What are Documents ?

A Research Note[1]

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.

Yiu 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.

An appendix, C, summarises essentials of the RAISE [37] Specification Language, RSL [36].

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

[1] © Dines Bjørner, 2017
Correspondence and offprint requests to: Dines Bjørner, Fredsvej 11, DK 2840 Holte, Denmark

2.11 Inter-Behaviour Messages 11
2.11.1 Management Messages with Respect to the Archive 11
2.11.2 Management Messages with Respect to Handlers 12
2.11.3 Document Access Rights 12
2.11.4 Archive Messages with Respect to Management 12
2.11.5 Archive Message with Respect to Documents 12
2.11.6 Handler Messages with Respect to Documents 12
2.11.7 Handler Messages with Respect to Management 13
2.11.8 A Summary of Behaviour Interactions 13
2.12 A General Discussion of Handler and Document Interactions 13
2.13 Channels: A Final View 14
2.14 An Informal Summary of Behaviours 14
2.14.1 The Create Behaviour: Left Fig. 3 on Page 15 14
2.14.2 The Edit Behaviour: Right Fig. 3 on Page 15 14
2.14.3 The Read Behaviour: Left Fig. 4 on Page 15 14
2.14.4 The Copy Behaviour: Right Fig. 4 on Page 15 15
2.14.5 The Grant Behaviour: Left Fig. 5 on Page 16 16
2.14.6 The Shred Behaviour: Right Fig. 5 on Page 16 16
2.15 The Behaviour Actions 16
2.15.1 Management Behaviour 16
Management Create Behaviour: Left Fig. 3 on Page 15 17
Management Copy Behaviour: Right Fig. 4 on Page 15 17
Management Grant Behaviour: Left Fig. 5 on Page 16 18
Management Shred Behaviour: Right Fig. 5 on Page 16 19
2.15.2 Archive Behaviour 19
The Archive Create Behaviour: Left Fig. 3 on Page 15 19
The Archive Copy Behaviour: Right Fig. 4 on Page 15 20
The Archive Shred Behaviour: Right Fig. 5 on Page 16 20
2.15.3 Handler Behaviours 21
The Handler Create Behaviour: Left Fig. 3 on Page 15 21
The Handler Edit Behaviour: Right Fig. 3 on Page 15 21
The Handler Read Behaviour: Left Fig. 4 on Page 15 22
The Handler Copy Behaviour: Right Fig. 4 on Page 15 22
The Handler Grant Behaviour: Left Fig. 5 on Page 16 22
2.15.4 Document Behaviours 22
The Document Edit Behaviour: Right Fig. 3 on Page 15 23
The Document Read Behaviour: Left Fig. 4 on Page 15 23
The Document Shred Behaviour: Right Fig. 5 on Page 16 24
2.16 Conclusion 24
3 Bibliography
3.1 Bibliographical Notes 24
3.2 Domain Modeling Experiments 24
3.3 References 25
A Documents in Public Gornment
B Documents in Urban Planning
C RSL: The RAISE Specification Language – A Primer
C.1 Type Expressions 30
C.1.1 Atomic Types 30
C.1.2 Composite Types 30
Concrete Composite Types 30
Sorts and Observer Functions 31
C.2 Type Definitions 31
C.2.1 Concrete Types 31
C.2.2 Subtypes 32
C.2.3 Sorts — Abstract Types 32
C.3 The RSL Predicate Calculus 32
C.4 Propositional Expressions 32
C.4.1 Simple Predicate Expressions 32
C.4.2 Quantified Expressions 32
C.5 Concrete RSL Types: Values and Operations 33
C.5.1 Arithmetic 33
C.5.2 Set Expressions 33
Set Enumerations 33
Set Comprehension 33
C.5.3 Cartesian Expressions 33
Cartesian Enumerations 33
C.5.4 List Expressions 33

List Enumerations 33
List Comprehension 34
C.5.5 Map Expressions 34
Map Enumerations 34
Map Comprehension 34
C.5.6 Set Operations 34
Set Operator Signatures 34
Set Examples 35
Informal Explication 35
Set Operator Definitions 35
C.5.7 Cartesian Operations 36
C.5.8 List Operations 36
List Operator Signatures 36
List Operation Examples 36
Informal Explication 36
List Operator Definitions 37
C.5.9 Map Operations 37
Map Operator Signatures and Map Operation Examples 37
Map Operation Explication 38
Map Operation Redefinitions 38
C.6 λ-Calculus + Functions 39
C.6.1 The λ-Calculus Syntax 39
C.6.2 Free and Bound Variables 39
C.6.3 Substitution 39
C.6.4 α-Renaming and β-Reduction 39
C.6.5 Function Signatures 39
C.6.6 Function Definitions 40
C.7 Other Applicative Expressions 40
C.7.1 Simple let Expressions 40
C.7.2 Recursive let Expressions 40
C.7.3 Predicative let Expressions 41
C.7.4 Pattern and "Wild Card" let Expressions 41
C.7.5 Conditionals 41
C.7.6 Operator/Operand Expressions 41
C.8 Imperative Constructs 42
C.8.1 Statements and State Changes 42
C.8.2 Variables and Assignment 42
C.8.3 Statement Sequences and skip 42
C.8.4 Imperative Conditionals 42
C.8.5 Iterative Conditionals 42
C.8.6 Iterative Sequencing 43
C.9 Process Constructs 43
C.9.1 Process Channels 43
C.9.2 Process Composition 43
C.9.3 Input/Output Events 43
C.9.4 Process Definitions 43
C.10 Simple RSL Specifications 44

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 &[2] 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, and by **documents** ? We give an ultra short survey. The survey relies on your prior knowledge of what you think documents are! **Management** decides[3] to direct **handler**s to work on **document**s. **Management** first directs the document archive to **create document**s. The document **archive creates document**s, as requested by **management**, and informs management of the **unique document identifiers** (by means of which handlers can handle these documents). **Management** then **grant**s its designated **handler**(s) **access rights** to **document**s, these access rights enable handlers to **edit, read** and **copy** documents. The **handlers**' **edit**ing and **read**ing of **document**s is accomplished by the **handlers** "working directly" with the **documents** (i.e., synchronising and communicating with **document behaviour**s). The **handlers**' **copy**ing of **document**s is accomplished by the **handler**s 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).

```
type
1  S, M, A
value
1  obs_M: S → M
1  obs_A: S → A
type
2  H, Hs = H-set
3  D, Ds = D-set
value
2  obs_Hs: M → Hs
3  obs_Ds: A → Ds
```

2.3. Unique Identifiers

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

4 We associate unique identifiers with aggregate, handler and document endurants.

5 These can be observed from respective parts[4].

```
type
4  MI[5], AI[6], HI, DI
value
5  uid_MI[7]: M → MI
5  uid_AI[8]: A → AI
```

[2] 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.

5 uid_HI: H → HI

5 uid_DI: D → DI

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 **create**d 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.
[9] From: https://en.wikipedia.org/wiki/Document
[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 **create**d, **edit**ed, **read**, **copied** and **shred**ed. 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.

