initiation}(\text{mdir}) \equiv \]

\[\text{let } \text{hi} : \text{HI} \quad \text{hi} \in \text{dom mdir}, \]

\[\text{is meaningful}(\text{mdir}) \quad \text{in} \]

\[\text{mgmt}_{\text{arch}} \quad \text{mkCreate}(\text{mdir})(\text{dis}, \text{info}) \quad \text{dn}, \quad \text{end} \quad \text{Left further undefined} \]

The **management create awaits response** behaviour starts by awaiting a reply from the archive behaviour with the identity, \(\text{di}\), of the document (that that behaviour has created).

It then selects suitable access rights,

with which it updates its handler/document directory

and offers to the chosen handler

whereupon it resumes, with the updated management directory, being the management behaviour.

\[
\text{value} \\
\text{mgmt}_{\text{create}}\text{awaits response}: \text{MDIR} \rightarrow \text{HI} \rightarrow \text{in}, \text{out} \quad \text{mgmt}_{\text{arch}} \quad \text{mgmt}_{\text{hdlr}}[\text{hi}] | \text{hi}: \text{HI} | \text{hi} \in \text{his} \quad \text{Unit}
\]

\[\text{mgmt}_{\text{create}}\text{awaits response}(\text{mdir}) \equiv \]

\[\text{let } \text{mkNewDocID}(\text{di}) = \text{mgmt}_{\text{arch}} ? \text{in} \]

\[\text{let } \text{acrs}: \text{ANm} - \text{set} \text{ in} \]

\[\text{let } \text{mdir} = \text{mdir} \uparrow [\text{hi} \rightarrow [\text{di} \rightarrow \text{acrs}]] \text{ in} \]

\[\text{mgmt}_{\text{hdlr}}[\text{hi}] \quad \text{mkGrant}(\text{di}, \text{acrs}) \]

\[\text{mgmt}(\text{mdir}) \quad \text{end} \quad \text{end} \quad \text{end} \]

Management Copy Behaviour: Right Fig. 4 on Page 15

The **management copy** behaviour accepts a copy document request from a handler behaviour (i.e., a request to the archive behaviour),
and then awaits a response from the archive behaviour;
after which it grants access rights to handlers to the document copy.

The **management accept copy** behaviour non-deterministically externally (⌈ ⌉ ⌊ ⌋) awaits a copy request from any handler (hi) behaviour –

80 with the request identifying the master document, j, to be copied.
81 The management accept copy behaviour forwards (!) this request to the archive behaviour –
while yielding the identity of the requesting handler.

82 The **management awaits copy response** behaviour awaits a reply from the archive behaviour as to the identity of the newly created copy (di) of master document j.
83 The management awaits copy response behaviour then informs the ‘copying-requesting’ handler, hi, that the copying has been completed and the identity of the copy (di) –
while yielding the identity, di, of the newly created copy.

84 The **management grants access rights** behaviour

85 selects suitable access rights for a suitable number of selected handlers.
86 It then offers these to the selected handlers.

87 The **management grant** behaviour

is a variant of the mgmt\_grant\_access\_rights function, Items 86–87.
88 The management behaviour selects a suitable subset of known handler identifiers, and
89 for these a suitable subset of document identifiers from which
90 it then constructs a map from handler identifiers to subsets of access rights.
91 With this the management behaviour then issues appropriate grants to the chosen handlers.
type
  MDIR = HI \rightarrow (DI \times \text{ANm-set})
value
88 \text{mgmt\_grant}: MDIR \rightarrow \text{in out} \{\text{mgmt\_hdlr\_ch[hi]} | hi:Hl \cdot hi \in \text{his}\} \text{ Unit}
88 \text{mgmt\_grant(mdir)} \equiv
89 \text{let his} \subseteq \text{dom dir} \text{ in}
90 \text{let dis} \subseteq \bigcup\{\text{dom mdir(hi)} | hi:Hl \cdot hi \in \text{his}\} \text{ in}
91 \text{let diarm} = [hi \rightarrow \text{acs}[hi:Hl,di:DI,\text{acs}\cdot \text{ANm-set} \cdot hi \in \text{his} \land di \in \text{dis} \land \text{acs} \subseteq (\text{diarm(hi)})(di)] \text{ in}
92 \parallel\{\text{mgmt\_hdlr\_ch[hi]} | \text{mkGrant}(di,acs) \mid
93 hi:Hl,di:DI,\text{acs}\cdot \text{ANm-set} \cdot hi \in \text{dom diarm} \land di \in \text{dis} \land \text{acs} \subseteq (\text{diarm(hi)})(di)\}\text{ end end end

Management Shred Behaviour: Right Fig. 5 on Page 16 The \text{management shred} behaviour
93 initiates a request to the archive behaviour.
94 First the management shred behaviour selects a document identifier (from its directory).
95 Then it communicates a shred document message to the archive behaviour;
96 then it notes the (to be shredded) document in its directory.
97 whereupon the management shred behaviour resumes being the management behaviour.

value
93 \text{mgmt\_shred}: MDIR \rightarrow \text{out mgmt\_arch\_ch} \text{ Unit}
93 \text{mgmt\_shred(mdir)} \equiv
94 93 \text{let di:DI} \cdot \text{is suitable(di)}(mdir) \text{ in}
95 93 \text{let mdir}' = [hi \rightarrow \text{mdir(hi)}] \cdot [di,hi:Hl \cdot hi \in \text{dom mdir} | \text{in}
96 \text{mgmt(mdir') end end end

2.15.2. Archive Behaviour

98 The archive behaviour is involved in the following action traces:

  a create Fig. 3 on Page 15 Left
  b copy Fig. 4 on Page 15 Right
  c shred Fig. 5 on Page 16 Right

The Archive Create Behaviour: Left Fig. 3 on Page 15 The \text{archive create} behaviour
99 accepts a request, from the management behaviour to create a document;
100 it then selects an available document identifier;
101 communicates this new document identifier to the management behaviour;
102 while initiating a new document behaviour, docu_{di}, with the document descriptor, \text{dd}, the initial document
  annotation being the singleton list of the note, \text{an}, and the initial document contents, \text{dc} – all received
from the management behaviour – and an initial document history of just one entry: the date of creation, all
103 in parallel with resuming the archive behaviour with updated programmable attributes.

98a. arch_create: AATTR → in, out mgmt.arch.ch, {arch.docu.ch[di]|di:DI\·di ∈ dis} Unit
98a. arch_create(avail,used,gone) ≡
99. [2] let mkCreate((hi,t),dd,an,dc) = mgmt.arch.ch ? in
100. let di:DI \· di ∈ avail in
102. [3] docu_di(dd)(⟨an⟩,dc,<(date of creation)>)
103. ∥ arch(avail\{di\},used∪{di\},gone)
98a. end end

The Archive Copy Behaviour: Right Fig. 4 on Page 15 The archive copy behaviour
104 accepts a copy document request from the management behaviour with the identity, j, of the master document;
105 it communicates (the request to obtain all the attribute values of the master document, j) to that document behaviour;
106 whereupon it awaits their communication (i.e., (dd,da,dc,dh));
107 (meanwhile) it obtains an available document identifier,
108 which it communicates to the management behaviour,
109 while initiating a new document behaviour, docu_di, with the master document descriptor, dd, the master document annotation, and the master document contents, dc, and the master document history, dh (all received from the master document),
110 in parallel with resuming the archive behaviour with updated programmable attributes.

