Towards a Formal Understanding of Urban Planning
Some Initial Thoughts

Dines Bjørner[1], Otthein Herzog[2], Siegfried ZhiQiang Wu[3]
[1] Techn.Univ. of Denmark and Fredsvej 11, DK-2840 Holte, Denmark
[2] Jacobs University, Campus Ring 1, 28759 Bremen, Germany
[3] CIUC - Tongji University, Siping Campus, Shanghai, China

E–Mail: bjorner@gmail.com, herzog@rundhof.de, ...

Version 2 January 3, 2018: 09:32 am CET

Abstract

[1] We examine concepts of urban planning. Emphasis, in this research note, is on the *information* ("data") and *functions* (*behaviours*) of urban planning. We abstract from the details of information (the "data") that urban planning is based on and results in. We distinguish between two kinds of urban planning behaviour: the *master*, 'ab initio', behaviour of determining "the general layout of the land (!)", and the *derived*, 'follow-up', behaviours focused on social and technological infrastructures. Master urban planning applies to descriptions of "the land": geographic, that is, geodetic, cadastral, geotechnical, meteorological, socio-economic and rules & regulations. Examples of derived urban plannings are such which are focused on humans and on social and technological artifacts: *industry zones, commercial (i.e., office and shopping) zones, residential zones, recreational areas, health care, schools, etc.* and *transport, electricity, water, waste, etc.* The overall aim of this paper is to suggest a formal foundation for urban planning. We must emphasize that all that is conceivable and describable in the domain can be described. We shall return to this remark, in this report, again-and-again.

Editorial Notes:

- Section 1.4 on Page 8 is new. It was added Mon., Dec. 25, 2017.
- Section 1.5 on Page 8 is new. It was added Mon., Dec. 25, 2017.
- Section 1.6 on Page 9 is new. It was added Tue., Dec. 26, 2017.
- Section 6.3 on Page 28 is new. It was added Tue., Jan. 2, 2018.

Contents

[1]This is Version 2 of the present document. Version 0 was issued 24 September 2017. Subsequent editions of Version 2 will appear, from day to day, during the winter of 2017/2018.

1 Introduction

“Urban planning is a technical and political process concerned with the development and use of land, planning permission, protection and use of the environment, public welfare, and the design of the urban environment, including air, water, and the infrastructure passing into and out of urban areas, such as transportation, communications, and distribution networks.”[2]

In this research note we shall try to understand two of the aspects of the domain underlying urban planning, (i) namely those of the “input” information to and “output” plans (etc.) from urban planning, and (ii) that of some possible urban planning (development) functions and processes. We are trying to understand and describe a domain, not requirements for IT for that domain and certainly not the IT (incl. its software). And: We are certainly not constructing any general or any specific urban plan!

The overall aim of this report is to suggest a formal foundation for urban planning. Another, secondary aim of this report is to suggest that a number of requirements must be satisfied before a fully professional urban development project can be commenced (cf. Sect. 7.1).

1.1 On Urban Planning

We search for answers to the question: *“What is Urban Planning?”*. First we identiffy “planning areas”. Then we sketch element of a first domain model for Urban Planning.

Urban planning seems to be also be about *infrastructure planning*. So we examine these terms. First the latter, then the former.

1.1.1 Infrastructures

The term ‘infrastructure’ has gained currency in the last 80 years.[3]. It is more frequently used in socio-economic than in scientific, let alone computing science, contexts. According to the World Bank, ‘infrastructure’ is an umbrella term for many activities referred to as ‘social overhead capital’ by some development economists, and encompasses activities that share technical and economic features (such as economies of scale and spill-overs from users to non-users). We take a more technical view, and see infrastructures as concerned with supporting other systems or activities. Software for infrastructures is likely to be distributed and concerned in particular with supporting communication of data, people

[2]https://en.wikipedia.org/wiki/Urban_planning

[3]Winston Churchill is quoted to have said, in the House of Commons, in 1936: *. . . the young Labourite speaker, that we just heard, obviously wishes to impress his constituency with the fact that he has attended Eton and Oxford when he uses such modern terms as ‘infrastructure’ . . .*

and/or materials. Hence issues of openness, timeliness, security, lack of corruption and resilience are often important.

Examples of infrastructures, or, more precisely, infrastructure components, are:

- transport systems (roads, railways, air traffic, canals/rivers/lake/ocean , etc.);
- water and sewage;
- telecommunications;
- postal service (physical letters, packages etc.);
- power: electricity, gas, oil, wind (generation, distribution); etc.
- the financial industry (banking, insurance, securities, clearing, etc.);
- documents (creation, editing, formatting, etc.);
- ministry of finance (taxation, budget, treasury, etc.);
- health care (private physicians, clinics, hospitals, pharmacies, etc.);
- education (kindergartens, pre-schools, primary schools, secondary schools, high schools, colleges, universities);
- manufacturing industry;
- etcetera.

1.1.2 Wikipedia: https://en.wikipedia.org/wiki/Urban_planning

"Urban planning is a technical and political process concerned with the **development and use of land** *planning permission, protection and use of the environment,* **public wellfare**, *and the design of the urban environment, including* **air, water, and the infrastructure** *passing into and out of urban areas, such as* **transportation, communications, and distribution networks** *[2]."*

"Urban planning is also referred to as urban and regional planning, regional planning, town planning, city planning, rural planning or some combination in various areas worldwide. It takes many forms and it can share perspectives and practices with urban design [1]."

"Urban planning guides orderly development in urban, suburban and rural areas. Although predominantly concerned with the planning of settlements and communities, urban planning is also responsible for the **planning and development of water use and resources, rural and agricultural land, parks and conserving areas of natural environmental** *significance. Practitioners of urban planning are concerned with research and analysis, strategic thinking, architecture, urban design, public consultation, policy recommendations, implementation and management [3]."*

"Urban planners work with the cognate fields of architecture, landscape architecture, civil engineering, and public administration to achieve strategic, policy and sustainability goals. Early urban planners were often members of these cognate fields. Today urban planning is a separate, independent professional discipline. The discipline is the broader category that includes different sub-fields such as land-use planning, zoning, economic development, environmental planning, and transportation planning [4]."

1.1.3 Theories of Urban Planning

"Planning theory is the body of scientific concepts, definitions, behavioral relationships, and assumptions that define the body of knowledge of urban planning. There are eight procedural theories of planning that remain the principal theories of planning procedure today: the rational-comprehensive approach, the incremental approach, the transactive approach, the communicative approach, the advocacy approach, the equity approach, the radical approach, and the humanist or phenomenological approach [5]."

Technical aspects Technical aspects of urban planning involve applying scientific, technical processes, considerations and features that are involved in planning for land use, urban design, natural resources, transportation, and infrastructure. Urban planning includes techniques such as: predicting population growth, zoning, geographic mapping and analysis, analyzing park space, surveying the water supply, identifying transportation patterns, recognizing food supply demands, allocating healthcare and social services, and analyzing the impact of land use.

Urban planners An urban planner is a professional who works in the field of urban planning for the purpose of optimizing the effectiveness of a community's land use and infrastructure. They formulate plans for the development and management of urban and suburban areas, typically analyzing land use compatibility as well as economic, environmental and social trends. In developing the plan for a community (whether commercial, residential, agricultural, natural or recreational), urban planners must also consider a wide array of issues such as sustainability, air pollution, traffic congestion, crime, land values, legislation and zoning codes.

The importance of the urban planner is increasing throughout the 21st century, as modern society begins to face issues of increased population growth, climate change and unsustainable development. An urban planner could be considered a green collar professional.[clarification needed]

1.1.4 References

1 "What is Urban Planning" (retrieved April 24, 2015)
https://mcgill.ca/urbanplanning/planning

"Modern urban planning emerged as a profession in the early decades of the 20th century, largely as a response to the appalling ***sanitary, social, and economic*** *conditions of rapidly-growing industrial cities. Initially the disciplines of architecture and civil engineering provided the nucleus of concerned professionals. They were joined by* ***public health*** *specialists, economists, sociologists, lawyers, and geographers, as the complexities of managing cities came to be more fully understood. Contemporary urban and regional planning techniques for survey, analysis, design, and implementation developed from an interdisciplinary synthesis of these fields. Today, urban planning can be described as a technical and political process concerned with the welfare of people, control of the use of land, design of the urban environment including transportation and communication networks, and protection and enhancement of the natural environment."*

2 Van Assche, K., Beunen, R., Duineveld, M., & de Jong, H. (2013). Co-evolutions of planning and design: Risks and benefits of design perspectives in planning systems. Planning Theory, 12(2), 177-198.

3 Taylor, Nigel (2007). Urban Planning Theory since 1945, London, Sage.

4 https://www.planning.org/aboutplanning/whatisplanning.htm: "What Is Planning?". www.planning.org. Retrieved 2015-09-28.

5 https://www.planetizen.com/node/73570/how-planners-use-planning-theory: How Planners Use Planning Theory. Andrew Whittmore of the University of North Carolina Department of Urban and Regional Planning identifies planning theory in everyday practice.

1.2 A Triptych of Software Development

Before hardware and software systems can be designed and coded we must have a reasonable grasp of "its" requirements; before requirements can be prescribed we must have a reasonable grasp of "the underlying" domain. To us, therefore, software engineering contains the three sub-disciplines:

- domain engineering,
- requirements engineering and

- software design.

By a domain description we understand a collection of pairs of narrative and commensurate formal texts, where each pair describes either aspects of an endurant (i.e., a data) entity or aspects of a perdurant (i.e., an action, event or behaviour) entity.

1.3 On Formality

We consider software programs to be formal, i.e., mathematical, quantities — rather than of social/psychological interest. We wish to be able to reason about software, whether programs, or program specifications, or requirements prescriptions, or domain descriptions. Although we shall only try to understand some facets of the domain of urban planning we shall eventually let such an understanding, in the form of a precise, formal, mathematical, although non-deterministic, i.e., "multiple choice", description be the basis for subsequent requirements prescriptions for software support, and, again, eventually, "the real software itself", that is, tools, for urban planners. We do so, so that we can argue, eventually prove formally, that the software *is correct* with respect to the (i.e., its) formally prescribed requirements, and that the software *meets customer*, i.e., domain users' *expectations* – as expressed in the formal domain description.