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
type
7 $\mathsf{DA} = \mathsf{DN}^*$
8 $\mathsf{DH} = (\mathsf{TIME} \times \mathsf{DO})^*$
9 DO == Crea | Edit | Read | Copy | Shre
10 $\mathsf{Crea} :: (\mathsf{HI} \times \mathsf{TIME}) \times (\mathsf{DI}\text{-}\mathbf{set} \times \mathsf{Info}) \times \mathsf{DN} \times \{|''\texttt{empty\_DC}''|\}$
10bi Info = ...
value
10bii embed_DIs_in_DD: DI-**set** × Info → DD
axiom
10d $''\texttt{empty\_DC}'' \in \mathsf{DC}$
type
11 Edit :: (HI × TIME) × (EDIT × UNDO)
12 Read :: (HI × TIME) × DI

```
13  Copy :: (HI × TIME) × DI × DN
14  Shre :: (HI × TIME) × DI
15  EDIT = (DA × DN × DC) → (DA × DC)
16  UNDO = (DA × DC) → (DA × DN × DC)
axiom
17  ∀ mkEdit(_,(e,u)):Edit •
17      ∀ (da,dn,dc):(DA×DN×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,

  the (distributed) parallel composition of

  22 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(...)|i:HI•i∈his}
22  ‖ ‖{docu_i(dd)(da,dc,dh)|i:DI•i∈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).

24 The management behaviour, mgtm, need to (synchronise and) communicate with the archive behaviour, arch, in order, for the management behaviour, to request the archive behaviour

- to **create** (ab initio or due to **copy**ing)
- or **shred** document behaviours, 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 mgtm_arch_ch:MA

25 The management behaviour, mgtm, need to (synchronise and) communicate with all handler behaviours, hdlr_i and they, in turn, to (synchronised) communicate with the handler management behaviour, 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 {mgtm_hdlr_ch[i]:MH|i:HI•i ∈ his}

26 The document archive behaviour, arch, need (synchronise and) communicate with all document behaviours, docu_j and they, in turn, to (synchronise and) communicate with the archive behaviour, arch.

channel
26 {arch_docu_ch[j]:AD|h:DI•j ∈ dis}

27 Handler behaviours, hdlr_i, need (synchronise and) communicate with all the document behaviours, docu_j, with which it has operational allowance to so do so[11], and document behaviours, docu_j, need (synchronise and) communicate with potentially all handler behaviours, hdlr_i, namely those handler behaviours, hdlr_i with which they have ("earlier" synchronised and) communicated.

channel
27 {hdlr_docu_ch[i,j]:HD|i:HI,j:DI•i ∈ his∧j ∈ dis}

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

type
28 MA, MH, AD, HD

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, n_h, of one or more handler behaviours, and a variable, initially zero number of document behaviours, with a maximum of these being n_d. The figure also indicates, again rather informally, the channels between these behaviours: one channel between the management and the archive behaviours; n_h channels (n_h is, again, informally indicated) between the management behaviour and the n_h handler behaviours; n_d channels (n_d is, again, informally indicated) between the archive behaviour and the n_d document behaviours; and $n_h \times n_d$ channels ($n_d \times n_d$ is, again, informally indicated) between the n_h handler behaviours and the n_d document behaviours

[11] The notion of operational allowance will be explained below.

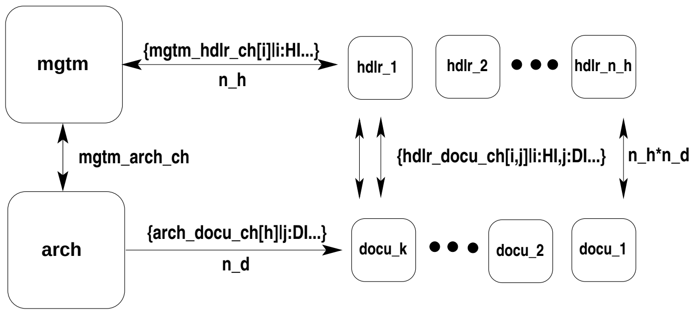
mgtm
{mgtm_hdlr_ch[i]|i:HI...}
n_h
hdlr_1
hdlr_2
hdlr_n_h
mgtm_arch_ch
{hdlr_docu_ch[i,j]|i:HI,j:DI...}
n_h*n_d
arch
{arch_docu_ch[h]|j:DI...}
n_d
docu_k
docu_2
docu_1

Fig. 1. An Informal Snapshot of System Behaviours

2.8. Behaviour Signatures

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

30 The archive 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 archive behaviour and that they [regularly] interacts with handler behaviours.

value
29 mgtm: ... → **in**,**out** mgtm_arch_ch, {mgtm_hdlr_ch[i]|i:HI•i ∈ his} **Unit**
30 arch: ... → **in**,**out** mgtm_arch_ch, {arch_docu_ch[j]|j:DI•j ∈ dis} **Unit**
31 hdlr_i: ... → **in** mgtm_hdlr_ch[i], **in**,**out** {hdlr_docu_ch[i,j]|j:DI•j∈dis} **Unit**
32 docu_j: ... → **in** mgtm_arch_ch, **in**,**out** {hdlr_docu_ch[i,j]|i:HI•i ∈ his} **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.

type
33 TIME
34 TIME_INTERVAL
value
35 add: TIME_INTERVAL × TIME → TIME
35 sub: TIME × TIME → TIME_INTERVAL
35 mpy: TIM_INTERVALE × **Nat** → TIME_INTERVAL
36 <,≤,=,≠,≥,>: ((TIME×TIME)|(TIME_INTERVAL×TIME_INTERVAL)) → **Bool**

2.9.2. A Time Behaviour and a Time Channel

37 We postulate a[n "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+ti, or – internally non-deterministically – chooses to [first] offer its time on a [global] channel, time_ch, then resumes being a time behaviour with unchanged time., t

38 The time interval increment, ti, 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, time_ch, on which the time behaviour offers time values to whoever so requests.

```
value
37  time: TIME → time_ch  TIME  Unit
37  time(t) ≡ (time(t) ⌈⌉ time(t+ti) ⌈⌉ time_ch!t ; time(t))
38  ti:TIME_INTERVAL ...
channel
39  time_ch: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:

- **wait** ti.

wait is a keyword; ti designates a time interval. A typical use of the wait construct is:

- ... *ptA* ; **wait** ti; *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*+ti.

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 mgtm (management), arch (archive), hdlrs (handlers), and docus (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.[12] 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 β we shall show the static attributes as one set of parameters and the programmable attributes as another set of parameters.

value β: Static → Program → ... **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.

[12] 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 avaliable (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
40 $\mathsf{MDIR} = \mathsf{HI} \overrightarrow{m} (\mathsf{DI} \overrightarrow{m} \mathsf{ANm}\text{-}\mathbf{set})$
41 $\mathsf{ADIR} = \mathsf{avail}:\mathsf{DI}\text{-}\mathbf{set} \times \mathsf{used}:\mathsf{DI}\text{-}\mathbf{set} \times \mathsf{gone}:\mathsf{DI}\text{-}\mathbf{set}$
42 $\mathsf{HDIR} = \mathsf{DI} \overrightarrow{m} \mathsf{ANm}\text{-}\mathbf{set}$
43 $\mathsf{SDATR} = \mathsf{DD},\ \mathsf{PDATR} = \mathsf{DA} \times \mathsf{DC} \times \mathsf{DH}$
axiom
41 $\forall\ (\mathsf{avail},\mathsf{used},\mathsf{gone}):\mathsf{ADIR} \bullet \mathsf{avail} \cap \mathsf{used} = \{\} \wedge \mathsf{gone} \subseteq \mathsf{used}$

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

value
29 $\mathsf{mgtm}:\ \mathsf{MDIR} \rightarrow \mathbf{in},\mathbf{out}\ \mathsf{mgtm\_arch\_ch},\ \{\mathsf{mgtm\_hdlr\_ch}[\,i\,]|i:\mathsf{HI}\bullet i \in \mathsf{his}\}\ \mathbf{Unit}$
30 $\mathsf{arch}:\ \mathsf{ADIR} \rightarrow \mathbf{in},\mathbf{out}\ \mathsf{mgtm\_arch\_ch},\ \{\mathsf{arch\_docu\_ch}[\,j\,]|j:\mathsf{DI}\bullet j \in \mathsf{dis}\}\ \mathbf{Unit}$
31 $\mathsf{hdlr}_i:\ \mathsf{HDIR} \rightarrow \mathbf{in}\ \mathsf{mgtm\_hdlr\_ch}[\,i\,],\ \mathbf{in},\mathbf{out}\ \{\mathsf{hdlr\_docu\_ch}[\,i,j\,]|j:\mathsf{DI}\bullet j \in \mathsf{dis}\}\ \mathbf{Unit}$
32 $\mathsf{docu}_j:\ \mathsf{SDATR} \rightarrow \mathsf{PDATR} \rightarrow \mathbf{in}\ \mathsf{mgtm\_arch\_ch},\ \mathbf{in},\mathbf{out}\ \{\mathsf{hdlr\_docu\_ch}[\,i,j\,]|i:\mathsf{HI}\bullet i \in \mathsf{his}\}\ \mathbf{Unit}$

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 wich 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
10. $\mathsf{Crea} :: (\mathsf{HI} \times \mathsf{TIME}) \times (\mathsf{DI}\text{-}\mathbf{set} \times \mathsf{Info}) \times \mathsf{DN} \times \{|''\mathtt{empty\_DC}''|\}$ [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 $\mathsf{Copy} :: \mathsf{DI} \times \mathsf{HI} \times \mathsf{TIME} \times \mathsf{DN}$ [cf. Item 55 on Page 13]

46 Management **schred**s documents by informing the archive behaviour to do so.

type
46 $\mathsf{Shred} :: \mathsf{TIME} \times \mathsf{DI}$

2.11.2. Management Messages with Respect to Handlers

47 Upon receiving, from the archive behaviour, the "feedback" the identifier of the created document (behaviour):

type
47. Create_Reply :: NewDocID(di:DI)

48 the management behaviour decides to **grant** access rights, acrs:ACRS[13], to a document handler, hi:HI.

type
48 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.

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

type
49 ACRS = ANm-**set**
50 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.

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

type
51 NewDocID :: DI

2.11.5. Archive Message with Respect to Documents

52 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.

type
52 Shred :: {|″`stop`″|}

2.11.6. Handler Messages with Respect to Documents

Handlers, generically referred to by hdlr_i, may perform the following operations on documents: **edit, read** and **copy**. (Management, via the archive behaviour, **create**s and **shred**s documents.)

53 To perform an **edit** action handler hdlr_i must provide the following:

- the document identity – in the form of a (i:HI,j:DI) channel hdlr_docu_ch 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.