98b. arch_copy: AATTR → in, out mgmt.arch.ch, {arch.docu.ch[di]|di:DI\·di ∈ dis} Unit
98b. arch_copy(avail,used,gone) ≡
104. [3] let mkDocID(j,hi) = mgmt.arch.ch ? in
105. arch.docu.ch[j] ! mkReqAttrs() ;
106. let mkAttrs(dd,da,dc,dh) = arch.docu.ch[j] ? in
107. let di:DI \· di ∈ avail in
108. mgmt.arch.ch ! mkCopyDocID(di) ;
110. ∥ arch(avail\{di\},used∪{di\},gone)
98b. end end end

where we presently leave the [overloaded] augment functions undefined.

The Archive Shred Behaviour: Right Fig. 5 on Page 16 The archive shred behaviour
111 accepts a shred request from the management behaviour.
112 It communicates this request to the identified document behaviour.
113 And then resumes being the archive behaviour, noting however, that the shredded document has been shredded.

98c. arch_shred: AATTR → in, out mgmt.arch.ch, {arch.docu.ch[di]|di:DI\·di ∈ dis} Unit
98c. arch_shred(avail,used,gone) ≡
111. [2] let mkShred(j) = mgmt.arch.ch ? in
112. arch.docu.ch[j] ! mkShred() ;
113. arch(avail,used,gone∪{j})
98c. end
2.15.3. **Handler Behaviours**

The handler behaviour is involved in the following action traces:

- **create**
- **edit**
- **read**
- **copy**
- **grant**

*Fig. 3 on Page 15 Left*  
*Fig. 3 on Page 15 Right*  
*Fig. 4 on Page 15 Left*  
*Fig. 4 on Page 15 Right*  
*Fig. 5 on Page 16 Right*

### The Handler Create Behaviour: Left Fig. 3 on Page 15

The **handler create** behaviour offers to accept the granting of access rights, acrs, to document di. It accordingly updates its programmable hattrs attribute; and resumes being a handler behaviour with that update.

114 hdlr_{hi}: HATTRS \rightarrow \text{in,out} \ mgtm_{hdlr_{ch}[hi]},\{hdlr_{docu_{ch}[hi,di]}|di:DI\cdot di\in\text{dis}\} \ Unit

114 hdlr_{hi}(hattrs) \equiv
114a  hdlr_{create_{hi}}(hattrs)
114b  \begin{cases} hdlr_{edit_{hi}}(hattrs) \end{cases}
114c  \begin{cases} hdlr_{read_{hi}}(hattrs) \end{cases}
114d  \begin{cases} hdlr_{copy_{hi}}(hattrs) \end{cases}
114e  \begin{cases} hdlr_{grant_{hi}}(hattrs) \end{cases}

### The Handler Edit Behaviour: Right Fig. 3 on Page 15

The handler behaviour, on its own volition, decides to edit a document, di, for which it has editing rights. The handler behaviour selects a suitable (...) pair of edit/undo functions and a suitable (annotation) note. It then communicates the desire to edit document di with \((e,u)\) (at time \(t=time_{ch}\)). Editing takes some time, \(ti\). We can therefore assert that the time at which editing has completed is \(t+ti\). The handler behaviour accepts the edit completion message from the document handler. The handler behaviour can therefore resume with an updated document history.

114b hdlr_{edit_{hi}}: HATTRS \times HHIST \rightarrow \text{in,out} \ \{hdlr_{docu_{ch}[hi,di]}|di:DI\cdot di\in\text{dis}\} \ Unit

114b hdlr_{edit_{hi}}(hattrs,hhist) \equiv
118 \begin{cases} 1 \text{let di:DI} \cdot \text{di} \in \text{dom hattrs} \land "edit" \in \text{hattrs(di)} \in \end{cases}
119 \begin{cases} 1 \text{let (e,u):(EDIT}\times\text{UNDO}) \cdot \text{...} \cdot \text{n:AN} \cdot \text{...} \in \end{cases}
120 \begin{cases} 1 \text{hdlg_{docu_{ch}[hi,di]}! mkEdit(hi,t=time_{ch},e,u,n)} ; \end{cases}
121 \begin{cases} 2 \text{let ti:TIME\_INTERVAL} \cdot \text{...} \in \end{cases}
122 \begin{cases} 2 \text{wait ti} ; \text{assert: time_{ch}? = t+ti} \end{cases}
123 \begin{cases} 3 \text{let mkEditComplete(ti',...) = hdlr_{docu_{ch}[hi,di]} \in \text{assert ti' \cong ti} \end{cases}
124 \begin{cases} 4 \text{hdlg_{hi}(hattrs,augment(hhist,(di,mkEdit(hi,t,t,e,u))))} \end{cases}
114b \end{end end end end
The Handler Read Behaviour: Left Fig. 4 on Page 15

125 The handler behaviour, on its own volition, decides to read a document, \( di \), for which it has reading rights.

126 It then communicates the desire to read document \( di \) with at time \( t = \text{time}_{\text{ch}} \) – with an annotation note \( (n) \).

127 Reading take some time, \( ti \).

128 We can therefore assert that the time at which reading has completed is \( t + ti \).

129 The handler behaviour accepts the read completion message from the document handler.

130 The handler behaviour can therefore resume with an updated document history.

The Handler Copy Behaviour: Right Fig. 4 on Page 15

131 The handler [copy] behaviour, on its own volition, decides to copy a document, \( di \), for which it has copying rights.

132 It communicates this copy request to the management behaviour.

133 After a while the handler [copy] behaviour receives acknowledgement of a completed copying from the management behaviour.

134 The handler [copy] behaviour records the request and acknowledgement in its, thus updated whereupon the handler [copy] behaviour resumes being the handler behaviour.

The Handler Grant Behaviour: Left Fig. 5 on Page 16

135 The handler [grant] behaviour offers to accept grant permissions from the management behaviour.

In response it updates its handler attribute while resuming being a handler behaviour.

2.15.4. Document Behaviours

137 The document behaviour is involved in the following action traces:

---

Fig. 3 on Page 15 Right
The Document Edit Behaviour: Right Fig. 3 on Page 15

138 The document [edit] behaviour offers to accept edit requests from document handlers.

a. The document contents is edited, over a time interval of \( t_i \), with respect to the handlers edit function \((e)\),
b. the document annotations are augmented with respect to the handlers note \((n)\), and
c. the document history is augmented with the fact that an edit took place, at a certain time, with a pair of edit/undo functions.