1.4 On Describing Domains

If we can describe some domain phenomenon in logical statements and if these can be transcribed into some form of mathematical logic and set theory then we may have to describe it: narratively and formally. That is, even though it may be humanly or even technologically very cumbersome or even impossible to implement what is described we may find it necessary to describe it. *As to when we have to describe something – that is another matter !*[4] Let us give an example: The example is that of the domain of *documents.* Documents may be *created, edited, read, copied, referred to,* and *shredded.* We may talk, meaningfully, that is, rationally, logically, about the previous *version* of a document, and hence we may be obliged to model document versions as from their first creation, *who created, who edited, who read, who copied,* and *who shredded* (sic !) a document, including, perhaps, the *location* and *time* of these operations, *how they were edited*, etc., etc. Let us take another example. As for the meteorological properties of any specific geographic area, these properties, like temperature, humidity, wind, etc., vary, in reality, continuously over time, from location to location, including altitude. In modeling meteorological properties we may be well-served when modeling exactly their continuous, however "sporadic" nature. To a first approximation we do not have to bother as to whether we can actually "implement" the recording of such continuous, "sporadic" "behaviours". In that sense the domain analyser cum describer is expected to be like the physicists,[5] certainly not like programmers. That is: *the domain analyser cum describer are not necessarily describing computable domains.*

1.5 Reiterating Domain Modeling

Any domain description is an approximation. One cannot ever hope to have described all facets of any domain. So, in setting out to analyse & describe a domain one is not trying to produce a definitive, final, model; one is merely studying and recording (some) results of that study. One is prepared to reiterate the study and produce alternative models. From such models one can develop requirements, [4], for software that in one way or another support activities of the domain. If you are

[4]We may find occasions in this document to discuss this "other matter" !

[5]It is written above: that domain descriptions are based on mathematical logic and set theory. Yes, unfortunately ! To properly describe domains involving continuity we need "mix" logic with classical calculus: differential equations, integrals, etc. And here we have nothing to say: the ability, in an informed ways, to blend mathematical logic and set theoretic descriptions with differential equations, integrals, etc., is almost non-existent as of 2017/2018 !

to seriously develop software in this way, for example for the support of urban planners, then you must be prepared to "restart" the process, to develop, from scratch, a domain model. You have a basis from which to start, namely this report [9]. But do not try to simply modify it. Study [9] in depth, but rethink that basis. A description, any description, can be improved. Perhaps the emphasis should be refocused. For the example of software (incl. IT) support for the keeping, production, editing, etc., of the very many documents that are needed during urban planning, you may, in addition to refocusing the present report's focus on the documents of the very many document categories that are presumed, introduced and further elaborated upon in the present report, also study [5]. A principle guiding us in the reformulation of a domain model to be the basis for a specific software product is that we must strive to document all the assumptions about the context in which this software is to serve – otherwise we cannot hope to achieve a product that meets its customers expectations.

1.6 Partial, Precise, and Approximate Descriptions

By a *partial description* we mean a description which covers only a fraction of the domain as a group of people working in that domain, that is, professionals, would otherwise talk about. Descriptions are here taken to describe *behaviours: first "do this", then "do that" !* By a *precise description* we mean a description which in whatever behaviour it describes, partially or fully, does so precisely, that is, it is precisely as described, no more, no less. By an *approximate description* we mean a description which in whatever behavior it describes, partially or fully, even when precisely so, allows for a set of interpretations. We shall then avail ourselves of two forms of 'approximation': *internal non-determinism* and *external non-determinism*. By *internal non-deterministic behaviour* we shall mean a behaviour whose "next step, next move" is "determined" by some "own flipping a coin". By *external non-deterministic behaviour* we shall mean a behaviour whose "next step, next move" is "determined" by some "outside demon" ! In describing urban planning we shall allow for: partial descriptions: not all is described and what has been selected for description has been so, perhaps rather arbitrarily, by us, i.e., me, and both forms of 'approximation'. We shall endeavour to indicate where and why we present only partial descriptions, and deploy 'approximation'.

1.7 On Formal Notations

To be able to *prove formal correctness* and *meeting customer expectations* we avail ourselves of some formal notation. In this research note we use the `RAISE` [12] Specification Language, `RSL`, [11]. Other formal notations, such as `Alloy` [14], `Event B` [1], `VDM-SL` [7, 8, 10] or `Z` [15] could be used. We choose `RSL` since it, to our taste, nicely embodies Hoare's concept of *Communicating Sequential Processes*, `CSP` [13].

1.8 On the Form of This Research Note

The present form of this research note, as of January 3, 2018: 09:32 am, is that of recording a development. The development is that of *trying to come to grips with what urban planning is*. We have made the decision, from an early start, that urban planning *"as a whole"* is a collection of one master and an evolving number of (initially zero) derived urban planning behaviours. Here we have made the choice to model the various behaviours of a complex of urban planning functions.

2 An Urban Planning System

We think of urban planning to be "dividable" into master urban planning, master_up_beh, and derived urban plannings, $\mathsf{derived\_up\_beh}_i$, where sub-index i indicate that there may be several, i.e., $i \in \{d_1, d_2, ..., d_n\}$, such derived urban plannings. We think of master urban planning to "convert" physical

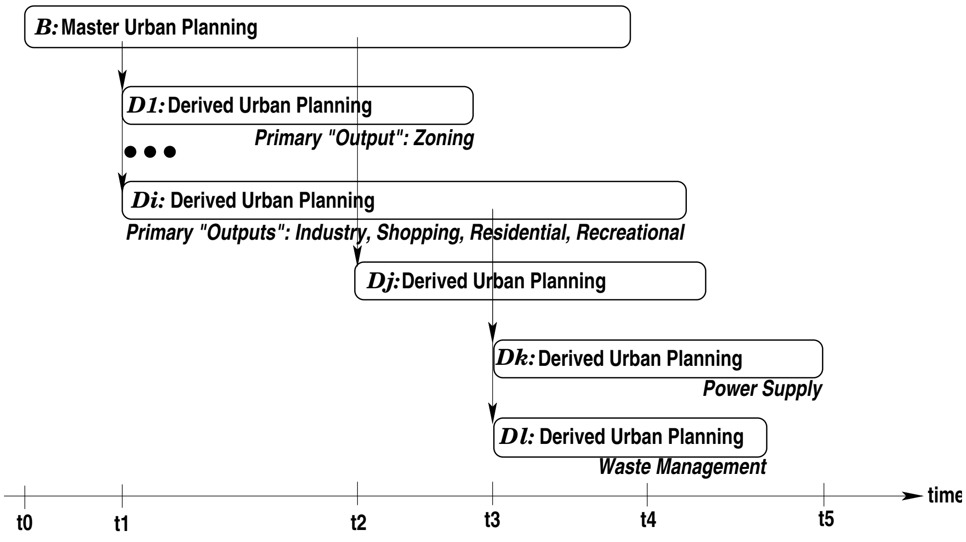

B: Master Urban Planning
D1: Derived Urban Planning
Primary "Output": Zoning
●●●
Di: Derived Urban Planning
Primary "Outputs": Industry, Shopping, Residential, Recreational
Dj: Derived Urban Planning
Dk: Derived Urban Planning
Power Supply
Dl: Derived Urban Planning
Waste Management
time
t0
t1
t2
t3
t4
t5

Figure 1: An Urban Planning Development

(geographic, that is, geodetic, cadestral, geo-technical, meteorological, etc.) information about the land area to be developed into a master plan, that is, cartographic, cadestral and other such information (zoning, etc.). And we think of derived urban planning to "convert" master plans into societal and/or technological plans. Societal and technological urban planning concerns are typical such as *industry zones, commercial (i.e., office and shopping) zones, residential zones, recreational areas, health care, schools, etc.* and *transport, electricity, water, waste, etc.* Each urban planning *behaviour*, whether 'master' or 'derived', is seen as a *sequence* of the application of "the same" urban planning *function*, i.e., an urban planning *action* – but possibly to different goals so that each application (of "the same" urban planning *action*) resolves a sub-goal. Each urban planning action takes a number of information *arguments* and yield information *results*. The master urban planning behaviour may **start** one or more derived urban planning behaviours, $\mathsf{der\_up\_beh}_i$, at the end of "completion" of a base urban planning *action*. Let $\{d_1, d_2, ..., d_n\}$ index separate derived urban plannings, each concerned with a distinct, i.e., reasonably delineated technological and/or societal urban planning concern. During master urban planning actions may start any of these derived urban plannings once. Thus we think of urban planning as a system of a single master urban planning process (i.e., behaviour), master_up_beh, which "spawns" zero, one or more (but a definite number of) derived urban planning processes (i.e., behaviours), $\mathsf{der\_up\_beh}_i$. Derived urban planning processes, $\mathsf{der\_up\_beh}_i$, may themselves start other derived urban planning processes, $\mathsf{der\_up\_beh}_j$, $\mathsf{der\_up\_beh}_k$, ..., $\mathsf{der\_up\_beh}_\ell$. Figure 1 is intended to illustrate the following: At time $t0$ a master urban planning is started. At time $t1$ the master urban planning initiates a number of derived urban development, $D1, ..., Di$. At time $t2$ the master urban planning initiates the Dj derived urban planning. At time $t3$ the derived urban planning Di initiates two derived urban plannings, Dk and $D\ell$. At time $t4$ the master urban planning ends. And at time $t5$ all urban plannings have ended. Urban planning actions are provided with "input" in the form of either geographic, geodetic, geo-technical, meteorological, etc., information, m_geo:mTUS[6], or auxiliary information, m_aux:mAUX, or requirements information, m_req:mREQ. The auxiliary ("management") information is such as *time and date, name (etc.) of information provider, "trustworthiness" of information,* etc. The requirements information serves to direct, to inform, the urban planners towards *what kind of urban plan* is desired.

[6]The m_ value prefixes and the m type prefixes shall designate master urban planning entities.

3 Formalisation of Urban Space and Planning Endurants

3.1 Some Auxiliary Concepts

3.1.1 Points and Areas

6 We shall assume a notion of *the urban space*, tus:TUS, from which we can observe the attribute: an infinite, compact Euclidean set of points.

7 By a *point* we shall understand a further undefined atomic notion.

8 By an *area* we shall understand a concept, related to the urban space, that allows us to speak of *"a point being in an area"* and *"an area being equal to or properly within another area"*.