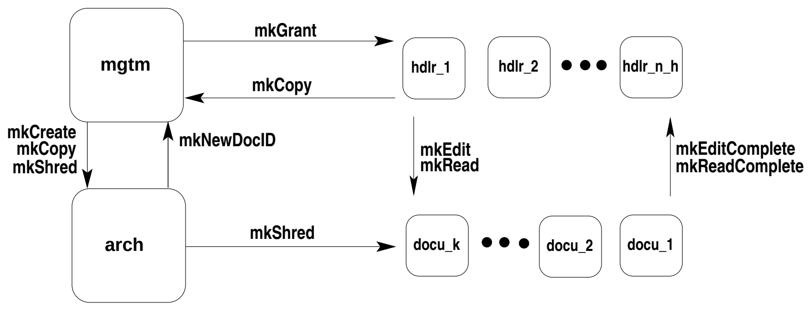

Fig. 2. A Summary of Behaviour Interactions

type
53 Edit :: DI × HI × TIME × (EDIT × UNDO)

54 To perform a **read** action handler hdlr_i must provide the following information:

- the document identity – in the form of a di:DI channel hdlr_docu_ch index value,
- the handler identity and
- the time of the read request.

type
54 Read :: DI × HI × TIME

2.11.7. Handler Messages with Respect to Management

55 To perform a **copy** action, a handler, hdlr_i, must provide the following information to the management behaviour, mgtm:

- the document identity,
- the handler identity – in the form of an hi:HI channel mgtm_hdlr_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 × HI × TIME × 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.

type
24 MA = Create (Item 44 on Page 11) | Shred (Item 44d on Page 11) | NewDocID (Item 51 on Page 12)
25 MH = Grant (Item 44c on Page 11) | Copy (Item 55 on the preceding page) |
26 AD = Shred (Item 52 on Page 12)
27 HD = Edit (Item 53 on Page 12) | Read (Item 54 on the previous page) | Copy (Item 55 on the preceding page)

2.14. An Informal Summary of Behaviours

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

56 [1] The management behaviour, at its own volition, initiates a create document behaviour. It does so by offering a create document message to the archive behaviour.
- a [1.1] That message contains a meaningful document descriptor,
- b [1.2] an initial document annotation,
- c [1.3] an "empty" document contents and
- d [1.4] a single element document history.

(We refer to Sect. 2.11.1 on Page 11, Items 44–**??**.)

57 [2] The archive behaviour offers to accept that management message. It then selects an available document identifier (here shown as k), henceforth marking k as used.

58 [3] The archive behaviour then "spawns off" document behaviour docu_k – here shown by the "dash–dotted" rounded edge square.

59 [4] The archive behaviour then offers the document identifier k message to the management behaviour. (We refer to Sect. 2.11.4 on Page 12, Item 51.)

60 [5] The management behaviour then
- a [5.1] selects a handler, here shown as i, i.e., hdlr_i,
- b [5.2] records that that handler is granted certain access rights to document k,
- c [5.3] and offers that granting to handler behaviour i.

(We refer to Sect. 2.11.2 on Page 12, Item 48 on Page 12.)

61 [6] Handler behaviour i records that it now has certain access rights to doccument i.

2.14.2. The Edit Behaviour: Right Fig. 3 on the facing 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 a(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 ti.

3 Handler behaviour i completes its edit action.

2.14.3. The Read Behaviour: Left Fig. 4 on the next 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 a(annotation) note.

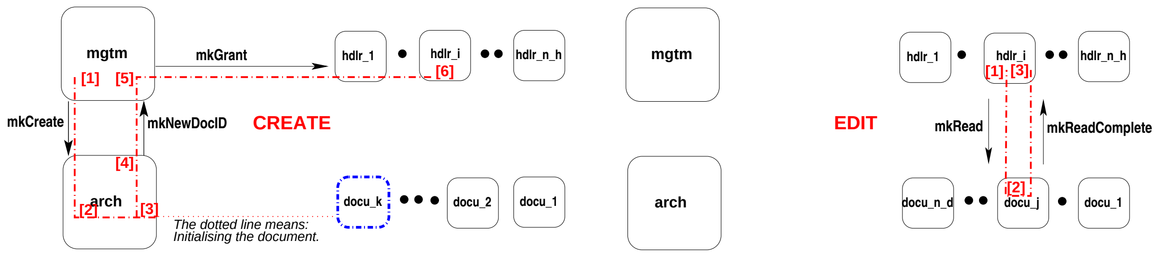

Fig. 3. Informal Snapshots of Create and Edit Document Behaviours

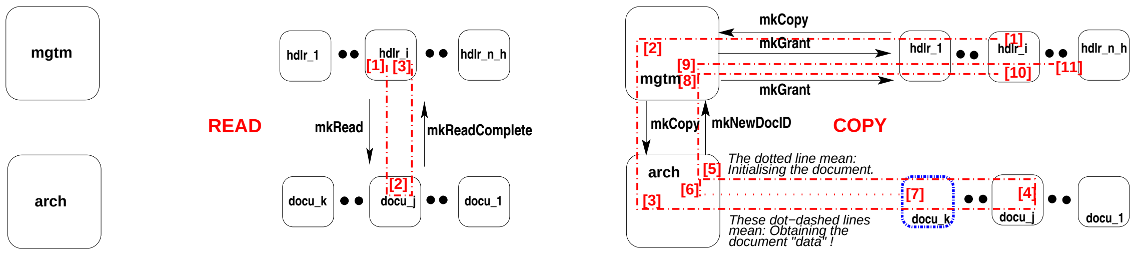
mgtm
arch
READ
hdlr_1
hdlr_i
hdlr_n_h
mkRead
mkReadComplete
docu_k
docu_j
docu_1
mkCopy
mkGrant
mkNewDocID
COPY
The dotted line mean: Initialising the document.
These dot–dashed lines mean: Obtaining the document "data" !

Fig. 4. Informal Snapshots of Read and Copy Document Behaviours

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 ti.

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 *d*ocument *d*escriptor, dd_j, its *d*ocument *a*nnotations, da_j, its *d*ocument *c*ontents, 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 *d*ocument *d*escriptor, dd_k, its *d*ocument *a*nnotations, da_k, its *d*ocument *c*ontents, 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,

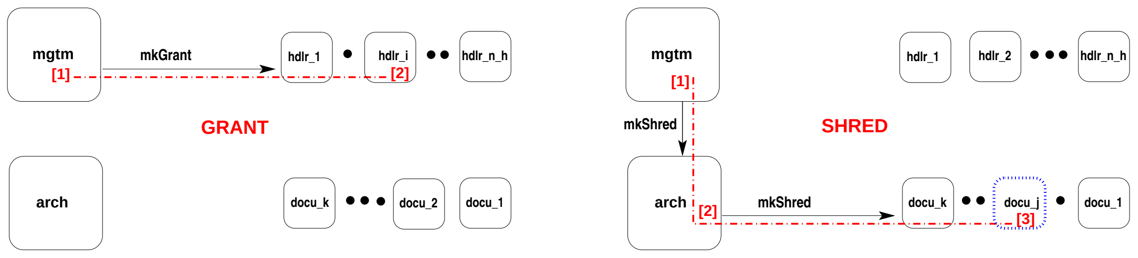

Fig. 5. Informal Snapshots of Grant and Shred Document Behaviours

10 while also informing the master copy of the copy identity (etcetera).
11 The handlers granted access to the copy record this fact.

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 Item [2] correspond, in essence, to Item [11] of the copy behaviour.

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 **stop**s (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 β is involved in τ traces, $\tau_1, \tau_2, ..., \tau_\tau$, then behaviour β shall be defined in terms of τ non-deterministic alternative behaviours named $\beta_{\tau_1}, \beta_{\tau_2}, ..., \beta_{\tau_\tau}$.

2.15.1. Management Behaviour

62 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   mgtm: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
62   mgtm(mdir) ≡
62a        mgtm_create(mdir)
62b     ⌈⌉ mgtm_copy(mdir)
62c     ⌈⌉ mgtm_grant(mdir)
62d     ⌈⌉ mgtm_shred(mdir)
```

Management Create Behaviour: Left Fig. 3 on Page 15

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

```
value
63  mgtm_create: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
63  mgtm_create(mdir) ≡
64  [1]   let hi = mgtm_create_initiation(mdir) ;          [Left Fig. 3 on Page 15]
65  [5]   mgtm_create_awaits_response(mdir)(hi) end  [Left Fig. 3 on Page 15]
```

The **management create initiation** behaviour

66 selects a handler on behalf of which it requests the document creation,
67 assembles the elements of the create message:

- by embedding a set of zero or more document references, dis, with some information, info, into a document descriptor, adding
- a document note, dn, and
- and initial, that is, empty document contents, "empty_DC",

68 offers such a create document message to the archive behaviour, and
69 yields the identifier of the chosen handler.

```
value
64  mgtm_create_initiation: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} HI
64  mgtm_create_initiation(mdir) ≡
66              let hi:HI • hi ∈ dom mdir,
67  [1.2−.4]        (dis,info):(DI-set×Info),dn:DN • is_meaningful(embed_DIs_in_DD(dis,info))(mdir) in
68  [1.1]         mgtm_arch_ch ! mkCreate(embed_DIs_in_DD(ds,info),dn,"empty_DC")
69                hi end

67  is_meaningful: DD → MDIR → Bool [left further undefined]
```

The **management create awaits response** behaviour

70 starts by awaiting a reply from the archive behaviour with the identity, *di*, of the document (that that behaviour has created).
71 It then selects suitable access rights,
72 with which it updates its handler/document directory
73 and offers to the chosen handler
74 whereupon it resumes, with the updated management directory, being the management behaviour.

```
value
65  mgtm_create_awaits_response: MDIR → HI → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
65  mgtm_create_awaits_response(mdir) ≡
70  [5]        let mkNewDocID(di) = mgtm_arch_ch ? in
71  [5.1]      let acrs:ANm-set in
72  [5.2]      let mdir′ = mdir † [hi ↦ [di ↦ acrs]] in
73  [5.3]      mgtm_hdlr_ch[hi] ! mkGrant(di,acrs)
74             mgtm(mdir′) end end end
```

Management Copy Behaviour: Right Fig. 4 on Page 15

75 The **management copy** behaviour
76 accepts a copy document request from a handler behaviour (i.e., a request to the archive behaviour),

77 and then awaits a response from the archive behaviour;
78 after which it grants access rights to handlers to the document copy.