139 The edit (etc.) function(s) take some time, \( t_i \), to do.

140 The handler behaviour is notified, \( \text{mkEditComplete}(\ldots) \) of the completion of the edit, and

141 the document behaviour is then resumed with updated programmable attributes.

\[
\begin{align*}
\text{value} \\
137 \text{docu}_{d_i}: DD \times (DA \times DC \times DH) & \rightarrow \text{in, out arch}_{docu,ch}[di], \text{hdr}_{docu,ch}[hi,di,hi:Hl\cdot hi\in his]} \text{ Unit} \\
137 \text{docu}_{d_i}(dattrs) & \equiv \\
137a \text{docu}_{edit}_{d_i}(dd)(da,dc,dh) \\
137b \text{docu}_{read}_{d_i}(dd)(da,dc,dh) \\
137c \text{docu}_{shred}_{d_i}(dd)(da,dc,dh)
\end{align*}
\]

The Document Read Behaviour: Left Fig. 4 on Page 15

142 The document [read] behaviour offers to receive a read request from a handler behaviour.

143 The reading takes some time to do.

144 The handler behaviour is advised on completion.

145 And the document behaviour is resumed with appropriate programmable attributes being updated.

\[
\begin{align*}
\text{value} \\
137 \text{docu}_{d_i}: DD \times (DA \times DC \times DH) & \rightarrow \text{in, out \{hdr_{docu,ch}[hi,di,hi:Hl\cdot hi\in his]} \text{ Unit} \\
137 \text{docu}_{d_i}(dattrs) & \equiv \\
138 \begin{cases} 
138a \text{docu}_{edit}_{d_i}(dd)(da,dc,dh) \\
138b \text{docu}_{read}_{d_i}(dd)(da,dc,dh) \\
138c \text{docu}_{shred}_{d_i}(dd)(da,dc,dh)
\end{cases}
\end{align*}
\]
The Document Shred Behaviour: Right Fig. 5 on Page 16

146 The document [shred] behaviour offers to accept a document shred request from the archive behaviour _

147 whereupon it stops!

\[
\text{value}
\]

\[
137c \quad \text{docu}_{\text{shred}_{di}} : DD \times (DA \times DC \times DH) \rightarrow \text{in,out} \arch_{\text{docu},ch}[di] \text{ Unit}
\]

\[
137c \quad \text{docu}_{\text{shred}_{di}}(dd)(da,dc,dh) =
\]

\[
146 \quad \text{let mkShred(...)} = \arch_{\text{docu},ch}[di] ? \text{ in}
\]

\[
147 \quad \text{stop}
\]

\[
137c \quad [3] \quad \text{end}
\]

2.16. Conclusion

This completes a first draft version of this document. The date time is: 15 June 2017: 10:43 am. Many things need to be done. First a careful checking of all types and functions: that all used names have been defined. The internal non-deterministic choices in formula Items 62 on Page 16, 98 on Page 19, 114 on Page 21 and 137 on Page 22, need be checked. I suspect there should, instead, be som mix of both internal and external non-deterministic choices. Then a careful motivation for all the other non-deterministic choices.

3. Bibliography

3.1. Bibliographical Notes

I have thought about domain engineering for more than 20 years. But serious, focused writing only started to appear since [7, Part IV] — with [4, 1] being exceptions: [9] suggests a number of domain science and engineering research topics; [13] covers the concept of domain facets; [33] explores compositionality and Galois connections. [10, 32] show how to systematically, but, of course, not automatically, “derive” requirements prescriptions from domain descriptions; [15] takes the tritycht software development as a basis for outlining principles for believable software management; [11, 22] presents a model for Stanisław Leśniewski’s [35] concept of mereology; [14, 16] present an extensive example and is otherwise a precursor for the present paper; [17] presents, based on the TripTych view of software development as ideally proceeding from domain description via requirements prescription to software design, concepts such as software demos and simulators; [19] analyses the TripTych, especially its domain engineering approach, with respect to [38, 39, Maslow]’s and [40, Peterson’s and Seligman’s]’s notions of humanity: how can computing relate to notions of humanity; the first part of [23] is a precursor for [26] with the second part of [23] presenting a first formal model of the elicitation process of analysis and description based on the prompts more definitively presented in [26]; and with [24] focus on domain safety criticality. The published paper [26] now constitutes the base introduction to domain science & engineering.

3.2. Domain Modeling Experiments

- Weather Information Systems\textsuperscript{15} [27] Result of my PhD lectures at Bergen, November 2016
- Documents\textsuperscript{16} [18] 2013.
- Transport Systems\textsuperscript{17} [21] 2010.

\textsuperscript{14} http://www.imm.dtu.dk/~dibj/2016/uppsala/accs.pdf
\textsuperscript{15} http://www.imm.dtu.dk/~dibj/2016/wis/wis-p.pdf
\textsuperscript{16} http://www.imm.dtu.dk/~dibj/doc-p.pdf
\textsuperscript{17} http://www.imm.dtu.dk/~dibj/comet/comet1.pdf
• The Tokyo Stock Exchange Trading Rules\textsuperscript{18} and\textsuperscript{19} [29] 2010.
• On Development of Web-based Software\textsuperscript{20} 2010.
• Pipelines – a Domain Description\textsuperscript{22} and\textsuperscript{23}, [20] 2009.
• Platooning\textsuperscript{24}.
• A Container Line Industry Domain\textsuperscript{25}, [8] 2007
• Markets\textsuperscript{27} [3]

\section*{Railway Systems Descriptions: 1996–2003}

\begin{itemize}
  \item Dines Bjørner: Formal Software Techniques in Railway Systems\textsuperscript{28} [2]
  \item Chris George, Dines Bjørner and Søren Prehn: Scheduling and Rescheduling of Trains\textsuperscript{29}, [34] 1996
  \item Dines Bjørner: A Railway Systems Domain\textsuperscript{30} An "old" UNU-IIST report, 1997
  \item Dines Bjørner: Formal Software Techniques in Railway Systems\textsuperscript{31}, 2002
  \item Albena Strupchanska, Martin Penicka and Dines Bjørner: Railway Staff Rostering\textsuperscript{32}, 2003 [42]
  \item Dines Bjørner: Dynamics of Railway Nets\textsuperscript{33}, 2003 [5]
  \item Martin Penicka, Albena Strupchanska and Dines Bjørner: Train Maintenance Routing\textsuperscript{34}, 2003 [41]
  \item Panagiotis Karras and Dines Bjørner: Train Composition and Decomposition: Domain and Requirements\textsuperscript{35}, 2003
  \item Dines Bjørner: Dynamics of Railway Nets: On an Interface between Automatic Control and Software Engineering\textsuperscript{36} [5] 2003
\end{itemize}

\section*{References}

\begin{thebibliography}{99}

\end{thebibliography}


Dines Bjørner, Chris W. George, and Søren Prehn. Scheduling and Rescheduling of Trains, chapter 8, pages 157–


A. Documents in Public Government

Public government, in the spirit of Charles-Louis de Secondat, Baron de La Brède et de Montesquieu (or just Montesquieu), has three branches:

- the legislative,
- the executive, and
- the judicial.

Our interpretation of these, with respect to documents, are as follows.

- The legislative branch produces laws, i.e., documents. To do so many preparatory documents are created, edited, read, copied, etc. Committees, subcommittees, individual lawmakers and ministry law office staff handles these documents. Parliament staff and legislators are granted limited or unlimited access rights to these documents. Finally laws are put into effect, are amended, changed or abolished. The legislative branch documents refer to legislative, executive and judicial branch documents.