9 To an[y] *urban space* we can associate an area; we may think of an area being an *attribute* of the urban space.

```
type
6   TUS
value
6   attr_Pts: TUS → Pt-infset
type
7   Pt
8   Area
value
9   attr_Area: TUS → Area
8   is_Pt_in_Area: Pt × (TUS|Area) → Bool
8   is_Area_within_Area: Area × (TUS|Area) → Bool
```

3.1.2 Time and Time Intervals

10 Time is modeled as a continuous entity.

11 One can subtract two times and obtain a time interval.

12 Time intervals are likewise modeled as continuous entities.

13 One can add or subtract a time interval to, resp. from a time and obtain a time.

14 One can compare two times, or two time intervals.

15 One can add and subtract time intervals.

16 One can multiply time intervals with real numbers.

```
type
10   T
11   TI
value
11   sub: T × T → TI
13   add,sub: TI × T → T
13   <,≤,=,≥,>: ((T×T)|(TI×TI)) → Bool
15   add,sub: TI × TI → TI
16  mpy: TI × Real → TI
```

3.2 Urban Space 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.

By *the urban space endurants* we shall here mean the facts by means of which we can characterize that which is subject to urban planning: the land, what is in and on it, its geodetics, its cadastre[7], its meteorology, its socio-economics, its rule of law, etc. As such we shall consider 'the urban space' to be a *part* in the sense of [6]. And we shall consider the *geodetic, cadastral, geotechnical, meteorological*, *"the law"* (i.e., *state, province, city* and *district ordinances*) and *socio-economic* properties as *attributes.*

Left: geodetic map, right: cadastral map.

3.2.1 Main Part and Attributes

One way of observing *the urban space* is presented: to the left, in the framed box, we **narrate** the story; to the right, in the framed box, we **formalise** it.

17 The Urban Space (TUS) has the following
 a Geodetic attributes,
 b Cadastre attributes,
 c Geotechnical attributes,
 d Meteorological attributes,
 e Law attributes,
 f Socio-Economic attributes, etcetera.

```
type
17   TUS, GeoD, Cada, GeoT, Met, Law, SocEco, ...
value
17a  attr_GeoD: TUS → GeoD
17a  attr_Cada: TUS → Cada
17c  attr_GeoT: TUS → GeoT
17d  attr_Met: TUS → Met
17f  attr_SocEco: TUS → SocEco
```

The attr_A: P → A is the **signature** of a postulated *attribute (observer) function.* From parts of type P it **observes** attributes of type A. attr_A are postulated functions. They express that we can always observe attributes of type A of parts of type P.

3.2.2 Urban Space Attributes – Narratives and Formalisation

We describe attributes of the domain of urban spaces. As they are, in real life. Not as we may record them or represent them (on paper or within the computer). We can "freely" model that reality as we think it is. If we can talk about and describe it, then it is so! For meteorological attributes it means that we describe precipitation, evaporation, humidity and atmospheric pressure as these physical phenomena "really" are: continuous over time! Similar for all other attributes. Etcetera.

[7]Cadastre: A Cadastre is normally a parcel based, and up-to-date land information system containing a record of interests in land (e.g. rights, restrictions and responsibilities). It usually includes a geometric description of land parcels linked to other records describing the nature of the interests, the ownership or control of those interests, and often the value of the parcel and its improvements. See http://www.fig.net/

General Form of Attribute Models

18 We choose to model the *General Form of Attributes*, such as geodetical, cadastral, geotechnical, meteorological, socio-economic, legal, etcetera, as [continuous] functions from time to maps from points or areas to the specific properties of the attributes.

19 The points or areas of the properties maps must be in, respectively within, the area of the urban space whose attributes are being specified.

```
type
18  GFA = T → ((Pt|Area) →m Properties)
value
19  wf_GFA: GFA × TUS → Bool
19  wf_GFA(gfa,tus) ≡
19      let area = attr_Area(tus) in
19      ∀ t:T • t ∈ 𝒟 gfa ⇒
19          ∀ pt:Pt • pt ∈ dom gfa(t) ⇒ is_Pt_in_Area(pt,area)
19       ∧ ∀ ar:Area • ar ∈ dom gfa(t) ⇒ is_within_Area(ar,area)
19      end
```

$\mathcal{D}$ is a hypothesized function which applies to continuous functions and yield their domain!

Geodetic Attribute[s]

20 Geodetic attributes map points to

   a land elevation and what kind of land it is; and (or) to
   b normal and current water depths and what kind of water it is.

21 Geodetic attributes also includes road nets and what kind of roads;

22 etcetera,

```
type
20   GeoD = T → (Pt →m ((Land|Water) × RoadNet × ...))
20a  Land = Elevation × (Farmland|Urban|Forest|Wilderness|Meadow|Swamp|...)
20b  Water = (NormDepth × CurrDepth) × (Spring|Creek|River|Lake|Dam|Sea|Ocean|...)
21   RoadNet = ...
22   ...
```

Cadastral Attribute[s] A cadastre is a public register showing details of ownership of the real property in a district, including boundaries and tax assessments.

23 Cadastral maps shows the boundaries and ownership of land parcels. Some cadastral maps show additional details, such as survey district names, unique identifying numbers for parcels, certificate of title numbers, positions of existing structures, section or lot numbers and their respective areas, adjoining and adjacent street names, selected boundary dimensions and references to prior maps.

24 Etcetera.

```
type
23  Cada = T → (Area →m (Owner × Value × ...))
24  ...
```

Geotechnical Attribute[s]

25 Geotechnical attributes map points to

- a top and lower layer soil etc. composition, by depth levels,
- b ground water occurrence, by depth levels,
- c gas, oil occurrence, by depth levels,
- d etcetera.

type
25 GeoT = (Pt $\overrightarrow{m}$ Composition)
25a Composition = VerticalScaleUnit × Composite*
25b Composite = (Soil|GroundWater|Sand|Gravel|Rock|...|Oil|Gas|...)
25c Soil,Sand,Gravel,Rock,...,Oil,Gas,... = [chemical analysis]
25d ...

Meteorological Attribute[s]

26 Meteorological information records, for points (of an area) precipitation, evaporation, humidity, etc.;

- a precipitation: the amount of rain, snow, hail, etc.; that has fallen at a given place and at the time-stamped moment[8], expressed, for example, in milimeters of water;
- b evaporation: the amount of water evaporated (to the air);
- c atmospheric pressure;
- d air humidity;
- e etcetera.

26 Met = T → (Pt $\overrightarrow{m}$ (Precip × Evap × AtmPress × Humid × ...))
26a Precip = MMs [milimeters]
26b Evap = MMs [milimeters]
26c AtmPress = MB [milibar]
26d Humid = Percent
26e ...

Socio-Economic Attribute[s]

27 Socio-economic attributes include time-stamped area sub-attributes:

- a income distribution;
- b housing situation, by housing category: apt., etc.;
- c migration (into, resp. out of the area);
- d social welfare support, by citizen category;
- e health status, by citizen category;
- f etcetera.

[8]– that is within a given time-unit

type
27 SocEco = T → (Area $\xrightarrow[m]{}$ (Inc×Hou×Mig×SoWe×Heal×...))
27a Inc = ...
27b Hou = ...
27c Mig = {|"in","out"|} $\xrightarrow[m]{}$ ({|"male","female"|} $\xrightarrow[m]{}$ (Agegroup × Skills × HealthSumm × ...))
27d SoWe = ...
27e CommHeal = ...
27f ...

Law Attribute[s]: State, Province, Region, City and District Ordinances

28 By the law we mean any state, province, region, city, district or other 'area' ordinance[9].

29 ...

type
28 Law
value
28 attr_Law: TUS → Law
type
28 Law = Area $\xrightarrow[m]{}$ Ordinances
29 ...

Industry and Business Economics

TO BE WRITTEN

Etcetera

TO BE WRITTEN

3.2.3 Discussion

TO BE WRITTEN

3.3 Urban Planning Auxiliaries

By *urban planning auxiliaries* we mean such information that are *not* of geodetic, cadestral, geotechnical, meteorological, etc., nature, that is, are of the land, but are of urban planning project nature: project plan, time & resource schedules, project staffing, project budget, project financing, et cetera.

3.4 Urban Planning Requirements

By *urban planning requirements* we mean such information as expresses what the goal of the urban planning project is, i.e., deliverables, when and where; who provides what information; who consumes which information; and project deliverable acceptance criteria for validation and correctness.

[9]Ordinance: a law set forth by a governmental authority; specifically a municipal regulation: for ex.: *A city ordinance forbids construction work to start before 8 a.m.*

3.5 Urban Planning Endurants

By an *urban planning endurant* we shall understand a tangible document that, as *urban plans*, can either be formally related to urban space endurants, that is, geodetic, cadestral, geotechnic and meteorological documents, or, as *urban planning ancillaries*, can be related to urban planning auxiliary documents.

3.5.1 Urban Plans

By an *urban plan* we mean a document which describes

MORE TO COME

3.5.2 Urban Planning Ancillaries

TO BE WRITTEN

3.6 Assumptions About Urban Space and Planning Attributes

In this section we shall distinguish between assumptions about urban space and planning attributes as these assumptions are concerned with the domain of urban planning. that to the actual, "real world" phenomena of such things as geodesy, cadastre, geo-techniques, meteorology, etc.; and assumptions about urban space and planning attributes as these assumptions are concerned with requirements to the recording of the "real world" phenomena, that is to how the "values" of the phenomena are recorded. Please observe that this report is exclusively about the former. That is, it is not about requirements to the 'data' that may be input to, or output from actual urban planning projects, for example in the form of software data! We shall refer to the former as *assumptions about urban space and planning attribute values*, and to the latter as *assumptions about urban space and planning data*.

3.6.1 Assumptions About Urban Space and Planning Attribute Values

The typical assumptions that we make about geodetical, geotechnical and meteorological phenomena are that their values are *continuous* over *time* and *space*, viewed separately and taken together. From this follows that *other* urban planning attributes derived from, or related to geodetical, geotechnical and meteorological phenomena, are themselves, in some sense (to be defined for each kind of *other* urban planning attribute) also *"continuous"*.

3.6.2 Assumptions About Urban Space and Planning Data

Urban space and planning data are data that are strongly *related* to urban space and planning attribute values. But, being thought of a data, as values input to or resulting from urban planning, they are *discrete*, not continuous. That is, the urban space and planning data are approximate, finite *representations* of continuous phenomena. As such we must be able to formalise the *postulated relations* between the continuous urban space and planning attribute values and the discrete urban space and planning data and these relations must be such that we can likewise formalise a *quality factor:* "how good, or bad, is the data representation with respect to the 'real' domain phenomena" ?

