Documents

A Basis for Government[1]

Dines Bjørner

Technical University of Denmark, DK-2800 Kgs. Lyngby, Denmark
Fredsvej 11, DK-2840 Holte, Danmark
E–Mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

Abstract. At least two facets characterise electronic government: (i) the timely, orderly and human-oriented delivery of government services to its citizens and private organisations (businesses, etc.), and (ii) the documents underlying the laws and regulations affording these services – with these documents originating and emanating from the legislative, the executive and the judicial authorities as first expressed by *Charles-Louis de Secondat, baron de La Brède et de Montesquieu* (1689–1755).

It seems that item (ii) is yet to be "in-depth-explored" by the electronic government community researchers – so that, then, is the purpose of this contribution.

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 one can develop such from our domain description.

You may find that the model is overly detailed with respect to a number of "operations" and properties of documents. We find that these operations must be part of the very basis of a document domain in order to cope with documents such as they occur in such wide applications as, for example, public government.

Contents

[1] Paper submited for the *Elsa Estavez & Tomasz Janowski* Festschrift: *Digital governance for sustainable development and empowered citizenship*, October 3, 2022
Correspondence and offprint requests to: Dines Bjørner, Fredsvej 11, DK 2840 Holte, Denmark

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 are 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 paper 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 **handler**s' **edit**ing and **read**ing of **document**s is accomplished by the **handler**s "working directly" with the **document**s (i.e., synchronising and communicating with **document behaviour**s). The **handler**s' **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 [1, 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 [1, Manifest Domains: Analysis & Description].

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

[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 paper – as it has nothing to do with the essence of document handling, but, perhaps, with 'management'.

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
5 uid_HI: H → HI
5 uid_DI: D → DI

As reasoned in [1, 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: *"18 July 2022: 13:46: This is version V."*, *"This document was released on 18 July 2022: 13:46."*, *"18 July 2022: 13:46: Section X.Y.Z of version III was deleted."*, *"18 July 2022: 13:46: References to documents doc_i and doc_j are inserted on Pages p and q, respectively."* and *"18 July 2022: 13:46: Final release."*

[4] [1, 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] 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.

2.4.4. 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*.

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   DA = DN*
8   DH = (TIME × DO)*
9   DO == Crea | Edit | Read | Copy | Shre
10  Crea :: (HI × TIME) × (DI-set × Info) × DN × {|"empty_DC"|}
```

```
10bi  Info = ...
value
10bii  embed_DIs_in_DD: DI-set × Info → DD
axiom
10d  ″empty_DC″ ∈ 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 [1, 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.

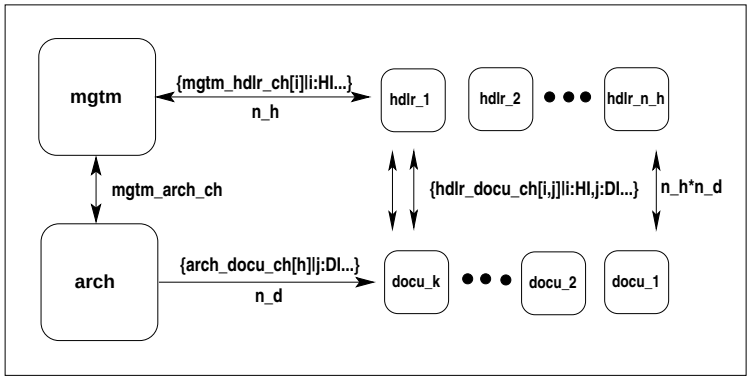

Fig. 1. An Informal Snapshot of System Behaviours

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 [5] Specification Language, RSL [4]. 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 11). 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 [1], 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 MDIR = HI $\overrightarrow{m}$ (DI $\overrightarrow{m}$ ANm-**set**)
41 ADIR = avail:DI-**set** × used:DI-**set** × gone:DI-**set**
42 HDIR = DI $\overrightarrow{m}$ ANm-**set**
43 SDATR = DD, PDATR = DA × DC × DH
axiom
41 ∀ (avail,used,gone):ADIR • avail ∩ used = {} ∧ gone ⊆ used

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

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

2.11. Inter-Behaviour Messages

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

2.11.1. Management Messages with Respect to the Archive

44 Management **create** documents. It does so by requesting the archive behaviour to allocate a document identifier and initialize the document "state" and start a document behaviour, with initial information, cf. Item 10 on Page 5:

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. Crea :: (HI × TIME) × (DI-**set** × Info) × DN × {|"empty_DC"|} [cf. formula Item 10, Page 5]

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

45 Copy :: DI × HI × TIME × DN [cf. Item 55 on Page 12]

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

type
46 Shred :: TIME × 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 × TIME × DI × 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.