- The executive branch produces guidelines, i.e., documents. Instructions on interpretation and implementation of laws; directives to ministry services on how to handle the laws; etcetera. These executive branch documents refer to legislative, executive and judicial branch documents.

- The judicial branch produces documents. Police cite citizens and enterprises for breach of law. Citizens and enterprise sue other citizens and/or enterprises. Attorneys on behalf of the governments, or citizens or enterprises prepare statements. Court proceedings are recorded. Justices pass verdicts. The judicial branch documents refer to legislative, executive and judicial branch documents.
B. Documents in Urban Planning

A separate research note [28, Urban Planning Processes] analyses & describes a domain of urban planning. There are the geographical documents:

- geodetic,
- geotechnic,
- meteorological,
- and other types of geographical documents.

In order to perform an informed urban planning further documents are needed:

- auxiliary documents which
- requirements documents which

Auxiliary documents present such information that “fill in” details concerning current ownership of the land area, current laws affecting this ownership, the use of the land, etcetera. Requirements documents express expectations about the (base) urban plans that should result from the base urban planning. As a first result of base urban planning we see the emergence of the following kinds of documents:

- base urban plans
- and ancillary notes.

The base urban plans deal with

- cadastral,
- cartographic and
- zoning

issues. The ancillary notes deal with such things as insufficiencies in the base plans, things that ought be improved in a next iteration base urban planning, etc. The base plans and ancillary notes, besides possible re-iteration of base urban planning, lead on to “derived urban planning” for

- light, medium and heavy industry zones,
- mixed shopping and residential zones,
- apartment building zones,
- villa zones,
- recreational zones,
- etcetera.

After these “first generation” derived urban plans are well underway, a “second generation” derived urban planning can start:

- transport infrastructure,
- water and waste resource management,
- electricity, natural gas, etc., infrastructure,
- etcetera.

And so forth. Literally “zillions upon zillions” of strongly and crucially interrelated documents accrue.

Urban planning evolves and revolves around documents.

Documents are the only “tangible” results or urban planning.\(^\text{40}\)

\(^{40}\) Once urban plans have been agreed upon by all relevant authorities and individuals, then urban development (“build”) and, finally, “operation” of the developed, new urban “landscape”. For development, the urban plans form one of the “tangible” inputs. Others are of financial and human and other resource nature.
C. RSL: The RAISE Specification Language – A Primer

C.1. Type Expressions

Type expressions are expressions whose value are types, that is, possibly infinite sets of values (of “that” type).

C.1.1. 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.

```
type
1  Bool   true, false
2  Int    ..., −2, −1, 0, 1, 2, ...
3  Nat    0, 1, 2, ...
4  Real   ..., −5.43, −1.0, 0.0, 1.23 · · ·, 2,7182 · · ·, 3,1415 · · ·, 4.56, ...
5  Char   "a", "b", ..., "0", ...
6  Text   "abracadabra"
```

C.1.2. Composite Types

Composite types have composite values. That is, values which we consider to have proper constituent (sub-)values, i.e., can be meaningfully “taken apart”. There are two ways of expressing composite types: either explicitly, using concrete type expressions, or implicitly, using sorts (i.e., abstract types) and observer functions.

Concrete Composite Types 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 the following are type expressions:

```
| 7  | A-set  | 13 | A → B  |
| 8  | A-infset | 14 | A ⇆ B  |
| 9  | A × B × ... × C | 15 | (A)   |
| 10 | A*     | 16 | A | B | ... | C  |
| 11 | A^ω   | 17 | mk_id(sel_a:A,...,sel_b:B) |
| 12 | A ↦ B  | 18 | sel_a:A ... sel_b:B |
```

The following the meaning of the atomic and the composite type expressions:

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", "bb", ...
6. The text type of character string values "aa", "aaa", ..., "abc", ...
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:
   \begin{itemize}
   \item either a Cartesian \(B \times C \times \ldots \times D\), in which case it is identical to type expression kind 9,
   \item 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., \((A \rightarrow B)\), or \((A^\ast)\)-set, or \((A\text{-set})\)list, or \((A|B) \rightarrow (C|D|(E \rightarrow F))\), etc.
   \end{itemize}
16 The postulated disjoint union of types \(A, B, \ldots,\) and \(C\).
17 The record type of \(\text{mk\_id}\)-named record values \(\text{mk\_id}(av, \ldots, bv)\), where \(av, \ldots, bv\), are values of respective
   types. The distinct identifiers \(\text{sel\_a}\), etc., designate selector functions.
18 The record type of unnamed record values \((av, \ldots, bv)\), where \(av, \ldots, bv\), are values of respective types.
   The distinct identifiers \(\text{sel\_a}\), etc., designate selector functions.

**Sorts and Observer Functions**

\[
type \\
A, B, C, \ldots, D \\
\]
\[
value \\
ob_1: A \rightarrow B, \ obs_2: A \rightarrow C, \ldots, \ obs_D: A \rightarrow D
\]

The above expresses that values of type \(A\) are composed from at least three values — and these are of type \(B, C, \ldots,\) and \(D\). A concrete type definition corresponding to the above presupposing material of the next

\[
type \\
B, C, \ldots, D \\
A = B \times C \times \ldots \times D
\]

**C.2. Type Definitions**

**C.2.1. Concrete Types**

Types can be concrete in which case the structure of the type is specified by type expressions:

\[
type \\
A = \text{Type\_expr}
\]

Some schematic type definitions are:

\[
[19] \text{Type\_name} = \text{Type\_expr} /* without \mid s or subtypes */ \\
[20] \text{Type\_name} = \text{Type\_expr\_1} \mid \text{Type\_expr\_2} \mid \ldots \mid \text{Type\_expr\_n} \\
[21] \text{Type\_name} == \\
    \text{mk\_id\_1}(s_{a1}:\text{Type\_name\_a1},\ldots,s_{ai}:\text{Type\_name\_ai}) \mid \\
    \ldots \\
    \text{mk\_id\_n}(s_{z1}:\text{Type\_name\_z1},\ldots,s_{zk}:\text{Type\_name\_zk}) \\
[22] \text{Type\_name} :: \text{sel\_a:Type\_name\_a} \ldots \text{sel\_z:Type\_name\_z} \\
[23] \text{Type\_name} = \{ v:\text{Type\_name'} \rightarrow \text{\_P(v)} \}
\]

where a form of [20]–[21] is provided by combining the types:

\[
\text{Type\_name} = A \mid B \mid \ldots \mid Z \\
A == \text{mk\_id\_1}(s_{a1}:A_{a1},\ldots,s_{ai}:A_{ai}) \\
B == \text{mk\_id\_2}(s_{b1}:B_{b1},\ldots,s_{bj}:B_{bj}) \\
\ldots \\
Z == \text{mk\_id\_n}(s_{z1}:Z_{z1},\ldots,s_{zk}:Z_{zk})
\]
Types A, B, ..., Z are disjoint, i.e., shares no values, provided all \( \text{mk}_{id}k \) are distinct and due to the use of the disjoint record type constructor ==.