● ● ●

We leave our (hence short) discourse into the two concepts: *assumptions about urban space and planning attribute values*, and *assumptions about urban space and planning data*, bearing in mind that we assume "ideal" properties of the domain attribute values, while leaving assumptions about urban space and planning data to further, more final treatment only when dealing with requirements to software for urban planning.

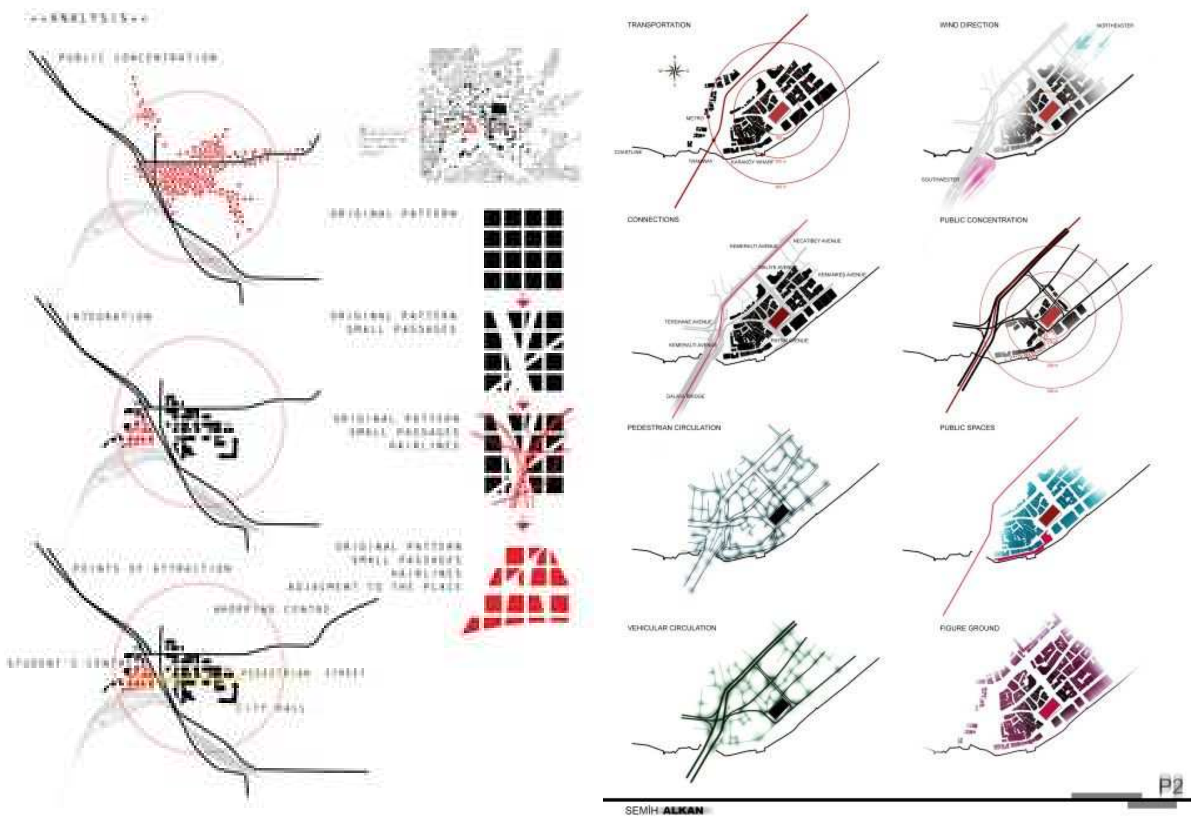
4 Requirements, Goals and Indicators

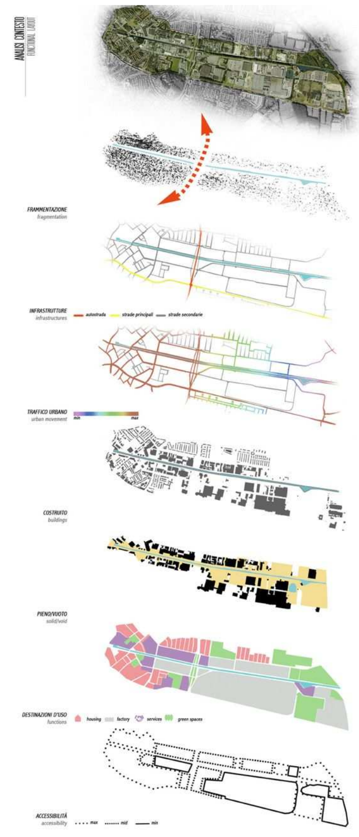
We refer to Sect. 3.6 on the facing page. The term ‘Requirements’ in the present section’s title refer to

It is suggested that this section be co-authored by
Prof. Otthein Herzog[1] and **Prof. Siegfried ZhiQiang Wu**[2]
[1] Jacobs University, Campus Ring 1, 28759 Bremen, Germany
[2] CIUC – Tongji University, Siping Campus, Shanghai, China

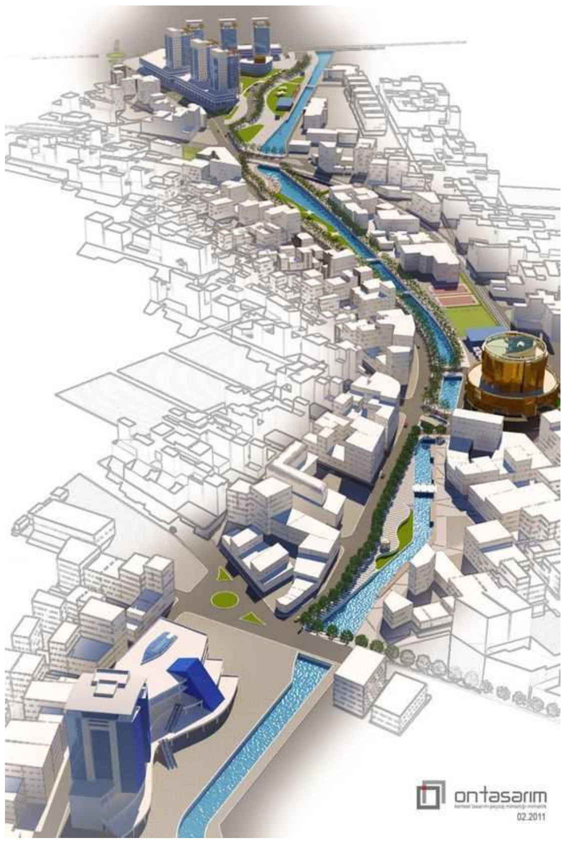
4.1 Example Graphics of Uran Plans

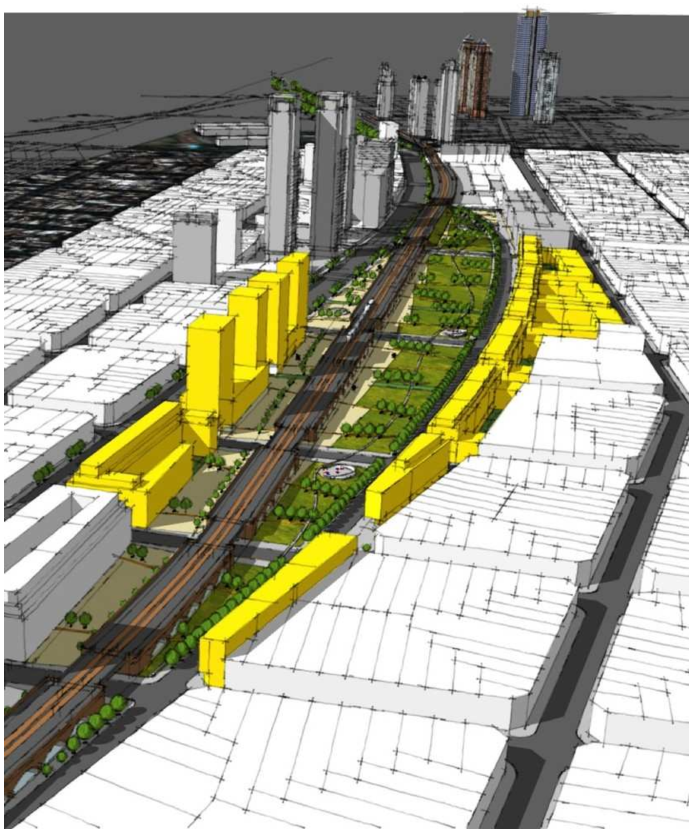
We show some uncommented graphics related to [“small”] urban plans.




FRAMMENTAZIONE
fragmentation
INFRASTRUTTURE
infrastructures
autostrada
strade principali
strade secondarie
TRAFFICO URBANO
urban movement
COSTRUITO
buildings
PIENO/VUOTO
solid/void
DESTINAZIONI D'USO
functions
housing
factory
services
green spaces
ACCESSIBILITÀ
accessibility
max
mid
min

MASTER PLAN 1"=50'
SECTION 1-7 1"=16'


ontasarim
02.2011



5 Master Urban Planning

We begin this section with abstractions of the, perhaps, two most important aspects of urban planning, such as it may be seen by its individual practitioners: the *information* (being handled: the "input", so-to-speak, to urban planning functions) and the urban planning *functions*. In two sections, in-between the *information* and the *function* sections (5.1 and 5.4), we very briefly discuss the *iterative nature* of urban planning, Sect. 5.2 on the next page, and *initial values*, Sect. 5.3, of the various information values.

5.1 Urban Planning Information Categories

5.1.1 "Input"

Among the arguments of urban planning are

30 information, mTSU[10], about *the urban space,* the demo-geographic area subject to planning: its geodetic "make-up", its cadastral, geotechnical and meteorological properties, etc.;

31 related, but not geographic, information, mAUX[iliary][11];

32 and some requirements, mREQ[uirements].

type
30 mTSU
31 mAUX

[10]The m prefix of certain type names, like mTSU, mAUX, mREQ, shall designate the term 'master'.
[11]Auxiliary: giving help or support.

32 mREQ

5.1.2 "Output"

Among results of urban planning are

33 "the plan" (or "plans"), mPLA[ns],

34 and possibly some other ancillary[12] documents, mANC[illerary].

type
33 mPLA
34 mANC

For this and the next sections we shall leave the mTUS, mAUX, and mREQ argument types and the mPLA, and mANC result types further undefined.