```
value
75   mgtm_copy: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
75   mgtm_copy(mdir) ≡
76   [2]  let hi = mgtm_accept_copy_request(mdir) in
77   [8]  let di = mgtm_awaits_copy_response(mdir)(hi) in
78   [9]  mgtm_grant_access_rights(mdir)(di) end end
```

79 The **management accept copy** behaviour non-deterministically externally ([]) awaits a copy request from a[ny] handler (i) 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 –
82 while yielding the identity of the requesting handler.

```
79.  mgtm_accept_copy_request: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} HI
79.  mgtm_accept_copy_request(mdir) ≡
80.       let mkCopy(di,hi,t,dn) = []{mgtm_hdlr_ch[i]?|i:HI•i ∈ his} in
81.       mgtm_arch_ch ! mkCopy(di,hi,t,dn) ;
81.       hi end
```

The **management awaits copy response** behaviour

83 awaits a reply from the archive behaviour as to the identity of the newly created copy (di) of master document j.
84 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) –
85 while yielding the identity, di, of the newly created copy.

```
62b.   mgtm_awaits_copy_response: MDIR → HI → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} DI
62b.   mgtm_awaits_copy_response(mdir)(hi) ≡
83.   [8]  let mkNewDocID(di) = mgtm_arch_ch ? in
84.        mgtm_hdlr_ch[hi] ! mkCopy(di) ;
85.        di end
```

The **management grants access rights** behaviour

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

```
78.  mgtm_grant_access_rights: MDIR → DI → in,out {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
78.  mgtm_grant_access_rights(mdir)(di) ≡
86.       let diarm = [hi↦acrs|hi:HI,arcs:ANm-set• hi ∈ dom mdir∧arcs⊆(diarm(hi))(di)] in
87.       ∥ {mgtm_hdlr_ch[hi]!mkGrant(hi,time_ch?,di,acrs) |
87.            hi:HI,acrs:ANm-set•hi ∈ dom diarm∧acrs⊆(diarm(hi))(di)} end
```

Management Grant Behaviour: Left Fig. 5 on Page 16 The **management grant** behaviour

88 is a variant of the mgtm_grant_access_rights function, Items 86–87.
89 The management behaviour selects a suitable subset of known handler identifiers, and
90 for these a suitable subset of document identifiers from which
91 it then constructs a map from handler identifiers to subsets of access rights.
92 With this the management behaviour then issues appropriate grants to the chosen handlers.

```
type
     MDIR = HI →m (DI →m ANm-set)
value
88  mgtm_grant: MDIR → in,out {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
88  mgtm_grant(mdir) ≡
89      let his ⊆ dom dir in
90      let dis ⊆ ∪{dom mdir(hi)|hi:HI•hi ∈ his} in
91      let diarm = [hi↦acrs|hi:HI,di:DI,acrs:ANm-set• hi ∈ his∧di ∈ dis∧acrs⊆(diarm(hi))(di)] in
92      ‖{mgtm_hdlr_ch[hi]!mkGrant(di,acrs) |
92        hi:HI,di:DI,acrs:ANm-set•hi ∈ dom diarm∧di ∈ dis∧acrs⊆(diarm(hi))(di)}
88      end end end
```

Management Shred Behaviour: Right Fig. 5 on Page 16 The **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  mgtm_shred: MDIR → out mgtm_arch_ch  Unit
93  mgtm_shred(mdir) ≡
94        let di:DI • is_suitable(di)(mdir) in
95  [1]   mgtm_arch_ch ! mkShred(time_ch?,di) ;
96        let mdir′ = [hi↦mdir(hi)\{di}|hi:HI•hi ∈ dom mdir] in
97        mgtm(mdir′) 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

```
type
41   ADIR = avail:DI-set × used:DI-set × gone:DI-set
axiom
41   ∀ (avail,used,gone):ADIR • avail ∩ used = {} ∧ gone ⊆ used
value
98   arch: ADIR → in,out mgmt_arch_ch, {arch_docu_ch[di]|di:DI•di ∈ dis} Unit
98a  arch(adir) ≡
98a          arch_create(adir)
98b        ⌈⌉ arch_copy(adir)
98c        ⌈⌉ arch_shred(adir)
```

The Archive Create Behaviour: Left Fig. 3 on Page 15 The **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, dd, the initial document annotation being the singleton list of the note, an, and the initial document contents, 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
101.   [4]  mgmt_arch_ch ! mkNewDocID(di) ;
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) = mgtm_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.          mgtm_arch_ch ! mkCopyDocID(di) ;
109.  [6,7] docu_di(augment(dd,″copy″,j,hi),augment(da,″copy″,hi),dc,augment(dh,(″copy″,date_and_time,j,hi)))
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

114 The handler behaviour is involved in the following action traces:

a **create**	Fig. 3 on Page 15 Left
b **edit**	Fig. 3 on Page 15 Right
c **read**	Fig. 4 on Page 15 Left
d **copy**	Fig. 4 on Page 15 Right
e **grant**	Fig. 5 on Page 16 Left

```
value
114   hdlr_hi: HATTRS → in,out mgtm_hdlr_ch[hi],{hdlr_docu_ch[hi,di]|di:DI•di∈dis} Unit
114   hdlr_hi(hattrs) ≡
114a        hdlr_create_hi(hattrs)
114b      ⌈⌉ hdlr_edit_hi(hattrs)
114c      ⌈⌉ hdlr_read_hi(hattrs)
114d      ⌈⌉ hdlr_copy_hi(hattrs)
114e      ⌈⌉ hdlr_grant_hi(hattrs)
```

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

115 The **handler create** behaviour offers to accept the granting of access rights, acrs, to document di.

116 It according updates its programmable hattrs attribute;

117 and resumes being a handler behaviour with that update.

```
114a  hdlr_create_hi: HATTRS × HHIST → in,out mgtm_hdlr_ch[hi] Unit
114a  hdlr_create_hi(hattrs,hhist) ≡
115     let mkGrant(di,acrs) = mgtm_hdlr_ch[hi] ? in
116     let hattrs′ = hattrs † [hi ↦ acrs] in
117     hdlr_create_hi(hattrs′,augment(hhist,mkGrant(di,acrs))) end end
```

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

118 The handler behaviour, on its own volition, decides to edit a document, *di*, for which it has editing rights.

119 The handler behaviour selects a suitable (...) pair of *e*dit/*u*ndo functions and a suitable (annotation) note.

120 It then communicates the desire to edit document *di* with (*e*,*u*) (at time *t*=time_ch?).

121 Editing take some time, *ti*.

122 We can therefore assert that the time at which editing has completed is *t*+*ti*.

123 The handler behaviour accepts the edit completion message from the document handler.

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

```
114b  hdlr_edit_hi: HATTRS × HHIST → in,out {hdlr_docu_ch[hi,di]|di:DI•di∈dis} Unit
114b  hdlr_edit_hi(hattrs,hhist) ≡
118  [1]  let di:DI • di ∈ dom hattrs ∧ ″edit″ ∈ hattrs(di) in
119  [1]  let (e,u):(EDIT×UNDO) • ... , n:AN • ... in
120  [1]  hdlr_docu_ch[hi,di] ! mkEdit(hi,t=time_ch?,e,u,n) ;
121  [2]  let ti:TIME_INTERVAL • ... in
122  [2]  wait ti ; assert: time_ch? = t+ti
123  [3]  let mkEditComplete(ti′,...) = hdlr_docu_ch[hi,di] ? in assert ti′ ≅ ti
124       hdlr_hi(hattrs,augment(hhist,(di,mkEdit(hi,t,ti,e,u))))
114b      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*=time_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.

```
114c  hdlr_edit_hi: HATTRS × HHIST → in,out {hdlr_docu_ch[hi,di]|di:DI•di∈dis} Unit
114c  hdlr_edit_hi(hattrs,hhist) ≡
125  [1]   let di:DI • di ∈ dom hattrs ∧ ″read″ ∈ hattrs(di), n:N • ... in
126  [1]   hdlr_docu_ch[hi,di] ! mkRead(hi,t=time_ch?,n) ;
127  [2]   let ti:TIME_INTERVAL • ... in
128  [2]   wait ti ; assert: time_ch? = t+ti
129  [3]   let mkReadComplete(ti,...) = hdlr_docu_ch[hi,di] ? in
130          hdlr_hi(hattrs,augment(hhist,(di,mkRead(di,t,ti))))
114c         end end end
```

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.

```
114d    hdlr_copy_hi: HATTRS × HHIST → in,out mgtm_hdlr_ch[hi] Unit
114d    hdlr_copy_hi(hattrs,hhist) ≡
131      [1] let di:DI • di ∈ dom hattrs ∧ ″copy″ ∈ hattrs(di) in
132      [1] mgtm_hdlr_ch[hi] ! mkCopy(di,hi,t=time_ch?) ;
133     [10] let mkCopyComplete(di′,di) = mgtm_hdlr_ch[hi] ? in
134     [10] hdlr_hi(hattrs,augment(hhist,time_ch?,(mkCopy(di,hi,,t),mkCopyComplete(di′))))
114d          end end
```

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

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

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

```
114e    hdlr_grant_hi: HATTRS × HHIST → in,out mgtm_hdlr_ch[hi] Unit
114e    hdlr_grant_hi(hattrs,hhist) ≡
135     [2] let mkGrant(di,acrs) = mgtm_hdlr_ch[hi] ? in
136     [2] hdlr_hi(hattrs†[di↦acrs],augment(hhist,time_ch?,mkGrant(di,acrs)))
114e        end
```

2.15.4. Document Behaviours

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

a **edit** Fig. 3 on Page 15 Right

b **read** Fig. 4 on Page 15 Left
c **shred** Fig. 5 on Page 16 Right

```
value
137  docu_{di}: DD × (DA × DC × DH) → in,out arch_docu_ch[di], {hdlr_docu_ch[hi,di]|hi:HI•hi∈his} Unit
137  docu_{di}(dattrs) ≡
137a          docu_edit_{di}(dd)(da,dc,dh)
137b       ⌈⌉ docu_read_{di}(dd)(da,dc,dh)
137c       ⌈⌉ docu_shred_{di}(dd)(da,dc,dh)
```

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 ti, 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, ti, to do.