**axiom**
\[
\forall a_1:A_1, a_2:A_2, \ldots, a_i:A_i \cdot \\
\text{s}_{a_1}(\text{mk}_{id}1(a_1, a_2, \ldots, a_i)) = a_1 \land \\
\text{s}_{a_2}(\text{mk}_{id}1(a_1, a_2, \ldots, a_i)) = a_2 \land \\
\ldots \land \\
\text{s}_{a_i}(\text{mk}_{id}1(a_1, a_2, \ldots, a_i)) = a_i \\
\forall a:A \cdot \text{let mk}_{id}1(a_1', a_2', \ldots, a_i') = a \text{ in} \\
a_1' = \text{s}_{a_1}(a) \land a_2' = \text{s}_{a_2}(a) \land \ldots \land a_i' = \text{s}_{a_i}(a) \text{ end}
\]

### C.2.2. 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 \( P \), constitute the subtype \( A \):

**type**
\[
A = \{ \mid b:B \cdot P(b) \}
\]

### C.2.3. Sorts — Abstract Types

Types can be (abstract) sorts in which case their structure is not specified:

**type**
\[
A, B, \ldots, C
\]

### C.3. The RSL Predicate Calculus

### C.4. Propositional Expressions

Let identifiers (or propositional expressions) \( a, b, \ldots, c \) designate Boolean values (true or false [or chaos]). Then:

- \( \text{false, true} \)
- \( a, b, \ldots, c \sim a, a \land b, a \lor b, a \Rightarrow b, a = b, a \neq b \)

are propositional expressions having Boolean values. \( \sim, \land, \lor, \Rightarrow, = \) and \( \neq \) are Boolean connectives (i.e., operators). They can be read as: not, and, or, if then (or implies), equal and not equal.

### C.4.1. Simple Predicate Expressions

Let identifiers (or propositional expressions) \( a, b, \ldots, c \) designate Boolean values, let \( x, y, \ldots, z \) (or term expressions) designate non-Boolean values and let \( i, j, \ldots, k \) designate number values, then:

- \( \text{false, true} \)
- \( a, b, \ldots, c \sim a, a \land b, a \lor b, a \Rightarrow b, a = b, a \neq b \)
- \( x = y, x \neq y, i < j, i \leq j, i \geq j, i \neq j, i > j \)

are simple predicate expressions.

### C.4.2. Quantified Expressions

Let \( X, Y, \ldots, C \) be type names or type expressions, and let \( P(x), Q(y) \) and \( R(z) \) designate predicate expressions in which \( x, y \) and \( z \) are free. Then:
∀ x:X • P(x)
∃ y:Y • Q(y)
∃ ! z:Z • R(z)

are quantified expressions — also being predicate expressions.
They are “read” as: For all x (values in type X) the predicate P(x) holds; there exists (at least) one y (value in type Y) such that the predicate Q(y) holds; and there exists a unique z (value in type Z) such that the predicate R(z) holds.

C.5. Concrete RSL Types: Values and Operations

C.5.1. Arithmetic

type
  Nat, Int, Real
value
  +, −, ∗: Nat × Nat → Nat | Int × Int → Int | Real × Real → Real
  /: Nat × Nat → Nat | Int × Int → Int | Real × Real → Real
  <, ≤, =, ≠, ≥, > (Nat | Int | Real) → (Nat | Int | Real)

C.5.2. Set Expressions

Set Enumerations Let the below a’s denote values of type A, then the below designate simple set enumerations:

\{\{\}, \{a\}, \{e_1, e_2, \ldots, e_n\}, \ldots\} ∈ A-set
\{\{\}, \{a\}, \{e_1, e_2, \ldots, e_n\}, \ldots, \{e_1, e_2, \ldots\}\} ∈ 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.

type
  A, B
  P = A → Bool
  Q = A ⊨ B
value
  comprehend: A-infset × P × Q → B-infset
  comprehend(s,P,Q) ≡ \{ Q(a) | a:A • a ∈ s ∧ P(a)\}

C.5.3. Cartesian Expressions

Cartesian Enumerations Let e range over values of Cartesian types involving A, B, . . . , C, then the below expressions are simple Cartesian enumerations:

type
  A, B, ..., C
  A × B × ... × C
value
  (e_1, e_2, ..., e_n)

C.5.4. List Expressions

List Enumerations Let a range over values of type A, then the below expressions are simple list enumerations:
\{\langle\rangle, \langle e\rangle, ..., \langle e_1, e_2, ..., e_n\rangle, ...\}\in A^*
\{\langle\rangle, \langle e\rangle, ..., \langle e_1, e_2, ..., e_n\rangle, ..., \langle e_1, e_2, ..., e_n\rangle\}\in A^\omega

\langle a_i \ldots a_j \rangle

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.

\[
\text{type}\quad A, B, P = A \to \text{Bool}, \quad Q = A \leadsto B
\]
\[
\text{value}\quad \text{comprehend}: \quad A^{\omega} \times P \times Q \leadsto B^{\omega}
\]
\[
\text{comprehend}(l, P, Q) \equiv \langle Q(l(i)) \mid i \in \langle 1.. \text{len} l \rangle \cdot P(l(i)) \rangle
\]

**C.5.5. Map Expressions**

**Map Enumerations** Let (possibly indexed) \(u\) and \(v\) range over values of type \(T_1\) and \(T_2\), respectively, then the below expressions are simple map enumerations:

\[
\text{type}\quad T_1, T_2
\]
\[
M = T_1 \rightarrow T_2
\]
\[
\text{value}\quad u, u_1, u_2, ..., u_n : T_1, \quad v, v_1, v_2, ..., v_n : T_2
\]
\[
\{[], [u \mapsto v], ..., [u_1 \mapsto v_1, u_2 \mapsto v_2, ..., u_n \mapsto v_n] \mid \text{all } \in M \}
\]

**Map Comprehension** The last line below expresses map comprehension:

\[
\text{type}\quad U, V, X, Y
\]
\[
M = U \rightarrow V
\]
\[
F = U \rightarrow X
\]
\[
G = V \rightarrow Y
\]
\[
P = U \rightarrow \text{Bool}
\]
\[
\text{value}\quad \text{comprehend}: \quad M \times F \times G \times P \rightarrow (X \rightarrow Y)
\]
\[
\text{comprehend}(m, F, G, P) \equiv \{ F(u) \mapsto G(m(u)) \mid u : U \cdot u \in \text{dom } m \land P(u) \}
\]