5.2 The Iterative Nature of Urban Planning

We take it that urban planning proceeds in "cycles":

35 In each cycle the master urban planning function, master_up_fct, is applied to an input argument triple, (m_tus,m_aux,m_req):(mTUS×mAUX×mREQ):mTRI, of "fresh" geodetic/cadastral/-geotechnical/meteorological (etc.), auxiliary and requirements information.

type
35 mTRI = mTUS × mAUX × mREQ

36 Each cycle, that is, each application of master_up_fct, results in a "most recent", not necessarily "final", plan and ancillary information, (m_pla,m_anc):mPLA×mANC:mRES.

type
36 mRES = mPLA × mANC

37 But, to "drive" the urban planning process, master_up_beh, towards a "final", that is, an adequately satisfactory plan etc., the urban planning function, master_up_fct, *need also be provided with the previous iteration's result* — which we take to be a ("quintuplet"[13], i.e., a) pair of an (i.e., the "previous") "input" triple and the previous result pair.

type
37 mQUI = mTRI × mRES

[12] Ancillary: providing necessary support to the main work
[13] We put double quotes around the term *quintuplet* to indicate that we do not really mean it to be a quintuplet, but, as here, a pair of a triplet and a pair!

We shall refer to the input argument triple as 'the triplet', and to the "driver" quintuplet as a resumption. The above decisions on triplet arguments and quintuplet resumptions, including the latter's "feedback" to a next iteration function invocation is motivated as follows. We think of each invocation, i.e., step, of the urban planning function to "apply" itself to a small fragment of urban planning. Each such "small" step is to result in useful contributions to the evolving urban plan. The ancillary information emerging from each step informs about which aspects of urban planning was pursued in that step: where, in the plans, the outcome of those analysis and plan development can be seen. The reason for small step invocations are to allow ongoing reviews (not shown here), and to pass on intermediary results to other urban planning developments, etc. The decision to "feed" back "records" of the entire state of urban planning development is motivated by the need for these "small step" invocations to analyse the ongoing, full state.

5.3 Initialisation

Urban planning proceeds in iterating from initial

38 urban space, auxiliary and requirements information, as well as

39 (usually "empty") plans and ancillaries.

We extend the notion of initial values to

40 triplet arguments,

41 result pairs, and

42 "quintuplet" argument/result pairs.

towards such results (plans and ancillaries) that are deemed satisfactory.

```
value
38  m_tus_init:mTUS,
38  m_aux_init:mAUX,
38  m_req_init:mREQ
39  m_pla_init: mPLA,
39  m_anc_init:mANC
40  m_tri_init: mTRI = (m_tus_init,m_aux_init,m_req_init)
40                 assert: m_tri_fit(m_tri_init,m_tri_init)
41  m_res_init: mRES = (m_pla_init,m_anc_init)
42  m_qui_init: mQUI = (m_tri_init,m_res_init)
```

We refer to Item 52 on Page 23 for an explanation of the m_tri_fit predicate.

5.3.1 Existing versus Evolving Plans

The quintuplet "feedback", which includes a 'plan' component, secures that possibly pre-existing plans are included, as initialised components of the plan results.[14] The iterative nature of urban planning thus allows for step-wise urban re-development, from existing "urban land-scapes" via mixed "previous" and "future" land-scapes to the final urban development plan.

[14] We refer to Item 30 on Page 19.

5.4 A Simple Functional Form

43 The *master* urban planning *function*, master_up_fct, thus applies to

- (i) a "most recent" triplet of urban space, auxiliary and requirements information, and to
- (ii) a "past quintuplet", a resumption[15], that is, pair of urban space, auxiliary and requirements information as well as a pair of a plan and ancillary information

and yields such a resumption "quintuplet" pair of a triplet and a pair.

To repeat:

44 The application of master_up_fct to such arguments, i.e., master_up_fct(m_tus,m_aux,m_req)(m_qui) yields a "quintuplet" result, a resumption, m_qui:((m_tus′,m_aux′,m_req′),(m_pla,m_anc)).

We "explain" the relations between "input" arguments and "output" (**as**) results:

45 The "input" argument m_tri is "carried forward", m_tri′ (=m_tri), to be redeposited as part of the result.

46 The main part of the result, (m_pla,m_anc), is related, $\mathcal{P}_{\text{master}}$, to the input argument including the previous "result", the resumption.

```
43  master_up_fct: mTRI → mQUI → mQUI
44  master_up_fct(m_tri)(m_qui) as (m_tri′,(m_pla,m_anc))
45      m_tri = m_tri′ ∧
46      $\mathcal{P}_{master}$(m_tri)(m_qui)(m_pla,m_anc)
```

For the time being we shall leave the master urban planning function, master_up_fct, that is, $\mathcal{P}_{\text{master}}$, uninterpreted. Of course, *"all the tricks of urban planning are 'hidden' in $\mathcal{P}_{master}$"*.

5.5 Oracles and Repositories

Oracles are simple behaviours that *offer* information to other behaviours. Repositories are simple behaviours that *store* information from behaviours and *offer* stored information to behaviours.

5.5.1 The Master 'Input' Oracle

An urban planning oracle, when so requested, will select some information – usually in some non-deterministic fashion, and usually subject to some constraint – and present this information to the requestor, i.e., an urban planning behaviour. In this section, i.e. Sect. 5.5.1, we shall deal with one specific oracle, m_tri_beh: one that "assembles" triplets, m_tri, of urban space, m_tus:mTUS, auxiliary, m_aux:mAUX, and requirements, m_req:mREQ, information to requesting behaviours. We introduce a pair of specification components:

47 a *channel*, m_tri_ch, over which a master urban planning behaviour, master_up_beh, offers to receive triplets, m_tri:mTRI, from an oracle, m_tri_beh,

48 and an *oracle*, m_tri_beh, which "remembers" its most recently communicated triplet[16].

[15]Resumption: like a repetition, a continuation
[16]The oracle is initialised with b_tri_beh(m_geo_init,m_aux_init,m_req_init).

```
channel
47  m_tri_ch:mTRI
value
48  m_tri_beh: mTRI → out m_tri_ch  Unit
```

49 The oracle assembles (m_tri′:mTRI), a master triplet which satisfies a predicate m_tri_fit(m_tri,m_tri′) – see Item 52.

50 That triplet is offered, m_tri_ch ! m_tri′, to the master urban behaviour –

51 whereupon the oracle resumes being the oracle, now, however, with the recently assembled master triplet as its resumption.

```
48  m_tri_beh(m_tri) ≡
49     let m_tri′:mTRI • m_tri_fit(m_tri,m_tri′) in
50     m_tri_ch ! m_tri′ ;
51     m_tri_beh(m_tri′)
48     end
53  pre: m_tri_fit(m_tri,m_tri)
```

52 The fitness predicate, m_tri_fit(m_tri,m_tri′), checks whether a “newly” assembled master triplet, m_tri′, stands in some suitable[17] relation $\mathcal{P}$(m_tri,m_tri′) to a a similar (f.ex., “earlier”) master triplet, m_tri.

53 The fitness predicate holds for m_tri_fit(m_tri,m_tri).

```
52  m_tri_fit: mTRI × mTRI → Bool
52  m_tri_fit(m_tri,m_tri′) ≡ 𝒫(m_tri,m_tri′)
```

54 The oracle, m_tri_beh, is initialised with an initial triplet value m_tri_init, cf. formula Item 40 on Page 21.

```
52  m_tri_beh(m_tri_init): assert: m_tri_fit(m_tri_init,m_tri_init)
```

5.5.2 The Master Resumption Repository

The “quintuplet” pair of an “input” triple and a result pair, m_qui: (m_tri:mTRI, (m_pla:mPLA, m_anc:mANC)) is thought of as residing in a *repository* behaviour, m_qui_beh, which (m_qui_ch ?) “receives” “quintuplets” from the urban planning behaviour, or “offers” (m_qui_ch ! m_qui) such to the urban planning behaviour.

55 There is therefore a channel, m_qui_ch, between the urban planning behaviour and the “quintuplet” repository behaviour,

56 m_qui_beh.

57 It either

[17]– to be defined for each specific urban planning project

58 accepts or

59 offers quintuplets.

```
channel
55   m_qui_ch:mQUI
value
56   m_qui_beh: mQUI → in,out m_qui_ch  Unit
56   m_qui_beh(m_qui) ≡
58      m_qui_beh(m_qui_ch?)
57      ⌈⌉
59      m_qui_ch!(m_qui) ; m_qui_beh(m_qui)
```

5.6 A Simple Behavioural Form

Urban planning, however, is a time-consuming "affair". So we model it as a behaviour.

60 The master_up_beh_0[18] behaviour takes no argument, hence the left signature element: **Unit**, avails itself of the input channel for obtaining proper input, m_tri, and m_qui, for the master urban function, master_up_fct, and output channel, for depositing a resumption, m_qui′, and (then) "goes on forever", as indicated by the right signature element: **Unit**.

```
60   master_up_beh_0: Unit → in m_tri_ch in,out m_qui_ch Unit
```

61 The simple (version of the) master_up_beh_0 behaviour

62 obtains the master triplet, m_tri, and the master resumption, m_qui, information,

63 performs the master_up_fct planning function and

64 provides its result, a resumption, m_qui′, to the master quintuplet repository,

65 whereupon it reverts to being master_up_beh_0.

```
value
61  master_up_beh_0() ≡
62     let (m_tri,m_qui) = (m_tri_ch?,m_qui_ch?) in
63     let m_qui′ = master_up_fct(m_tri)(m_qui) in
64     m_qui_ch ! m_qui′ end end ;
65     master_up_beh_0()
```

The master_up_beh_0 behaviour repeatedly "performs" urban planning, "from scratch", as if new urban space, auxiliary and requirements information was "new" in every re-planning — "ad infinitum"! We now revise master_up_beh_0 into master_up_beh_1 — a behaviour "almost" like master_up_beh_0, but one which may terminate.

66 master_up_beh_1

67 first behaves like master_up_beh_0 (Items 62–64)

[18] As there will be several versions, from simple towards more elaborate, of the master_up_beh behaviour, we index them.

68 then checks whether the obtained master resumption is satisfactory, that is, is OK as an end-result of master urban planning.

69 If so then master_up_beh_1 terminates,

70 else it resumes being master_up_beh_1.

```
value
60   master_up_beh_1: Unit → in m_tri_ch in,out m_qui_ch Unit
66   master_up_beh_1() ≡
62     let (m_tri,m_qui) = (m_tri_ch?,m_qui_ch?) in
63     let m_qui′ = master_up_fct(m_tri)(m_qui) in
64     m_qui_ch ! m_qui′ ;
68     if master_qui_satisfactory(m_qui′)
69         then skip
70         else master_up_beh_1() end
66     end end

68   master_qui_satisfactory: mQUI → Bool
```

The m_qui_satisfactory predicate inquires the master quintuplet, m_qui, as for its suitability as a final candidate for an urban plan[19].