140 The handler behaviour is notified, mkEditComplete(...) of the completion of the edit, and

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

```
value
137a  docu_edit_{di}: DD × (DA × DC × DH) → in,out {hdlr_docu_ch[hi,di]|hi:HI•hi∈his} Unit
137a  docu_edit_{di}(dd)(da,dc,dh) ≡
138    [2] let mkEdit(hi,t,e,u,n) = ⌈⌉{hdlr_docu_ch[hi,di]?|hi:HI•hi∈his} in
138a    [2] let dc′ = e(dc),
138b           da′ = augment(da,((hi,t),(″edit″,e,u),n)),
138c           dh′ = augment(dh,((hi,t),(″edit″,e,u))) in
139        let ti = time_ch? − t in
140        hdlr_docu_ch[hi,di] ! mkEditComplete(ti,...) ;
141        docu_{di}(dd)(da′,dc′,dh′)
137a       end end end
```

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

142 The 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.

```
value
137b  docu_read_{di}: DD × (DA × DC × DH) → in,out {hdlr_docu_ch[hi,di]|hi:HI•hi∈his} Unit
137b  docu_read_{di}(dd)(da,dc,dh) ≡
142   [2] let mkRead(hi,t,n) = {hdlr_docu_ch[hi,di]?|hi:HI•hi∈his} in
143   [2] let ti:TIME_INTERVAL • ... in
143   [2] wait ti ;
144   [2] hdlr_docu_ch[hi,di] ! mkReadComplete(ti,...) ;
145   [2] docu_{di}(dd)(augment(da,n),dc,augment(dh,(hi,t,ti,″read″)))
137b      end end
```