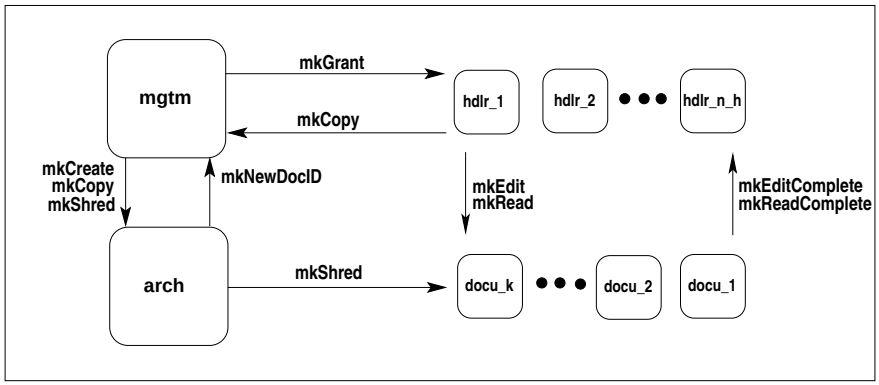

Fig. 2. A Summary of Behaviour Interactions

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 10]

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 10) | Shred (Item 44d on Page 10) | NewDocID (Item 51 on Page 11)
25 MH = Grant (Item 44c on Page 10) | Copy (Item 55 on the facing page) |
26 AD = Shred (Item 52 on Page 11)
27 HD = Edit (Item 53 on Page 11) | Read (Item 54 on the preceding page) | Copy (Item 55 on the facing 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 10, Items 44–44e.)

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 11, 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 11, Item 48 on Page 11.)

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 following 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.