6 Derived Urban Plannings

6.1 Preliminaries

It is a conjecture that urban planning can be "divided" into master urban planning and derived urban plannings.

MORE TO COME

6.1.1 Derived Urban Plan Indices

We think of *master* urban planning function, modeled by master_up_fct, as being concerned with the overall "division" of the urban space (i.e., geographical area, land and water) into zones for building, recreation, and other (i.e., the *master plan*). Aggregations of these zones, one, more or all (usually several), can then be further "[derive] planned" into zones:

- (d_1) light, medium and heavy industry,
- (d_2) public works,
- (d_3) office,
- (d_4) mixed shopping and residential,
- (d_5) apartment bldg.,
- ..., etc.,
- (d_{m-1}) villa, and
- (d_m) recreational.

Additional forms of derived plannings are:

[19]The m_qui_satisfactory argument, m_qui, embodies not only that plan, but also the basis for its determination.

- (d_{m+1}) transport;
- (d_{m+2}) electricity supply;
- (d_{m+3}) water supply;
- (d_{m+4}) waste management;
- (d_{m+5}) health care;
- (d_{m+6}) fire brigades;
- . . . , etc.,
- (d_n) schools.

We refer to the d_i's as derived urban plan indices.

71 We think of this variety of "derived" plannings as indexed such as hinted at above,

72 and dups as the set of all indices.

```
type
71    DP == {|d$_1$,d$_2$,...,d$_n$|}
value
72    dups:DP-set = {d$_1$,d$_2$,...,d$_n$}
```

6.1.2 A "Reservoir" of Derived Urban Planning Indices

73 To secure that at most one derived planning, d_i, is initiated we introduce a global variable, dps_var, initialised to an empty set of derived planning tokens and updated with the addition of selected DP tokens.

```
variable
73    dps_var:DP-set := {} comment dps_var denotes a reference
```

6.1.3 A Derived Urban Planning Index Selector

74 A function, sel_dps, selects zero, one or more "fresh" DP indices, that is, DP tokens that have not been selected before.

```
value
74    sel_dps: Unit → DP-set
74    sel_dps() ≡
74        let dps:DP-set•dps⊆dups \ c dps_var in
74        dps_var := c dps_var ∪ dps; dps end
comment
73    [ c denotes a contents-taking operator ]
```

We shall revise the above selector in Sect. 6.7 on Page 32.

6.1.4 The Derived Urban Plan Generator

75 We therefore edit the master_up_beh_1 behaviour slightly into the revised master_up_beh_2. In master_up_beh_2 we insert, "in parallel" (‖) with the "resumption" of master_up_beh_2 (cf. Item 70 on the previous page), an internal non-deterministic choice behaviour, der_up(). It specifies the selection of zero, one of more DP tokens, and initiates corresponding derived planning behaviours, der_up_beh$_i$(), as well as their corresponding "input" triplet oracles, d_tri_beh$_i$(). But only at most once. These derived planning behaviours, der_up_beh$_i$, and "input" triplet oracles, der_tri_beh$_i$() are like master_up_beh_1, respectively m_tri_beh, only now they are "tuned" to the specific derived planning issues (i.e., $_i$).

When behaviour and function invocations where the names of these behaviors or functions names are prefixed with der_, e.g., der_name, and are indexed by some $_i$, i.e., der_name$_i$, then we mean the invocation of one specific i indexed behaviour or function from the indexed set of such, as defined by their behaviour and function definitions, see below.

```
value
75  der_up: Unit → Unit
75  der_up() ≡ let dps = sel_dps() in ‖{der_up_beh_i()‖d_tri_beh_i()|i:DP•i ∈ dps} end
```

We shall introduce the der_up_beh$_i$ and d_tri_beh$_i$ behaviours below.

6.1.5 The Revised Master Urban Planning Behaviour

We "take over", i.e., "copy", the basic structure and definition ("contents") of the urban planning function and behaviour from that of the *master* version. that is: master_up_beh_2().

76 We think of zero, one or more derived plannings (der_up_beh$_1$, der_up_beh$_2$, ..., der_up_beh$_n$) being initiated after some stage of *master* function, master_up_fct, has concluded.

```
value
66   master_up_beh_2() ≡
62       let (m_tri,m_qui) = (m_tri_ch?,m_qui_ch?) in
63       let m_qui′ = master_up_fct(m_tri)(m_qui) in
64       m_qui_ch ! m_qui′ ;
68       if master_satisfactory(m_qui′)
69           then skip
70′          else der_up() ‖ master_up_beh_2() end
66       end end
```

6.2 The Derived Urban Planning Functions

An important form of information for each derived urban planning function is the resumption, i.e., the "quintuplet" information from the master urban behaviour: mQui.

77 The new forms of information are: the derived urban planning auxiliary, dAUX$_i$, the derived urban planning requirements information, dREQ$_i$, as well as the derived urban planning plans, dPLA$_i$, and their ancillary information, dANC$_i$.

78 The primary arguments for the derived urban planning function, master_up_fct, is therefore a "quintuplet", d_qui:dQUI, of a master triplet, m_tri:mTRI, and the pair of the derived urban planning auxiliary information, d_aux$_i$:dAUX$_i$, and the derived urban planning requirements, d_req$_i$:dREQ$_i$.

The result of derived urban planning function, der_up_fct$_i$, as for the master urban planning function, master_up_fct,

79 is that of a "quintuplet", also referred to as a resumption, dQUI$_i$, of the primary arguments, b_tri:bTRI, and

80 the result, a pair of a derived plan, d_pla$_i$, and derived ancillaries, d_anc$_i$.

81 As for the master urban planning function, master_up_fct, it has a secondary, derived "quintuplet" argument (which, as for master_up_fct, helps "kick-start" urban planning). This second argument is the result of a previous application of the der_up_fct_i.

82 The derived urban planning function der_up_fct_i signature is therefore that of a function from a triplet of a most recent master "quintuplet", derived urban planning auxiliary and derived urban planning requirements information to functions from derived "quintuplet" arguments to derived "quintuplet" results.

83 The triplet argument, d_tri_i, and the first part of the result, also a triplet, $\text{d\_tri}'_i$, are the same.

84 The derived urban planning function der_up_fct_i is further characterised by a predicate, $\mathcal{P}_{\text{der}_i}$, which we leave further undefined.

```
type
77  dAUX$_1$, dAUX$_2$, ..., dAUX$_n$
77  dREQ$_1$, dREQ$_2$, ..., dREQ$_n$
77  dPLA$_1$, dPLA$_2$, ..., dPLA$_n$
77  dANC$_1$, dANC$_2$, ..., dANC$_n$
78  dTRI$_i$ = mQUI×dAUX$_i$×dREQ$_i$     [i:DP•i∈dups]
80  dRES$_i$ = dPLA$_i$ × dANC$_i$         [i:DP•i∈dups]
79  dQUI$_i$ = dTRI$_i$×dRES$_i$           [i:DP•i∈dups]
value
81  der_up_fct$_i$: dTRI$_i$ → dQUI$_i$m → dQUI$_i$  i:DP
82  der_up_fct$_i$(d_tri$_i$)(d_qui$_i$) as (d_tri$'_i$,d_res$_i$)
83      d_tri$_i$ = d_tri$'_i$ ∧
84      $\mathcal{P}_{\text{der}_i}$(d_tri$'_i$,d_res$_i$)

84  $\mathcal{P}_{\text{der}_i}$: dTRI$_i$ × dRES$_i$ → Bool
```

6.3 The Derived Urban Planning "Oracle" Behaviour

85 We introduce the (indexed) type, dPAIR_i, of a *pair* of indexed derived auxiliaries and indexed derived requirements.

86 And we need an array channel that communicates the master quintuplet from the master quintuplet repository to an indexed derived triplet behaviour.

The d_tri_beh_i oracle evolves around:

87 the most recent *dpair*; it inputs master quintuplets over the master quintuplet repository to derived triplet oracle channel; and it outputs a triplet, dTRI_i, to the drived urban planning behaviour.

88 The d_tri_beh_i behaviour "cycles" between forming (internal non-deterministically) a suitable, a *fit*, "next" *dpair* which it combines,

89 mq_dt_ch[i]? offers to the derived urban planning behaviour

90 whereupon it resumes being an oracle.

91 We leave the definition of *fit*ness of *dpair*s open.

```
type
85   dPAIR_i = dAUX_i × dREQ_i [i:DP•i∈dups]
channel
86   {mq_dt_ch[i]|i:DP}:mQUI
value
87   d_tri_beh_i: dPAIR_i → in mq_dt_ch[i] out d_tri_ch[i] Unit
87   d_tri_beh_i(d_pair_i) ≡
88     let d_pair'_i:dPAIR_i • d_fit_pair(d_pair_i,d_pair'_i) in
89     d_tri_ch[i] ! (mq_dt_ch[i]?,d_pair'_i) ;
90     d_tri_beh_i(d_pair'_i) end

91   d_fit_pair: dPAIR_i × dPAIR_i → Bool
```

6.4 The Derived Urban Planning Behaviour

92 We think of zero, one or more derived plannings (der_up_beh$_{i_1}$, der_up_beh$_{i_2}$, ..., der_up_beh$_{i_k}$) being initiated after some stage of the *der_up_fct$_i$* function has concluded.

```
value
66   der_up_beh_i() ≡
62     let (d_tri_i,d_qui_i) = (d_tri_ch[i]?,d_qui_ch[i]?) in
63     let d_qui' = der_up_fct_i(d_tri_i)(d_qui_i) in
70     d_qui_ch[i] ! d_qui'_i ;
68     if der_qui_satisfactory_i(d_qui'_i)
66        then skip
67,92     else der_up() ∥ der_up_beh_i()  end
66     end end

68   der_qui_satisfactory: dQUI → Bool
```

6.5 The Derived Resumption Repository

6.5.1 The Consolidated Derived Resumption Map

93 The derived urban planning functions (and thus behaviours) operate, not on simple resumptions, as do the master urban planning functions (and behaviours), but on the aggregation of all derived functions' (etc.) "quintuplets", that is, an indexed set of "quintuplets" – modeled as a derived resumptions map.

```
type
93   dQUIm = DP →m dQUI_i
```