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 **stop**s!

```
value
137c   docu_shred$_{di}$: DD × (DA × DC × DH) → in,out arch_docu_ch[di] Unit
137c   docu_shred$_{di}$(dd)(da,dc,dh) ≡
146   [3]  let mkShred(...) = arch_docu_ch[di] ? in
147          stop
137c  [3]  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 triptych 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

- Credit Card System[14], [25] 2016. Result of my PhD lectures at Uppsala, May 2016
- Weather Information Systems[15] [27] Result of my PhD lectures at Bergen, November 2016
- Documents[16] [18] 2013.
- Transport Systems[17] [21] 2010.

[14] http://www.imm.dtu.dk/ dibj/2016/uppsala/accs.pdf
[15] http://www.imm.dtu.dk/ dibj/2016/wis/wis-p.pdf
[16] http://www.imm.dtu.dk/˜dibj/doc-p.pdf
[17] http://www.imm.dtu.dk/˜dibj/comet/comet1.pdf

- The Tokyo Stock Exchange Trading Rules[18] and[19] [29] 2010.
- On Development of Web-based Software[20] 2010.
- What is Logistics ?[21] [12] 2009.
- Pipelines – a Domain Description[22] and[23], [20] 2009.
- Platooning[24],
- A Container Line Industry Domain[25], [8] 2007
- Models of IT Security: Security Rules & Regulations[26] [30] 2006.
- Markets[27] [3]
- **Railway Systems Descriptions: 1996–2003**
  - ∞ Dines Bjørner: Formal Software Techniques in Railway Systems[28] [2]
  - ∞ Chris George, Dines Bjørner and Søren Prehn: Scheduling and Rescheduling of Trains[29], [34] 1996
  - ∞ Dines Bjørner: A Railway Systems Domain[30] An "old" UNU-IIST report, 1997
  - ∞ Dines Bjørner: Formal Software Techniques in Railway Systems[31], 2002
  - ∞ Albena Strupchanska, Martin Penicka and Dines Bjørner: Railway Staff Rostering[32], 2003 [42]
  - ∞ Dines Bjørner: Dynamics of Railway Nets[33], 2003 [5]
  - ∞ Martin Penicka, Albena Strupchanska and Dines Bjørner: Train Maintenance Routing[34], 2003 [41]
  - ∞ Panagiotis Karras and Dines Bjørner: Train Composition and Decomposition: Domain and Requirements[35], 2003
  - ∞ Dines Bjørner: Dynamics of Railway Nets: On an Interface between Automatic Control and Software Engineering[36] [5] 2003

3.3. References

[1] Dines Bjørner. Michael Jackson's Problem Frames: Domains, Requirements and Design. In Li ShaoYang and Michael Hinchley, editors, *ICFEM'97: International Conference on Formal Engineering Methods*, Los Alamitos, November 12–14 1997. IEEE Computer Society. Final Version.

[2] Dines Bjørner. Formal Software Techniques in Railway Systems. In Eckehard Schnieder, editor, *9th IFAC Symposium on Control in Transportation Systems*, pages 1–12, Technical University, Braunschweig, Germany, 13–15 June 2000. VDI/VDE-Gesellschaft Mess– und Automatisieringstechnik, VDI-Gesellschaft für Fahrzeug– und Verkehrstechnik. Invited talk.

[3] Dines Bjørner. Domain Models of "The Market" — in Preparation for E–Transaction Systems. In *Practical Foundations of Business and System Specifications (Eds.: Haim Kilov and Ken Baclawski)*, The Netherlands, December 2002. Kluwer Academic Press. Final draft version.

[4] Dines Bjørner. Domain Engineering: A "Radical Innovation" for Systems and Software Engineering ? In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, Heidelberg, October 7–11 2003. Springer–Verlag. The Zohar Manna International Conference, Taormina, Sicily 29 June – 4 July 2003. .

[5] Dines Bjørner. Dynamics of Railway Nets: On an Interface between Automatic Control and Software Engineering. In *CTS2003: 10th IFAC Symposium on Control in Transportation Systems*, Oxford, UK, August 4-6 2003. Elsevier Science Ltd. Symposium held at Tokyo, Japan. Editors: S. Tsugawa and M. Aoki. Final version.

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

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

[8] Dines Bjørner. A Container Line Industry Domain. Techn. report, Fredsvej 11, DK-2840 Holte, Denmark, June 2007. Extensive Draft.

[18] http://www.imm.dtu.dk/~dibj/todai/tse-1.pdf
[19] http://www.imm.dtu.dk/~dibj/todai/tse-2.pdf
[20] http://www.imm.dtu.dk/~dibj/wfdftp.pdf
[21] http://www.imm.dtu.dk/~dibj/logistics.pdf
[22] http://www.imm.dtu.dk/~dibj/pipeline.pdf
[23] http://www.imm.dtu.dk/~dibj/pipe-p.pdf
[24] http://www.imm.dtu.dk/~dibj/platoon-p.pdf
[25] http://www.imm.dtu.dk/~dibj/container-paper.pdf
[26] http://www.imm.dtu.dk/~dibj/it-security.pdf
[27] http://www2.imm.dtu.dk/~db/themarket.pdf
[28] http://www2.compute.dtu.dk/~dibj/rails.pdf
[29] http://www.imm.dtu.dk/ dibj/amore/docs/scheduling.pdf
[30] http://www.imm.dtu.dk/ dibj/UNU-IIST-railways.pdf
[31] http://www.imm.dtu.dk/ dibj/amore/docs/dines-ifac.pdf
[32] http://www.imm.dtu.dk/ dibj/amore/docs/albena-amore.pdf
[33] http://www.imm.dtu.dk/ dibj/amore/docs/ifac-dynamics.pdf
[34] http://www.imm.dtu.dk/ dibj/amore/docs/martin-amore.pdf
[35] http://www.imm.dtu.dk/ dibj/amore/docs/panos-amore.pdf
[36] http://www2.imm.dtu.dk/~db/ifac-dynamics.pdf

[9] Dines Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, pages 1–17, Heidelberg, September 2007. Springer.

[10] Dines Bjørner. From Domains to Requirements. In *Montanari Festschrift*, volume 5065 of *Lecture Notes in Computer Science (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer)*, pages 1–30, Heidelberg, May 2008. Springer.

[11] Dines Bjørner. On Mereologies in Computing Science. In *Festschrift: Reflections on the Work of C.A.R. Hoare*, History of Computing (eds. Cliff B. Jones, A.W. Roscoe and Kenneth R. Wood), pages 47–70, London, UK, 2009. Springer.

[12] Dines Bjørner. What is Logistics ? A Domain Analysis. Techn. report, Incomplete Draft, Fredsvej 11, DK-2840 Holte, Denmark, June 2009.

[13] Dines Bjørner. Domain Engineering. In Paul Boca and Jonathan Bowen, editors, *Formal Methods: State of the Art and New Directions*, Eds. Paul Boca and Jonathan Bowen, pages 1–42, London, UK, 2010. Springer.

[14] Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics, Part I of II: The Engineering Part*. *Kibernetika i sistemny analiz*, (4):100–116, May 2010.

[15] Dines Bjørner. Believable Software Management. *Encyclopedia of Software Engineering*, 1(1):1–32, 2011.

[16] Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics Part II of II: The Science Part*. *Kibernetika i sistemny analiz*, (2):100–120, May 2011.

[17] Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. In *Rainbow of Computer Science, Festschrift for Hermann Maurer on the Occasion of His 70th Anniversary.*, Festschrift (eds. C. Calude, G. Rozenberg and A. Saloma), pages 167–183. Springer, Heidelberg, Germany, January 2011.

[18] Dines Bjørner. Documents – a Domain Description[37]. Experimental Research Report 2013-3, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.

[19] Dines Bjørner. *Domain Science and Engineering as a Foundation for Computation for Humanity*, chapter 7, pages 159–177. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC [Francis & Taylor], 2013. (eds.: Justyna Zander and Pieter J. Mosterman).

[20] Dines Bjørner. Pipelines – a Domain Description[38]. Experimental Research Report 2013-2, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.

[21] Dines Bjørner. Road Transportation – a Domain Description[39]. Experimental Research Report 2013-4, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.

[22] Dines Bjørner. *A Rôle for Mereology in Domain Science and Engineering*. Synthese Library (eds. Claudio Calosi and Pierluigi Graziani). Springer, Amsterdam, The Netherlands, October 2014.

[23] Dines Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In Shusaku Iida, José Meseguer, and Kazuhiro Ogata, editors, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014.

[24] Dines Bjørner. Domain Engineering – A Basis for Safety Critical Software. Invited Keynote, ASSC2014: Australian System Safety Conference, Melbourne, 26–28 May, December 2014.

[25] Dines Bjørner. A Credit Card System: Uppsala Draft. Technical Report: Experimental Research, Fredsvej 11, DK–2840 Holte, Denmark, November 2016. http://www.imm.dtu.dk/˜dibj/2016/credit/accs.pdf.

[26] Dines Bjørner. Manifest Domains: Analysis & Description. *Formal Aspects of Computing*, ...(...):1–51, 2016. DOI 10.1007/s00165-016-0385-z http://link.springer.com/article/10.1007/s00165-016-0385-z.

[27] Dines Bjørner. Weather Information Systems: Towards a Domain Description. Technical Report: Experimental Research, Fredsvej 11, DK–2840 Holte, Denmark, November 2016. http://www.imm.dtu.dk/˜dibj/2016/wis/wis-p.pdf.

[28] Dines Bjørner. Urban Planning Processes. Research Note, July 2017. http://www.imm.dtu.dk/˜dibj/2017/up/-urban-planning.pdf.

[29] Dines Bjørner. The Tokyo Stock Exchange Trading Rules. R&D Experiment, Fredsvej 11, DK-2840 Holte, Denmark, January and February, 2010. Version 1, 78 pages: many auxiliary appendices, Version 2, 23 pages: omits many appendices and corrects some errors..

[30] Dines Bjørner. *[31] Chap. 9: Towards a Model of IT Security — – The ISO Information Security Code of Practice – An Incomplete Rough Sketch Analysis*, pages 223–282. JAIST Press, March 2009.

[31] Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering.* A JAIST Press Research Monograph # 4, 536 pages, March 2009.

[32] Dines Bjørner. The Rôle of Domain Engineering in Software Development. Why Current Requirements Engineering Seems Flawed! In *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 2–34, Heidelberg, Wednesday, January 27, 2010. Springer.

[33] Dines Bjørner and Asger Eir. Compositionality: Ontology and Mereology of Domains. Some Clarifying Observations in the Context of Software Engineering in July 2008, eds. Martin Steffen, Dennis Dams and Ulrich Hannemann. In *Festschrift for Prof. Willem Paul de Roever Concurrency, Compositionality, and Correctness*, volume 5930 of *Lecture Notes in Computer Science*, pages 22–59, Heidelberg, July 2010. Springer.

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

[37] http://www.imm.dtu.dk/˜dibj/doc-p.pdf

[38] http://www.imm.dtu.dk/˜dibj/pipe-p.pdf

[39] http://www.imm.dtu.dk/˜dibj/road-p.pdf

184. *Industrial Strength Formal Methods in Practice*, Eds.: Michael G. Hinchey and Jonathan P. Bowen. FACIT, Springer–Verlag, London, England, 1999.

[35] R. Casati and A. Varzi. *Parts and Places: the structures of spatial representation*. MIT Press, 1999.

[36] Chris W. George, Peter Haff, Klaus Havelund, Anne Elisabeth Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.

[37] Chris W. George, Anne Elisabeth Haxthausen, Steven Hughes, Robert Milne, Søren Prehn, and Jan Storbank Pedersen. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[38] Abraham Maslow. A Theory of Human Motivation. *Psychological Review*, 50(4):370–96, 1943. http://psychclassics.yorku.ca/Maslow/motivation.htm.

[39] Abraham Maslow. *Motivation and Personality*. Harper and Row Publishers, 3rd ed., 1954.

[40] Christopher Peterson and Martin E.P. Seligman. *Character strengths and virtues: A handbook and classification*. Oxford University Press, 2004.

[41] Martin Pěnička, Albena Kirilova Strupchanska, and Dines Bjørner. Train Maintenance Routing. In *FORMS'2003: Symposium on Formal Methods for Railway Operation and Control Systems*. L'Harmattan Hongrie, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany. Final version.

[42] Albena Kirilova Strupchanska, Martin Pěnička, and Dines Bjørner. Railway Staff Rostering. In *FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems*. L'Harmattan Hongrie, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany. Final version.

A. Documents in Public Gornment

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 guide lines, 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 presents 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

- cadestral,
- cartograhic and
- zoning

issues. The ancillary notes deal with such things as insufficiencies in the base planss, things that ought be improved in a next iteration base urban plannin, etc. The base plans and ancillerary 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.[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, -2, 0, 1, 2, ...
[3] **Nat** 0, 1, 2, ...
[4] **Real** ..., -5.43, -1.0, 0.0, $1.23\cdots$, $2{,}7182\cdots$, $3{,}1415\cdots$, 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**
[8] A-**infset**
[9] A $\times$ B $\times$ ... $\times$ C
[10] A^*
[11] A^ω
[12] A $\overrightarrow{m}$ B
[13] A $\rightarrow$ B
[14] A $\overset{\sim}{\rightarrow}$ B
[15] (A)
[16] A | B | ... | C
[17] mk_id(sel_a:A,...,sel_b: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:

- either a Cartesian B × C × ... × D, in which case it is identical to type expression kind 9,
- or not to be the name of a built-in type (cf., 1–6) or of a type, in which case the parentheses serve as simple delimiters, e.g., (A $\overrightarrow{m}$ B), or (A*)-**set**, or (A-**set**)list, or (A|B) $\overrightarrow{m}$ (C|D|(E $\overrightarrow{m}$ F)), etc.

16 The postulated disjoint union of types A, B, ..., and C.
17 The record type of mk_id-named record values mk_id(av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.
18 The record type of unnamed record values (av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

Sorts and Observer Functions

```
type
   A, B, C, ..., D
value
   obs_B: A → B, obs_C: A → C, ..., obs_D: A → D
```

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

```
type
   B, C, ..., D
   A = B × C × ... × 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 = Type_expr
```

Some schematic type definitions are:

```
[19]  Type_name = Type_expr /* without |s or subtypes */
[20]  Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[21]  Type_name ==
                mk_id_1(s_a1:Type_name_a1,...,s_ai:Type_name_ai) |
                ... |
                mk_id_n(s_z1:Type_name_z1,...,s_zk:Type_name_zk)
[22]  Type_name :: sel_a:Type_name_a  ...  sel_z:Type_name_z
[23]  Type_name = {| v:Type_name′ • 𝒫(v) |}
```

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

```
   Type_name = A | B | ... | Z
   A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
   B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
   ...
   Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)
```

Types A, B, ..., Z are disjoint, i.e., shares no values, provided all mk_id_k are distinct and due to the use of the disjoint record type constructor ==.