**C.5.6. Set Operations**

**Set Operator Signatures**

\[
\text{value}\quad 19 \in: A \times A\text{-infset} \rightarrow \text{Bool}
\]
\[
20 \notin: A \times A\text{-infset} \rightarrow \text{Bool}
\]
\[
21 \cup: A\text{-infset} \times A\text{-infset} \rightarrow A\text{-infset}
\]
\[
22 \cup: (A\text{-infset})\text{-infset} \rightarrow A\text{-infset}
\]
\[
23 \cap: A\text{-infset} \times A\text{-infset} \rightarrow A\text{-infset}
\]
\[
24 \cap: (A\text{-infset})\text{-infset} \rightarrow A\text{-infset}
\]
\[
25 \setminus: A\text{-infset} \times A\text{-infset} \rightarrow A\text{-infset}
\]
\[
26 \subseteq: A\text{-infset} \times A\text{-infset} \rightarrow \text{Bool}
\]
\[
27 \not\subseteq: A\text{-infset} \times A\text{-infset} \rightarrow \text{Bool}
\]
\[
28 =: A\text{-infset} \times A\text{-infset} \rightarrow \text{Bool}
\]
\[
29 \neq: A\text{-infset} \times A\text{-infset} \rightarrow \text{Bool}
\]
\[
30 \text{card}: A\text{-infset} \rightarrow \text{Nat}
\]
Set Examples

examples

\[ a \in \{a,b,c\} \]
\[ a \notin \{\} \]
\[ a \notin \{b,c\} \]
\[ \{a,b,c\} \cup \{a,b,d,e\} = \{a,b,c,d,e\} \]
\[ \cup \{\{a\},\{a,bb\},\{a,d\}\} = \{a\} \]
\[ \{a,b,c\} \setminus \{c,d\} = \{a,bb\} \]
\[ \{a,bb\} \subset \{a,b,c\} \]
\[ \{a,bb\} \subseteq \{a,b,c\} \]
\[ \{a,bb\} \neq \{a,b,c\} \]
\[ \text{card} \{\} = 0, \text{card} \{a,b,c\} = 3 \]

Informal Explication

19 \(\in\): The membership operator expresses that an element is a member of a set.
20 \(\notin\): The nonmembership operator expresses that an element is not a member of a set.
21 \(\cup\): 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 \(\cup\): 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 \(\cap\): 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 \(\cap\): 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 \(\setminus\): 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 \(\subseteq\): The proper subset operator expresses that all members of the left operand set are also in the right operand set.
27 \(\subset\): 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 \(\equiv\): The equal operator expresses that the two operand sets are identical.
29 \(\neq\): The nonequal operator expresses that the two operand sets are not identical.
30 \text{card}: The cardinality operator gives the number of elements in a finite set.

Set Operator Definitions

The operations can be defined as follows (\(\equiv\) is the definition symbol):

\[
\text{value} \\
\quad s' \cup s'' \equiv \{ a \mid a : A \land a \in s' \lor a \in s'' \} \\
\quad s' \cap s'' \equiv \{ a \mid a : A \land a \in s' \land a \in s'' \} \\
\quad s' \setminus s'' \equiv \{ a \mid a : A \land a \in s' \land a \notin s'' \} \\
\quad s' \subseteq s'' \equiv \forall a : A \land a \in s' \Rightarrow a \in s'' \\
\quad s' \subset s'' \equiv s' \subseteq s'' \land \exists a : A \land a \in s'' \land a \notin s' \\
\quad s' = s'' \equiv \forall a : A \land a \in s' \equiv a \in s'' \equiv s' \subseteq s' \land s' \subseteq s \\
\quad s' \neq s'' \equiv s' \cap s'' \neq \{\} \\
\quad \text{card} s \equiv \begin{cases} 0 & \text{if } s = \{\} \\ \text{let } a : A \land a \in s \text{ in } 1 + \text{card} (s \setminus \{a\}) \text{ end end} \\ \text{pre } s /\ast \text{ is a finite set } /\ast/ \\ \text{card} s \equiv \text{chaos} /\ast/ \text{ tests for infinity of } s /\ast/ 
\]
C.5.7. Cartesian Operations

\[ g_0 : G_0 = A \times B \times C \]
\[ g_1 : G_1 = (A \times B \times C) \]
\[ g_2 : G_2 = (A \times B) \times C \]
\[ g_3 : G_3 = A \times (B \times C) \]

value
\[ (va, vb, vc) : G_0, \]
\[ ((va, vb), vc) : G_2 \]
\[ (va_3, (vb_3, vc_3)) : G_3 \]

decomposition expressions
\[ \text{let } (a_1, b_1, c_1) = g_0, \]
\[ (a_1', b_1', c_1') = g_1 \text{ in } \ldots \text{ end} \]
\[ \text{let } ((a_2, b_2), c_2) = g_2 \text{ in } \ldots \text{ end} \]
\[ \text{let } (a_3, (b_3, c_3)) = g_3 \text{ in } \ldots \text{ end} \]

C.5.8. List Operations

List Operator Signatures

value
\[ \text{hd} : A^\omega \rightsquigarrow A \]
\[ \text{tl} : A^\omega \rightsquigarrow A^\omega \]
\[ \text{len} : A^\omega \rightsquigarrow \text{Nat} \]
\[ \text{inds} : A^\omega \rightarrow \text{Nat-infset} \]
\[ \text{elems} : A^\omega \rightarrow A^\text{infset} \]
\[ .(i) : A^\omega \times \text{Nat} \rightsquigarrow A \]
\[ ^\circ : A^* \circ A^\omega \circ A^\omega \rightarrow A^\omega \]
\[ ^\neq : A^* \circ A^\omega \neq A^* \circ A^\omega \]

List Operation Examples

examples
\[ \text{hd}(a_1, a_2, \ldots, a_m) = a_1 \]
\[ \text{tl}(a_1, a_2, \ldots, a_m) = (a_2, \ldots, a_m) \]
\[ \text{len}(a_1, a_2, \ldots, a_m) = m \]
\[ \text{inds}(a_1, a_2, \ldots, a_m) = \{1, 2, \ldots, m\} \]
\[ \text{elems}(a_1, a_2, \ldots, a_m) = \{a_1, a_2, \ldots, a_m\} \]
\[ (a_1, a_2, \ldots, a_m)(i) = a_i \]
\[ (a, b, c)^\circ (a, b, d) = (a, b, c, a, b, d) \]
\[ (a, b, c) = (a, b, c) \]
\[ (a, b, c) \neq (a, b, d) \]

Informal Explication

- \text{hd}: Head gives the first element in a nonempty list.
- \text{tl}: Tail gives the remaining list of a nonempty list when Head is removed.
- \text{len}: Length gives the number of elements in a finite list.
- \text{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.
- \text{elems}: Elements gives the possibly infinite set of all distinct elements in a list.
- \text{\ell}(i): Indexing with a natural number, \( i \) larger than 0, into a list \( \ell \) having a number of elements larger than or equal to \( i \), gives the \( i \)th element of the list.
- \( ^\circ \): 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.
- \( \neq \): The nonequal operator expresses that the two operand lists are not identical.