6.5.2 The Consolidated Derived Resumption Repository Channel

94 Communications between the individual derived urban planning behaviours and the consolidated derived resumption repository are via an indexed set of channels communicating derived resumptions maps.

```
channel
94   {d_qui_ch[i]:dQUIm|i:DP• i ∈ dups}
```

6.5.3 The Consolidated Derived Resumption Repository

95 The consolidated derived resumption repository behaviour either ($\lceil\rceil$) updates its state map with received individual derived resumptions, or offers the entire such state maps to whichever derived urban planning behaviour so requests.

```
value
95  d_qui_beh: dQUIm → in,out der_qui_ch[ i ]  Unit  i:DP
95  d_qui_beh(d_qui_m) ≡
95      (⌈⌉{d_qui_beh(d_qui_m†[ i↦d_qui_ch[ i ]? ])|i:DP•i ∈ dup}
95      ⌈⌉
95       ⌈⌉{d_qui_ch[ i ]!d_qui_m|i:DP•i ∈ dup});
95      d_qui_beh(d_qui_m)
```

6.5.4 Initial Consolidated Derived Urban Plannings

value
  init_d_qui$_1$:dQUI$_1$, ..., init_d_qui$_n$:dQUI$_n$
  init_d_qui_m:dQUIm = [d$_1$ ↦ init_d_qui$_1$, ..., d$_n$ ↦ init_d_qui$_n$]

6.5.5 Initialisation of The Derived "Quintuplet" Oracle

As for master oracle and repository behaviours we initialise the derived "quintuplet" oracle:

  der_qui_beh(init_d_qui_m)

6.6 A Visual Rendition of Urban Planning Development

The urban planning project domain, when operating at "full speed", consists of the master urban planning behaviour (i.e., project), zero, one or more derived urban planning behaviours, each of the latter initiated by either the master urban planning project or a derived urban planning project. See Fig. 2 on the facing page. The planning behaviours, both the master and the deriveds, invoke respective urban planning functions, and these produce, such as we have modeled them, "quintuplets" of information, which are deposited with respective "quintuplet" repository behaviours: the master "quintuplet" repository behaviour, and the derived "quintuplet" repository behaviour — which maintains these "quintuplets" for all (invoked and thus ongoing) derived urban planning projects. We kindly ask you to review Fig. 2. All you have to grasp is the fact that there is one master urban planning project, with its repository of master urban planning "quintuplets", and between 0 and n derived urban planning projects, with their shared (consolidated[20]), derived urban planning "quintuplets", Then there are the channels: the query (input) channels providing auxiliary and requirements information to both the one master urban planning project and the n derived urban planning projects; and the query/repository channels providing "quintuplet" aggregated information to the master urban planning project, as well as "quintuplet" aggregated information to the derived urban planning projects. Finally there is the "global" value representing the index set of derived urban planning indices, and the variable which holds the index set of derived urban planning indices of ongoing derived urban planning projects.

[20]– into a map

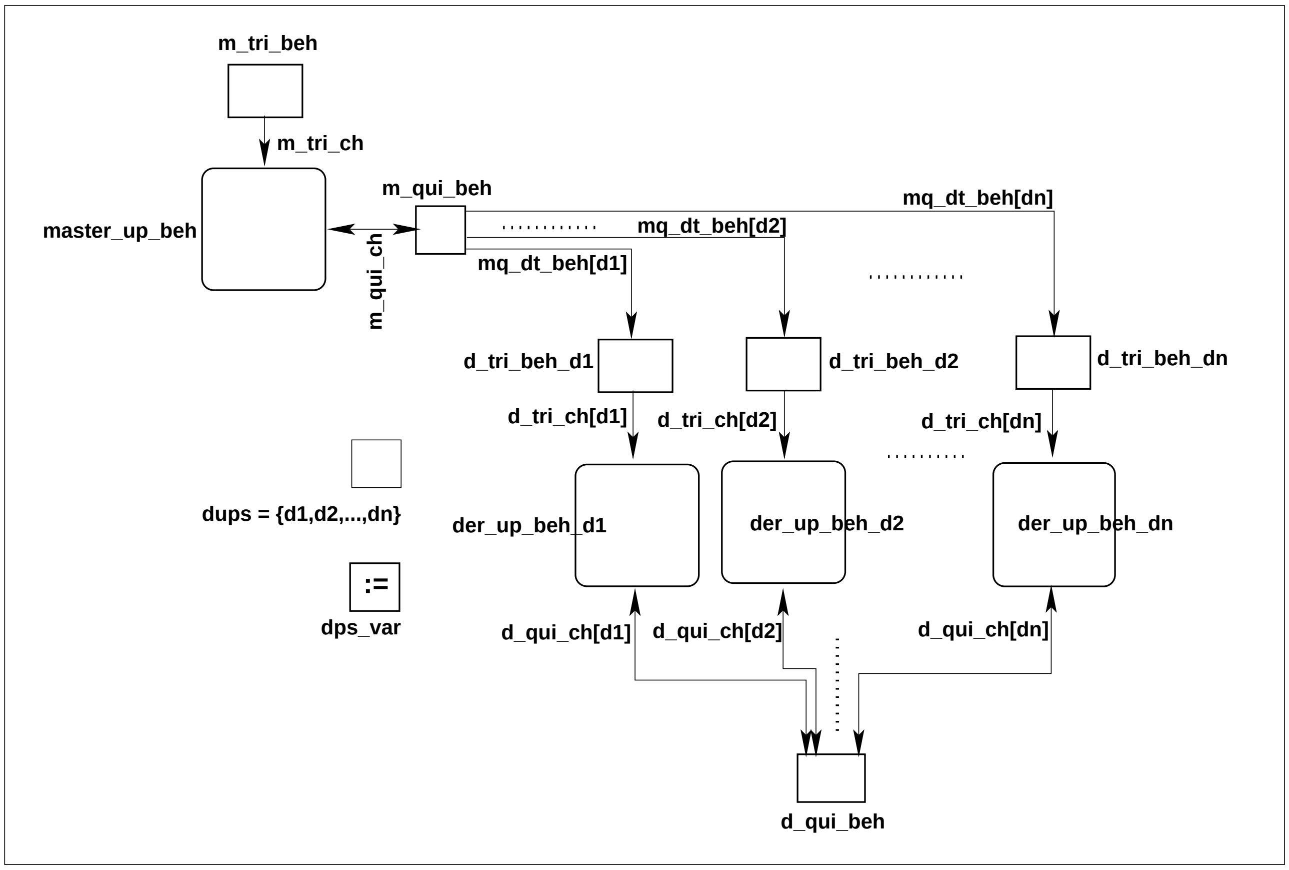


Figure 2: An Urban Planning:
n+1 Planning Behaviours, 2 Repository Behaviours, n+1 Oracles,
a Variable, a Value and 3n+2 Channels

6.7 Revised Selection of Derived Urban Plannings

TO BE WRITTEN

6.8 The Urban Planning System

96 Finally we can define an urban planning development as a system of concurrent behaviours:

- the master urban planning behaviour,
- the master "quintuplet" repository and
- the derived and consolidate "quintuplet" repository

value
96 up_sys: **Unit** $\rightarrow$ **Unit**
96 up_sys() $\equiv$ master_up_beh() $\|$ m_qui_beh(m_qui_init) $\|$ d_qui_beh(init_d_qui_m)

Recall that the derived urban planning behaviours as well as the derived triplet behaviours are started by the master as well as the derived urban planning behaviours.

7 Further Work

7.1 Requirements to Urban Planning

In this section we list a few requirements that we think it wise to have fulfilled before a proper urban planning development project can commence.

MORE TO COME

8 Conclusion

TO BE WRITTEN

9 Bibliograhy

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

[2] 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.

[3] 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.

[4] Dines Bjørner. From Domain Descriptions to Requirements Prescriptions – A Different Approach to Requirements Engineering. 2016. Extensive revision of [3].

[5] Dines Bjørner. What are Documents? Research Note, July 2017. http://www.imm.dtu.dk/~dibj/2017/docs/docs.pdf.

[6] Dines Bjørner. Manifest Domains: Analysis & Description. *Formal Aspects of Computing*, 29(2):175–225, March 2017. DOI 10.1007/s00165-016-0385-z http://link.springer.com/article/-10.1007/s00165-016-0385-z.

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

[8] Dines Bjørner and Cliff B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.

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

[10] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.

[11] 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.

[12] 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.

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

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

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

A A Document System

A.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.

A.2 A Document Systems Description

This research note unravels its analysis &[21] description in stages.

A.3 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[22] 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.

A.4 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 [6, Manifest Domains: Analysis & Description]. We begin by identifying the principal endurants.

97 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.

98 From aggregates of handlers (i.e., M) we can observe sets of handlers (i.e., H).

99 From aggregates of documents (i.e., A) we can observe sets of documents (i.e., D).

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

[22]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'.

```
type
97   S, M, A
value
97   obs_M: S → M
97   obs_A: S → A
type
98   H, Hs = H-set
99   D, Ds = D-set
value
98   obs_Hs: M → Hs
99   obs_Ds: A → Ds
```

A.5 Unique Identifiers

The notion of unique identifiers is treated, at length, in [6, Manifest Domains: Analysis & Description].

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

101 These can be observed from respective parts[23].

```
type
100   MI[24], AI[25], HI, DI
value
101   uid_MI[26]: M → MI
101   uid_AI[27]: A → AI
101   uid_HI: H → HI
101   uid_DI: D → DI
```

As reasoned in [6, 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.

A.6 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".*[28] 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.

A.6.1 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).

[23] [6, 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.
[24] We shall not, in this research note, make use of the (one and only) management identifier.
[25] We shall not, in this research note, make use of the (one and only) archive identifier.
[26] Cf. Footnote 24: hence we shall not be using the uid_MI observer.
[27] Cf. Footnote 25: hence we shall not be using the uid_AI observer.
[28] From: https://en.wikipedia.org/wiki/Document

A.6.2 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.[29] 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.

A.6.3 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: *"January 3, 2018: 09:32 am: This is version V."*, *"This document was released on January 3, 2018: 09:32 am."*, *"January 3, 2018: 09:32 am: Section X.Y.Z of version III was deleted."*, *"January 3, 2018: 09:32 am: References to documents* doc_i *and* doc_j *are inserted on Pages* p *and* q*, respectively."* and *"January 3, 2018: 09:32 am: Final release."*

A.6.4 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*.

A.6.5 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.

A.6.6 A Summary of Document Attributes

102 As separate attributes of documents we have document descriptors, document annotations, document contents and document histories.