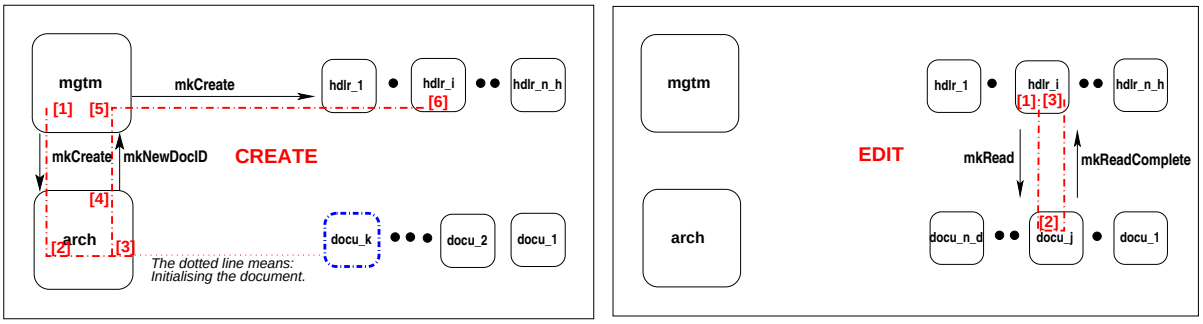

Fig. 3. Informal Snapshots of Create and Edit Document Behaviours

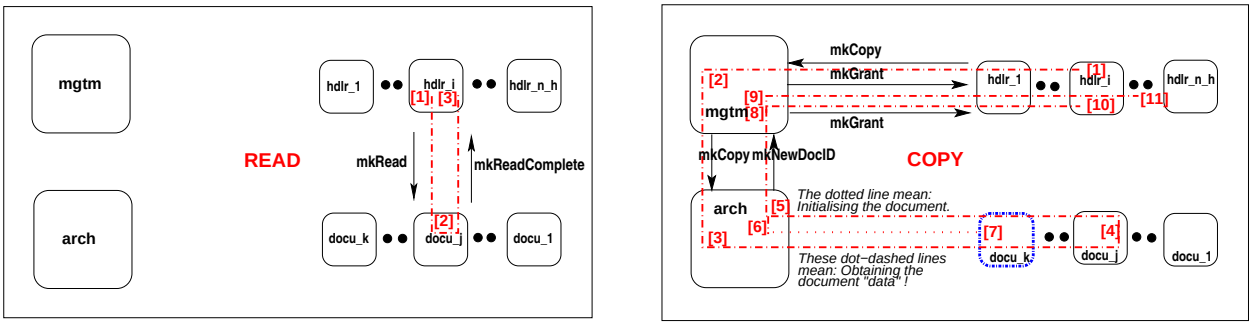

Fig. 4. Informal Snapshots of Read and Copy Document Behaviours

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,

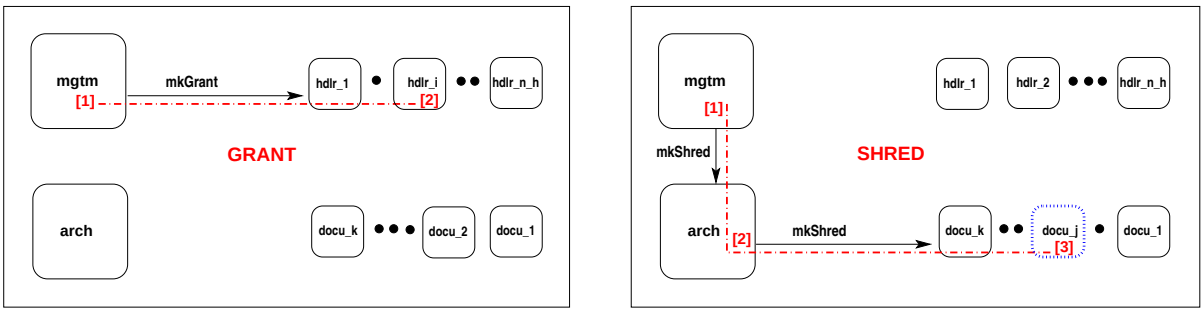

Fig. 5. Informal Snapshots of Grant and Shred Document Behaviours

9 while informing these handlers (informally indicated by a "dangling" dash-dotted line) of their grants,
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 preceding page Left
b **copy** Fig. 4 on the facing 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 14

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 14]
65  [5]   mgtm_create_awaits_response(mdir)(hi) end  [Left Fig. 3 on Page 14]
```

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 14

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 15

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,arcs: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 15 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 14 Left
b **copy** — Fig. 4 on Page 14 Right
c **shred** — Fig. 5 on Page 15 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 14 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 14 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 15 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 14 Left
b **edit** — Fig. 3 on Page 14 Right
c **read** — Fig. 4 on Page 14 Left
d **copy** — Fig. 4 on Page 14 Right
e **grant** — Fig. 5 on Page 15 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 14

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 14

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 14

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 14

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 15

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 14 Right

b **read** Fig. 4 on Page 14 Left
c **shred** Fig. 5 on Page 15 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 14

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 *e*dit (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 14

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 15

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

We have shown an example of an analysis and description of a domain of documents. The method, its principles, techniques and tools, for analysing and describing domains has been developed in [1, 2] and has found a monographic in-depth treatment in [3].

3. References

[1] Dines Bjørner. Manifest Domains: Analysis & Description `www.imm.dtu.dk/ dibj/2015/faoc/faoc-bjorner.pdf`. *Formal Aspects of Computing*, 29(2):175–225, March 2017. Online: 26 July 2016.

[2] Dines Bjørner. Domain Analysis & Description – Principles, Techniques and Modelling Languages. `www.imm.dtu.-dk/~dibj/2018/tosem/Bjorner-TOSEM.pdf`. *ACM Trans. on Software Engineering and Methodology*, 28(2), April 2019. 68 pages.

[3] Dines Bjørner. *Domain Science & Engineering – A Foundation for Software Development*. EATCS Monographs in Theoretical Computer Science. Springer, 2021.

[4] 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.

[5] 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.