The operations can also be defined as follows:
List Operator Definitions

value

\text{is\_finite\_list}: A^\omega \to \text{Bool}

\text{len} q \equiv
  \begin{cases} 
    \text{true} & \text{if } q = () \text{ then } 0 \text{ else } 1 + \text{len } \text{tl } q \text{ end}, \\
    \text{false} & \text{end}  
  \end{cases}

\text{inds} q \equiv
  \begin{cases} 
    \text{true} & \{ i \mid i: \text{Nat} \cdot 1 \leq i \leq \text{len } q \}, \\
    \text{false} & \{ i \mid i: \text{Nat} \cdot i \neq 0 \} \text{ end}  
  \end{cases}

\text{elems} q \equiv \{ q(i) \mid i: \text{Nat} \cdot i \in \text{inds } q \}

q(i) \equiv
  \begin{cases} 
    \text{if } i = 1 
    \begin{cases} 
      \text{if } q \neq () 
      \begin{cases} 
        \text{then let } a:A, q'Q: q = \langle a \rangle \sim q' \text{ in } a \text{ end} \\
        \text{else } \text{chaos } \text{ end}  
      \end{cases} 
    \end{cases} 
    \text{else } q(i - 1) \text{ end}  
  \end{cases}

fq \sim iq' \equiv
  \begin{cases} 
    \text{if } 1 \leq i \leq \text{len } fq \text{ then } fq(i) \text{ else } iq(i - \text{len } fq) \text{ end} \\
    | i: \text{Nat} \cdot \text{if len } iq \neq \text{chaos } \text{then } i \leq \text{len } fq + \text{len } end  
  \end{cases}

\text{pre } \text{is\_finite\_list}(fq)

iq' = iq'' \equiv
  \text{inds } iq' = \text{inds } iq'' \land \forall i: \text{Nat} \cdot i \in \text{inds } iq' \Rightarrow iq'(i) = iq''(i)

iq' \neq iq'' \equiv \sim(iq' = iq'')

C.5.9. Map Operations

Map Operator Signatures and Map Operation Examples

value

m(a): M \to A \sim B, m(a) = b

\text{dom}: M \to A-\text{infset } \text{[domain of map]}

\begin{array}{l}
  \text{dom } [a_1 \mapsto b_1, a_2 \mapsto b_2, \ldots, a_n \mapsto b_n] = \{ a_1, a_2, \ldots, a_n \}
\end{array}

\text{rng}: M \to B-\text{infset } \text{[range of map]}

\begin{array}{l}
  \text{rng } [a_1 \mapsto b_1, a_2 \mapsto b_2, \ldots, a_n \mapsto b_n] = \{ b_1, b_2, \ldots, b_n \}
\end{array}

\text{†}: M \times M \to M \text{ [override extension]}

\begin{array}{l}
  [a \mapsto b, a' \mapsto bb, a'' \mapsto bb'] \uparrow [a' \mapsto bb'', a'' \mapsto bb''] = [a \mapsto b, a' \mapsto bb'', a'' \mapsto bb']
\end{array}

\text{∪}: M \times M \to M \text{ [merge } \cup \text{]}

\begin{array}{l}
  [a \mapsto b, a' \mapsto bb, a'' \mapsto bb'] \cup [a'' \mapsto bb'''] = [a \mapsto b, a' \mapsto bb, a'' \mapsto bb'', a''\mapsto bb''']
\end{array}

\backslash: M \times A-\text{infset } \to M \text{ [restriction by]}

\begin{array}{l}
  [a \mapsto b, a' \mapsto bb, a'' \mapsto bb'] \backslash \{ a \} = [a' \mapsto bb, a'' \mapsto bb']
\end{array}
Map Operation Explication

- $m(a)$: Application gives the element that $a$ maps to in the map $m$.
- $\text{dom}$: Domain/Definition Set gives the set of values which maps to in a map.
- $\text{rng}$: Range/Image Set gives the set of values which are mapped to in a map.
- $\triangleright$: 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.
- $\cup$: Merge. When applied to two operand maps, it gives a merge of these maps.
- $\setminus$: 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.
- $\neq$: The nonequal operator expresses that the two operand maps are not identical.
- $\circ$: 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

\begin{align*}
\text{rng} \ m & \equiv \{ \ m(a) \mid a:A \cdot a \in \text{dom} \ m \} \\
\text{dom} \ m_1 \text{=} & \text{dom} \ m_2 \ \land \ \forall \ a:A \cdot a \in \text{dom} \ m_1 \Rightarrow m_1(a) = m_2(a) \\
\text{dom} \ m_1 \text{=} & \text{dom} \ m_2 \ \land \ a:A \cdot a \in \text{dom} \ m_1 \Rightarrow m_1(a) = m_2(a) \\
\text{pre} \ \text{rng} \ m & \subseteq \text{dom} \ n
\end{align*}
C.6. \(\lambda\)-Calculus + Functions

C.6.1. The \(\lambda\)-Calculus Syntax

type /* A BNF Syntax: */

\(\langle L \rangle ::= \langle V \rangle \mid \langle F \rangle \mid \langle A \rangle \mid (\langle A \rangle)\)

\(\langle V \rangle ::= /* variables, i.e. identifiers */\)

\(\langle F \rangle ::= \lambda \langle V \rangle \cdot (\langle L \rangle)\)

\(\langle A \rangle ::= \langle (L)(L) \rangle\)

value /* Examples */

\(\langle L \rangle: e, f, a, ...

\(\langle V \rangle: x, ...

\(\langle F \rangle: \lambda x \cdot e, ...

\(\langle A \rangle: f a, (f a), f(a), (f)(a), ...

C.6.2. Free and Bound Variables

Let \(x, y\) be variable names and \(e, f\) be \(\lambda\)-expressions.

- (V): Variable \(x\) is free in \(x\).
- (F): \(x\) is free in \(\lambda y \cdot e\) 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.6.3. Substitution

In RSL, the following rules for substitution apply:

- \(\text{subst}(\langle N/x \rangle x) \equiv N;\)
- \(\text{subst}(\langle N/x \rangle a) \equiv a,\)
  for all variables \(a \neq x;\)
- \(\text{subst}(\langle N/x \rangle (P Q)) \equiv \text{subst}(\langle N/x \rangle P) \text{subst}(\langle N/x \rangle Q);\)
- \(\text{subst}(\langle N/x \rangle (\lambda y \cdot P)) \equiv \lambda y \cdot P;\)
- \(\text{subst}(\langle N/x \rangle (\lambda y \cdot P)) \equiv \lambda y \cdot \text{subst}(\langle N/x \rangle P),\)
  if \(x \neq y\) and \(y\) is not free in \(N\) or \(x\) is not free in \(P;\)
- \(\text{subst}(\langle N/x \rangle (\lambda y \cdot P)) \equiv \lambda z \cdot \text{subst}(\langle N/z \rangle \text{subst}(\langle z/y \rangle P)),\)
  if \(y \neq x\) and \(y\) is free in \(N\) and \(x\) is free in \(P\)
  (where \(z\) is not free in \((N P)).\)

C.6.4. \(\alpha\)-Renaming and \(\beta\)-Reduction

- \(\alpha\)-renaming: \(\lambda x \cdot M\)
  If \(x, y\) are distinct variables then replacing \(x\) by \(y\) in \(\lambda x \cdot M\) results in \(\lambda y \cdot \text{subst}(\langle y/x \rangle M).\) We can rename the formal parameter of a \(\lambda\)-function expression provided that no free variables of its body \(M\) thereby become bound.
- \(\beta\)-reduction: \((\lambda x \cdot M)(N)\)
  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. \((\lambda x \cdot M)(N) \equiv \text{subst}(\langle N/x \rangle M)\)

C.6.5. Function Signatures

For sorts we may want to postulate some functions:
type
A, B, C
value
obs_B: A → B,
obs_C: A → C,
gen_A: BB × C → A

C.6.6. Function Definitions

Functions can be defined explicitly:

value
f: Arguments → Result
f(args) ≡ DValueExpr

g: Arguments → Result
f(args) ≡ ValueAndStateChangeClause
pre P(args)

Or functions can be defined implicitly:

value
f: Arguments → Result
f(args) as result
post P1(args,result)

g: Arguments → Result
g(args) as result
pre P2(args)
post P3(args,result)

The symbol ∼→ 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.