```
axiom
   ∀ a1:A_1, a2:A_2, ..., ai:Ai •
      s_a1(mk_id_1(a1,a2,...,ai))=a1 ∧ s_a2(mk_id_1(a1,a2,...,ai))=a2 ∧
      ... ∧ s_ai(mk_id_1(a1,a2,...,ai))=ai ∧
   ∀ a:A • let mk_id_1(a1′,a2′,...,ai′) = a in
      a1′ = s_a1(a) ∧ a2′ = s_a2(a) ∧ ... ∧ ai′ = s_ai(a) end
```

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 $\mathcal{P}$, constitute the subtype A:

```
type
   A = {| b:B • 𝒫(b) |}
```

C.2.3. Sorts — Abstract Types

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

```
type
   A, B, ..., C
```

C.3. The RSL Predicate Calculus

C.4. Propositional Expressions

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

```
   false, true
   a, b, ..., c ∼a, a∧b, a∨b, a⇒b, a=b, a≠b
```

are propositional expressions having Boolean values. ∼, ∧, ∨, ⇒, = and ≠ are Boolean connectives (i.e., operators). They can be read as: *not*, *and*, *or*, *if then* (or *implies*), *equal* and *not equal*.

C.4.1. Simple Predicate Expressions

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

```
   false, true
   a, b, ..., c
   ∼a, a∧b, a∨b, a⇒b, a=b, a≠b
   x=y, x≠y,
   i<j, i≤j, i≥j, i≠j, i≥j, i>j
```

are simple predicate expressions.

C.4.2. Quantified Expressions

Let X, Y, . . ., C be type names or type expressions, and let $\mathcal{P}(x)$, $\mathcal{Q}(y)$ and $\mathcal{R}(z)$ designate predicate expressions in which x, y and z are free. Then:

```
∀ x:X • $\mathcal{P}(x)$
∃ y:Y • $\mathcal{Q}(y)$
∃ ! z:Z • $\mathcal{R}(z)$
```

are quantified expressions — also being predicate expressions.

They are "read" as: For all x (values in type X) the predicate $\mathcal{P}(x)$ holds; there exists (at least) one y (value in type Y) such that the predicate $\mathcal{Q}(y)$ holds; and there exists a unique z (value in type Z) such that the predicate $\mathcal{R}(z)$ holds.

C.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}, {e₁,e₂,...,eₙ}, ...} ∈ A-set
{{}, {a}, {e₁,e₂,...,eₙ}, ..., {e₁,e₂,...}} ∈ 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, $\ldots$, C, then the below expressions are simple Cartesian enumerations:

```
type
   A, B, ..., C
   A × B × ... × C
value
   (e1,e2,...,en)
```

C.5.4. List Expressions

List Enumerations Let a range over values of type A, then the below expressions are simple list enumerations:

$\{\langle\rangle, \langle e\rangle, \ldots, \langle e1,e2,\ldots,en\rangle, \ldots\} \in A^*$
$\{\langle\rangle, \langle e\rangle, \ldots, \langle e1,e2,\ldots,en\rangle, \ldots, \langle e1,e2,\ldots,en,\ldots\rangle, \ldots\} \in A^\omega$

$\langle$ a_i .. 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.

type
  A, B, P = A → **Bool**, Q = A $\xrightarrow{\sim}$ B
value
  comprehend: A^ω × P × Q $\xrightarrow{\sim}$ B^ω
  comprehend(l,P,Q) ≡ ⟨ Q(l(i)) | i **in** ⟨1..**len** l⟩ • P(l(i))⟩

C.5.5. Map Expressions

Map Enumerations Let (possibly indexed) u and v range over values of type $T1$ and $T2$, respectively, then the below expressions are simple map enumerations:

type
  T1, T2
  M = T1 $\underset{m}{\rightarrow}$ T2
value
  u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
  [ ], [u↦v], ..., [u1↦v1,u2↦v2,...,un↦vn] all ∈ M

Map Comprehension The last line below expresses map comprehension:

type
  U, V, X, Y
  M = U $\underset{m}{\rightarrow}$ V
  F = U $\xrightarrow{\sim}$ X
  G = V $\xrightarrow{\sim}$ Y
  P = U → **Bool**
value
  comprehend: M×F×G×P → (X $\underset{m}{\rightarrow}$ Y)
  comprehend(m,F,G,P) ≡ [F(u) ↦ G(m(u)) | u:U • u ∈ **dom** m ∧ P(u)]

C.5.6. Set Operations

Set Operator Signatures

value
  19 ∈: A × **A-infset** → **Bool**
  20 ∉: A × **A-infset** → **Bool**
  21 ∪: **A-infset** × **A-infset** → **A-infset**
  22 ∪: (**A-infset**)-**infset** → **A-infset**
  23 ∩: **A-infset** × **A-infset** → **A-infset**
  24 ∩: (**A-infset**)-**infset** → **A-infset**
  25 \: **A-infset** × **A-infset** → **A-infset**
  26 ⊂: **A-infset** × **A-infset** → **Bool**
  27 ⊆: **A-infset** × **A-infset** → **Bool**
  28 =: **A-infset** × **A-infset** → **Bool**
  29 ≠: **A-infset** × **A-infset** → **Bool**
  30 **card**: **A-infset** $\xrightarrow{\sim}$ **Nat**

Set Examples

examples

$a \in \{a,b,c\}$
$a \notin \{\}$, $a \notin \{b,c\}$
$\{a,b,c\} \cup \{a,b,d,e\} = \{a,b,c,d,e\}$
$\cup\{\{a\},\{a,bb\},\{a,d\}\} = \{a,b,d\}$
$\{a,b,c\} \cap \{c,d,e\} = \{c\}$
$\cap\{\{a\},\{a,bb\},\{a,d\}\} = \{a\}$
$\{a,b,c\} \setminus \{c,d\} = \{a,bb\}$
$\{a,bb\} \subset \{a,b,c\}$
$\{a,b,c\} \subseteq \{a,b,c\}$
$\{a,b,c\} = \{a,b,c\}$
$\{a,b,c\} \neq \{a,bb\}$
card $\{\} = 0$, **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 $=$: 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 **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):

```
value
    $s' \cup s'' \equiv \{ a \mid a{:}A \bullet a \in s' \vee a \in s'' \}$
    $s' \cap s'' \equiv \{ a \mid a{:}A \bullet a \in s' \wedge a \in s'' \}$
    $s' \setminus s'' \equiv \{ a \mid a{:}A \bullet a \in s' \wedge a \notin s'' \}$
    $s' \subseteq s'' \equiv \forall a{:}A \bullet a \in s' \Rightarrow a \in s''$
    $s' \subset s'' \equiv s' \subseteq s'' \wedge \exists a{:}A \bullet a \in s'' \wedge a \notin s'$
    $s' = s'' \equiv \forall a{:}A \bullet a \in s' \equiv a \in s'' \equiv s{\subseteq}s' \wedge s'{\subseteq}s$
    $s' \neq s'' \equiv s' \cap s'' \neq \{\}$
    card s ≡
        if s = {} then 0 else
        let a:A • a ∈ s in 1 + card (s \ {a}) end end
        pre s /∗ is a finite set ∗/
    card s ≡ chaos /∗ tests for infinity of s ∗/
```

C.5.7. Cartesian Operations

type
  A, B, C
  g0: $G0 = A \times B \times C$
  g1: $G1 = (A \times B \times C)$
  g2: $G2 = (A \times B) \times C$
  g3: $G3 = A \times (B \times C)$

value
  va:A, vb:B, vc:C, vd:D
  (va,vb,vc):G0,
  (va,vb,vc):G1
  ((va,vb),vc):G2
  (va3,(vb3,vc3)):G3

decomposition expressions
  let (a1,b1,c1) = g0,
    (a1′,b1′,c1′) = g1 **in** .. **end**
  let ((a2,b2),c2) = g2 **in** .. **end**
  let (a3,(b3,c3)) = g3 **in** .. **end**

C.5.8. List Operations

List Operator Signatures

value
  hd: $A^\omega \xrightarrow{\sim} A$
  tl: $A^\omega \xrightarrow{\sim} A^\omega$
  len: $A^\omega \xrightarrow{\sim}$ **Nat**
  inds: $A^\omega \rightarrow$ **Nat-infset**
  elems: $A^\omega \rightarrow$ A-**infset**
  .(.): $A^\omega \times$ **Nat** $\xrightarrow{\sim} A$
  $\frown$: A^*: $\neq A$: $A^\omega A^\omega$ A^ω $A^\omega \rightarrow$ **BoBbol**

List Operation Examples

examples
  hd$\langle$a1,a2,...,am$\rangle$=a1
  tl$\langle$a1,a2,...,am$\rangle$=$\langle$a2,...,am$\rangle$
  len$\langle$a1,a2,...,am$\rangle$=m
  inds$\langle$a1,a2,...,am$\rangle$={1,2,...,m}
  elems$\langle$a1,a2,...,am$\rangle$={a1,a2,...,am}
  $\langle$a1,a2,...,am$\rangle$(i)=ai
  $\langle$a,b,c$\rangle\frown\langle$a,b,d$\rangle$ = $\langle$a,b,c,a,b,d$\rangle$
  $\langle$a,b,c$\rangle$=$\langle$a,b,c$\rangle$
  $\langle$a,b,c$\rangle \neq \langle$a,b,d$\rangle$

Informal Explication

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i, gives the ith element of the list.
- $\frown$: 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
  is_finite_list: A^ω → Bool

  len q ≡
    case is_finite_list(q) of
      true → if q = ⟨⟩ then 0 else 1 + len tl q end,
      false → chaos end

  inds q ≡
    case is_finite_list(q) of
      true → { i | i:Nat • 1 ≤ i ≤ len q },
      false → { i | i:Nat • i≠0 } end

  elems q ≡ { q(i) | i:Nat • i ∈ inds q }

  q(i) ≡
    if i=1
      then
        if q≠⟨⟩
          then let a:A,q′:Q • q=⟨a⟩^q′ in a end
          else chaos end
      else q(i−1) end

  fq ^ iq ≡
      ⟨ if 1 ≤ i ≤ len fq then fq(i) else iq(i − len fq) end
       | i:Nat • if len iq≠chaos then i ≤ len fq+len end ⟩
    pre is_finite_list(fq)

  iq′ = iq″ ≡
    inds iq′ = inds iq″ ∧ ∀ i:Nat • i ∈ inds iq′ ⇒ iq′(i) = iq″(i)

  iq′ ≠ iq″ ≡ ∼(iq′ = iq″)
```