C.7. Other Applicative Expressions

C.7.1. Simple let Expressions

Simple (i.e., nonrecursive) let expressions:

let a = E_d in E_b(a) end

is an “expanded” form of:

(λa.E_b(a))(E_d)

C.7.2. Recursive let Expressions

Recursive let expressions are written as:

let f = λa:A • E(f) in B(f,a) end

is “the same” as:

let f = YF in B(f,a) end
where:
\[
F \equiv \lambda g \cdot \lambda a \cdot (E(g)) \text{ and } YF = F(YF)
\]

C.7.3. **Predicative let Expressions**

Predicative let expressions:

\[
\text{let } a:A \cdot P(a) \text{ in } B(a) \text{ end}
\]

express the selection of a value \(a\) of type \(A\) which satisfies a predicate \(P(a)\) for evaluation in the body \(B(a)\).

C.7.4. **Pattern and “Wild Card” let Expressions**

Patterns and wild cards can be used:

\[
\begin{align*}
\text{let } \{a\} \cup s = \text{set} \text{ in } ... \text{ end} \\
\text{let } \{a,\_\} \cup s = \text{set} \text{ in } ... \text{ end} \\
\text{let } (a,b,\_,\_,\_\_,c) = \text{cart} \text{ in } ... \text{ end} \\
\text{let } (a,\_,\_,\_,c) = \text{cart} \text{ in } ... \text{ end} \\
\text{let } (a)\hat{\ell} = \text{list} \text{ in } ... \text{ end} \\
\text{let } (a,\_,\_,\_,c)\hat{\ell} = \text{list} \text{ in } ... \text{ end} \\
\text{let } [a\mapsto bb] \cup m = \text{map} \text{ in } ... \text{ end} \\
\text{let } [a\mapsto b,\_] \cup m = \text{map} \text{ in } ... \text{ end}
\end{align*}
\]

C.7.5. **Conditionals**

Various kinds of conditional expressions are offered by RSL:

\[
\text{if } b_{\text{expr}} \text{ then } c_{\text{expr}} \text{ else } a_{\text{expr}} \text{ end}
\]

\[
\text{if } b_{\text{expr}} \text{ then } c_{\text{expr}} \text{ end} \equiv /\ast \text{ same as: } */
\]

\[
\text{if } b_{\text{expr}} \text{ then } c_{\text{expr}} \text{ else skip end}
\]

\[
\text{if } b_{\text{expr}}_1 \text{ then } c_{\text{expr}}_1 \\
\text{elsif } b_{\text{expr}}_2 \text{ then } c_{\text{expr}}_2 \\
\text{elsif } b_{\text{expr}}_3 \text{ then } c_{\text{expr}}_3 \\
... \\
\text{elsif } b_{\text{expr}}_n \text{ then } c_{\text{expr}}_n \text{ end}
\]

\[
\text{case } \text{expr} \text{ of} \\
\text{choice_pattern}_1 \rightarrow \text{expr}_1, \\
\text{choice_pattern}_2 \rightarrow \text{expr}_2, \\
... \\
\text{choice_pattern}_n \text{ or wild card } \rightarrow \text{expr}_n
\]

end

C.7.6. **Operator/Operand Expressions**

\[
(\text{Expr}) ::= 
\]
C.8. Imperative Constructs

C.8.1. Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative constructs. Imperative constructs are thus inevitable in RSL.

Unit value
stmt: Unit → Unit
stmt()

- Statements accept no arguments.
- Statement execution changes the state (of declared variables).
- Unit → Unit designates a function from states to states.
- Statements, stmt, denote state-to-state changing functions.
- Writing () as “only” arguments to a function “means” that () is an argument of type Unit.

C.8.2. Variables and Assignment

0. variable v:Type := expression
1. v := expr

C.8.3. Statement Sequences and skip

Sequencing is expressed using the ‘;’ operator. skip is the empty statement having no value or side-effect.

2. skip
3. stmt_1;stmt_2;...;stmt_n

C.8.4. Imperative Conditionals

4. if expr then stmt_c else stmt_a end
5. case e of: p_1→S_1(p_1),...,p_n→S_n(p_n) end

C.8.5. Iterative Conditionals

6. while expr do stmt end
7. do stmt until expr end
C.8.6. Iterative Sequencing

8. \texttt{for e in list\_expr • P(b) do S(b) end}

C.9. Process Constructs

C.9.1. Process Channels

Let A and B stand for two types of (channel) messages and i:\texttt{KIdx} for channel array indexes, then:

\begin{verbatim}
channel c:A
channel { k[i]:B • i:KIdx }
\end{verbatim}

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.9.2. 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 P() and Q stand for process expressions, then:

\begin{verbatim}
P || Q Parallel composition
P \parallel Q Nondeterministic external choice (either/or)
P \parallel Q Nondeterministic internal choice (either/or)
P –\parallel Q Interlock parallel composition
\end{verbatim}

express the parallel (\texttt{||}) of two processes, or the nondeterministic choice between two processes: either external (\texttt{\parallel}) or internal (\texttt{\parallel}). The interlock (–\texttt{\parallel}) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

C.9.3. Input/Output Events

Let c, k[i] and e designate channels of type A and B, then:

\begin{verbatim}
c ?, k[i] ? Input
c ! e, k[i] ! e Output
\end{verbatim}

expresses the willingness of a process to engage in an event that “reads” an input, respectively “writes” an output.

C.9.4. Process Definitions

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.

\begin{verbatim}
value
P: Unit \rightarrow in c out k[i] | Unit
Q: i:KIdx \rightarrow out c in k[i] | Unit
\end{verbatim}

\begin{verbatim}
P() ≡ ... c ? ... k[i] ! e ...
Q(i) ≡ ... k[i] ? ... c ! e ...
\end{verbatim}

The process function definitions (i.e., their bodies) express possible events.
C.10. Simple RSL Specifications

Often, we do not want to encapsulate small specifications in schemes, classes, and objects, as is often done in RSL. An RSL specification is simply a sequence of one or more types, values (including functions), variables, channels and axioms:

```
type
...
variable
...
channel
...
value
...
axiom
...
```

In practice a full specification repeats the above listings many times, once for each “module” (i.e., aspect, facet, view) of specification. Each of these modules may be “wrapped” into scheme, class or object definitions.\footnote{For schemes, classes and objects we refer to [6, Chap. 10]}