C.5.9. Map Operations

Map Operator Signatures and Map Operation Examples

```
value
  m(a): M → A ~→ B, m(a) = b

  dom: M → A-infset [domain of map]
     dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}

  rng: M → B-infset [range of map]
     rng [a1↦b1,a2↦b2,...,an↦bn] = {b1,b2,...,bn}

  †: M × M → M [override extension]
     [a↦b,a′↦bb′,a″↦bb″] † [a′↦bb″,a″↦bb′] = [a↦b,a′↦bb″,a″↦bb′]

  ∪: M × M → M [merge ∪]
     [a↦b,a′↦bb′,a″↦bb″] ∪ [a‴↦bb‴] = [a↦b,a′↦bb′,a″↦bb″,a‴↦bb‴]

  \: M × A-infset → M [restriction by]
     [a↦b,a′↦bb′,a″↦bb″]\{a} = [a′↦bb′,a″↦bb″]
```

```
/: M × A-infset → M [restriction to]
    [a↦b,a′↦bb′,a″↦bb″]/{a′,a″} = [a′↦bb′,a″↦bb″]

=,≠: M × M → Bool

°: (A →m B) × (B →m C) → (A →m C) [composition]
    [a↦b,a′↦bb′] ° [bb↦c,bb′↦c′,bb″↦c″] = [a↦c,a′↦c′]
```

Map Operation Explication

- $m(a)$: Application gives the element that a maps to in the map m.
- **dom**: Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.
- †: Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some "pairings" of the right operand map.
- ∪: Merge. When applied to two operand maps, it gives a merge of these maps.
- \: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements that are not in the right operand set.
- /: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.
- =: The equal operator expresses that the two operand maps are identical.
- ≠: The nonequal operator expresses that the two operand maps are *not* identical.
- °: Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1, to the range elements of the right operand map, m_2, such that if a is in the definition set of m_1 and maps into b, and if b is in the definition set of m_2 and maps into c, then a, in the composition, maps into c.

Map Operation Redefinitions The map operations can also be defined as follows:

```
value
  rng m ≡ { m(a) | a:A • a ∈ dom m }

  m1 † m2 ≡
    [ a↦b | a:A,b:B •
       a ∈ dom m1 \ dom m2 ∧ bb=m1(a) ∨ a ∈ dom m2 ∧ bb=m2(a) ]

  m1 ∪ m2 ≡ [ a↦b | a:A,b:B •
               a ∈ dom m1 ∧ bb=m1(a) ∨ a ∈ dom m2 ∧ bb=m2(a) ]

  m \ s ≡ [ a↦m(a) | a:A • a ∈ dom m \ s ]
  m / s ≡ [ a↦m(a) | a:A • a ∈ dom m ∩ s ]

  m1 = m2 ≡
    dom m1 = dom m2 ∧ ∀ a:A • a ∈ dom m1 ⇒ m1(a) = m2(a)
  m1 ≠ m2 ≡ ∼(m1 = m2)

  m°n ≡
    [ a↦c | a:A,c:C • a ∈ dom m ∧ c = n(m(a)) ]
    pre rng m ⊆ dom n
```

C.6. λ-Calculus + Functions

C.6.1. The λ-Calculus Syntax

```
type /∗ A BNF Syntax: ∗/
    ⟨L⟩ ::= ⟨V⟩ | ⟨F⟩ | ⟨A⟩ | ( ⟨A⟩ )
    ⟨V⟩ ::= /∗ variables, i.e. identifiers ∗/
    ⟨F⟩ ::= λ⟨V⟩ • ⟨L⟩
    ⟨A⟩ ::= ( ⟨L⟩⟨L⟩ )
value /∗ Examples ∗/
    ⟨L⟩: e, f, a, ...
    ⟨V⟩: x, ...
    ⟨F⟩: λ x • e, ...
    ⟨A⟩: f a, (f a), f(a), (f)(a), ...
```

C.6.2. Free and Bound Variables

Let x, y be variable names and e, f be λ-expressions.

- ⟨V⟩: Variable x is free in x.
- ⟨F⟩: x is free in λy •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:

- **subst**([N/x]x) ≡ N;
- **subst**([N/x]a) ≡ a,

  for all variables a≠ x;
- **subst**([N/x](P Q)) ≡ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x](λx•P)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ λy• **subst**([N/x]P),

  if x≠y and y is not free in N or x is not free in P;
- **subst**([N/x](λy•P)) ≡ λz•**subst**([N/z]**subst**([z/y]P)),

  if y≠x and y is free in N and x is free in P
  (where z is not free in (N P)).

C.6.4. α-Renaming and β-Reduction

- α-renaming: λx•M
  If x, y are distinct variables then replacing x by y in λx•M results in λy•**subst**([y/x]M). We can rename the formal parameter of a λ-function expression provided that no free variables of its body M thereby become bound.
- β-reduction: (λx•M)(N)
  All free occurrences of x in M are replaced by the expression N provided that no free variables of N thereby become bound in the result. (λx•M)(N) ≡ **subst**([N/x]M)

C.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
    g(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 $\overset{\sim}{\to}$ 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 = $\mathcal{E}_d$ **in** $\mathcal{E}_b$(a) **end**

is an "expanded" form of:

(λa.$\mathcal{E}_b$(a))($\mathcal{E}_d$)

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 = **Y**F **in** B(f,a) **end**

where:

$$F \equiv \lambda g \bullet \lambda a \bullet (E(g)) \text{ and } YF = F(YF)$$

C.7.3. Predicative let Expressions

Predicative **let** expressions:

```
let a:A • 𝒫(a) in ℬ(a) end
```

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

C.7.4. Pattern and "Wild Card" let Expressions

Patterns and *wild cards* can be used:

```
let {a} ∪ s = set in ... end
let {a,_} ∪ s = set in ... end

let (a,b,...,c) = cart in ... end
let (a,_,...,c) = cart in ... end

let ⟨a⟩^ℓ = list in ... end
let ⟨a,_,bb⟩^ℓ = list in ... end

let [a↦bb] ∪ m = map in ... end
let [a↦b,_] ∪ m = map in ... end
```

C.7.5. Conditionals

Various kinds of conditional expressions are offered by RSL:

```
if b_expr then c_expr else a_expr
end

if b_expr then c_expr end ≡ /∗ same as: ∗/
   if b_expr then c_expr else skip end

if b_expr_1 then c_expr_1
elsif b_expr_2 then c_expr_2
elsif b_expr_3 then c_expr_3
...
elsif b_expr_n then c_expr_n end

case expr of
   choice_pattern_1 → expr_1,
   choice_pattern_2 → expr_2,
   ...
   choice_pattern_n_or_wild_card → expr_n
end
```

C.7.6. Operator/Operand Expressions

```
⟨Expr⟩ ::=
              ⟨Prefix_Op⟩ ⟨Expr⟩
             | ⟨Expr⟩ ⟨Infix_Op⟩ ⟨Expr⟩
             | ⟨Expr⟩ ⟨Suffix_Op⟩
             | ...
⟨Prefix_Op⟩ ::=
             − | ∼ | ∪ | ∩ | card | len | inds | elems | hd | tl | dom | rng
⟨Infix_Op⟩ ::=
              = | ≠ | ≡ | + | − | ∗ | ↑ | / | < | ≤ | ≥ | > | ∧ | ∨ | ⇒
             | ∈ | ∉ | ∪ | ∩ | \ | ⊂ | ⊆ | ⊇ | ⊃ | ^ | † | °
⟨Suffix_Op⟩ ::= !
```

C.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. stm_1;stm_2;...;stm_n
```

C.8.4. Imperative Conditionals

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

C.8.5. Iterative Conditionals

```
6. while expr do stm end
7. do stmt until expr end
```

C.8.6. Iterative Sequencing

8. **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:KIdx for channel array indexes, then:

```
channel c:A
channel { k[i]:B • i:KIdx }
```

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:

P ‖ Q	Parallel composition
P ⌷ Q	Nondeterministic external choice (either/or)
P ⌈⌉ Q	Nondeterministic internal choice (either/or)
P ╫ Q	Interlock parallel composition

express the parallel (‖) of two processes, or the nondeterministic choice between two processes: either external (⌷) or internal (⌈⌉). The interlock (╫) 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:

c ?, k[i] ?	Input
c ! e, k[i] ! e	Output

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.

```
value
    P: Unit → in c out k[i]
    Unit
    Q: i:KIdx →  out c in k[i] Unit

    P() ≡ ... c ? ... k[i] ! e ...
    Q(i) ≡ ... k[i] ? ... c ! e ...
```

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.[41]

[41] For schemes, classes and objects we refer to [6, Chap. 10]