103 Document annotations are lists of document notes.

104 Document histories are lists of time-stamped document operation designators.

105 A document operation designator is either a create, or an edit, or a read, or a copy, or a shred designator.

106 A create designator identifies

[29] 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.

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.

107 An edit designator identifies a handler, a time, and specifies a pair of edit/undo functions.

108 A read designator identifies a handler.

109 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.

110 A shred designator identifies a handler.

111 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.

112 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.

113 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
102    DD, DA, DC, DH
value
102    attr_DD: D → DD
102    attr_DA: D → DA
102    attr_DC: D → DC
102    attr_DH: D → DH
type
103    DA = DN*
104    DH = (TIME × DO)*
105    DO == Crea | Edit | Read | Copy | Shre
106    Crea :: (HI × TIME) × (DI-set × Info) × DN × {|"empty_DC"|}
106(b)i   Info = ...
value
106(b)ii   embed_DIs_in_DD: DI-set × Info → DD
axiom
106d   "empty_DC" ∈ DC
type
107    Edit :: (HI × TIME) × (EDIT × UNDO)
108    Read :: (HI × TIME) × DI
109    Copy :: (HI × TIME) × DI × DN
110    Shre :: (HI × TIME) × DI
111    EDIT = (DA × DN × DC) → (DA × DC)
112    UNDO = (DA × DC) → (DA × DN × DC)
axiom
```

```
113  ∀ mkEdit(_,(e,u)):Edit •
113     ∀ (da,dn,dc):(DA×DN×DC) •
113        u(e(da,dn,dc))=(da,dn,dc)
```

A.7 Behaviours: An Informal, First View

In [6, 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

  114 of the system core behaviour,

  115 of the handler aggregate (the management) behaviour

  116 and the document aggregate (the archive) behaviour,

  with the (distributed) parallel composition of

  117 all the behaviours of handlers and,

  the (distributed) parallel composition of

  118 at any one time, zero, one or more behaviours of documents.

- To express the latter

  119 we need introduce two "global" values: an indefinite set of handler identifiers and an indefinite set of document identifiers.

```
value
119  his:HI-set, dis:DI-set

114     sys(...)
115  ‖ mgtm(...)
116  ‖ arch(...)
117  ‖ ‖{hdlr_i(...)|i:HI•i∈his}
118  ‖ ‖{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 115–118, 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 118. 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.

A.8 Channels, A First View

Channels are means for behaviours to synchronise and communicate values (such as unique identifiers, mereologies and attributes).

120 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
120 mgtm_arch_ch:MA

121 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
121 $\{$mgtm_hdlr_ch[i]:MH$|$i:HI•i $\in$ his$\}$

122 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
122 $\{$arch_docu_ch[j]:AD$|$h:DI•j $\in$ dis$\}$

123 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[30], 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
123 $\{$hdlr_docu_ch[i,j]:HD$|$i:HI,j:DI•i $\in$ his$\wedge$j $\in$ dis$\}$

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

type
124 MA, MH, AD, HD

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

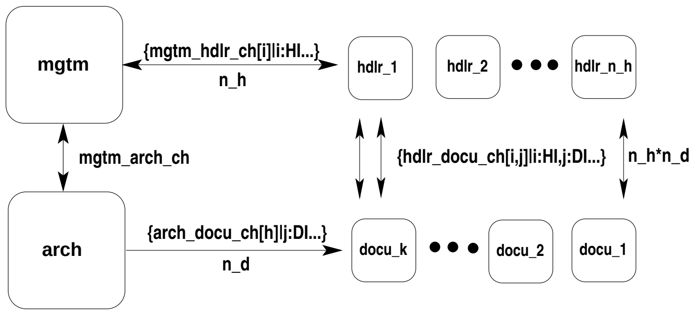

mgtm
{mgtm_hdlr_ch[i]|i:HI...}
n_h
hdlr_1
hdlr_2
hdlr_n_h
mgtm_arch_ch
{hdlr_docu_ch[i,j]|i:HI,j:DI...}
n_h*n_d
arch
{arch_docu_ch[h]|j:DI...}
n_d
docu_k
docu_2
docu_1

Figure 3: An Informal Snapshot of System Behaviours

A.9 An Informal Graphical System Rendition

Figure 3 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

A.10 Behaviour Signatures

125 The mgtm behaviour (synchronises and) communicates with the archive behaviour and with all of the handler behaviours, hdlr$_i$.

126 The archive behaviour (synchronises and) communicates with the mgtm behaviour and with all of the document behaviours, docu$_j$.

127 The signature of the generic handler behaviours, hdlr$_i$ expresses that they [occasionally] receive "orders" from management, and otherwise [regularly] interacts with document behaviours.

128 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

```
125 mgtm: ... → in,out mgtm_arch_ch, {mgtm_hdlr_ch[i]|i:HI•i ∈ his} Unit
126 arch: ...  → in,out mgtm_arch_ch, {arch_docu_ch[j]|j:DI•j ∈ dis} Unit
127 hdlr_i: ...  → in mgtm_hdlr_ch[i], in,out {hdlr_docu_ch[i,j]|j:DI•j∈dis} Unit
128 docu_j: ...  → in mgtm_arch_ch, in,out {hdlr_docu_ch[i,j]|i:HI•i ∈ his} Unit
```

A.11 Time

A.11.1 Time and Time Intervals: Types and Functions

129 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).

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

131 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.

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

```
type
129  TIME
130  TIME_INTERVAL
value
131  add: TIME_INTERVAL × TIME → TIME
131  sub: TIME × TIME → TIME_INTERVAL
131  mpy: TIM_INTERVALE × Nat → TIME_INTERVAL
132  <,≤,=,≠,≥,>: ((TIME×TIME)|(TIME_INTERVAL×TIME_INTERVAL)) → Bool
```

A.11.2 A Time Behaviour and a Time Channel

133 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

134 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.

135 We also postulate a channel, time_ch, on which the time behaviour offers time values to whoever so requests.

```
value
133   time: TIME → time_ch  TIME  Unit
133   time(t) ≡ (time(t) ⌈⌉ time(t+ti) ⌈⌉ time_ch!t ; time(t))
134   ti:TIME_INTERVAL ...
channel
135   time_ch:TIME
```

A.11.3 An Informal RSL Construct

The formal-looking specifications of this report appear in the style of the RAISE [12] Specification Language, RSL [11]. 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.

A.12 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.[31] 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**

136 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. A.13.3 on Page 44). Initially that management directory lists a number of handlers, by their identifiers, but with no granted documents.

137 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.

138 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.

139 Document attributes we mentioned above, cf. Items 102–105.

type
136 MDIR = HI $\overrightarrow{m}$ (DI $\overrightarrow{m}$ ANm-**set**)
137 ADIR = avail:DI-**set** × used:DI-**set** × gone:DI-**set**
138 HDIR = DI $\overrightarrow{m}$ ANm-**set**
139 SDATR = DD, PDATR = DA × DC × DH
axiom
137 ∀ (avail,used,gone):ADIR • avail ∩ used = {} ∧ gone ⊆ used

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

[31] We refer to Sect. 3.4 of [6], and in particular its subsection 3.4.4.

value
125 mgtm: MDIR $\rightarrow$ **in**,**out** mgtm_arch_ch, {mgtm_hdlr_ch[i]|i:HI•i $\in$ his} **Unit**
126 arch: ADIR $\rightarrow$ **in**,**out** mgtm_arch_ch, {arch_docu_ch[j]|j:DI•j $\in$ dis} **Unit**
127 hdlr_i: HDIR $\rightarrow$ **in** mgtm_hdlr_ch[i], **in**,**out** {hdlr_docu_ch[i,j]|j:DI•j$\in$dis} **Unit**
128 docu_j: SDATR $\rightarrow$ PDATR $\rightarrow$ **in** mgtm_arch_ch, **in**,**out** {hdlr_docu_ch[i,j]|i:HI•i $\in$ his} **Unit**

A.13 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.

A.13.1 Management Messages with Respect to the Archive

140 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 106 on Page 36:

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
106. Crea :: (HI $\times$ TIME) $\times$ (DI-**set** $\times$ Info) $\times$ DN $\times$ {|"empty_DC"|} [cf. formula Item 106, Page 37]

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

141 Copy :: DI $\times$ HI $\times$ TIME $\times$ DN [cf. Item 151 on Page 45]

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

type
142 Shred :: TIME $\times$ DI

A.13.2 Management Messages with Respect to Handlers

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

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

144 the management behaviour decides to **grant** access rights, acrs:ACRS[32], to a document handler, hi:HI.

type
144 Gran :: HI × TIME × DI × ACRS

A.13.3 Document Access Rights

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

145 By document access rights we mean a set of action names.

146 By an action name we mean such tokens that indicate either of the document handler operations indicate above.

type
145 ACRS = ANm-**set**
146 ANm = {|″edit″,″read″,″copy″|}

A.13.4 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.

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

type
147 NewDocID :: DI

A.13.5 Archive Message with Respect to Documents

148 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
148 Shred :: {|″stop″|}

A.13.6 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.)

149 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,

[32]For the concept of access rights see Sect. A.13.3.

- 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 !

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

150 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
150 Read :: DI × HI × TIME

A.13.7 Handler Messages with Respect to Management

151 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).

151 Copy :: DI × HI × TIME × DN [cf. Item 141 on Page 43]

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

A.13.8 A Summary of Behaviour Interactions

Figure 4 on the next page summarises the sources, **out**, resp. !, and the targets, **in**, resp. ?, of the messages covered in the previous sections.

A.14 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.

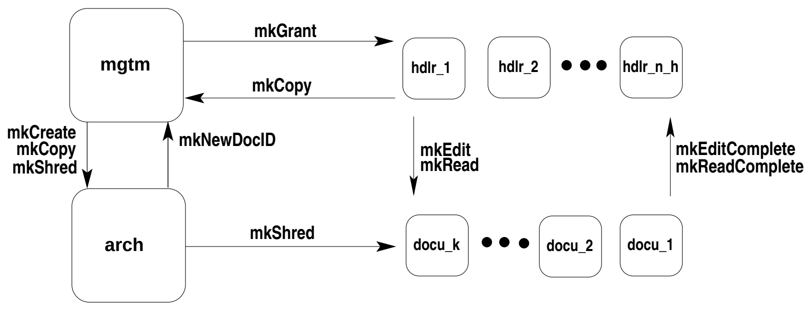


Figure 4: A Summary of Behaviour Interactions

A.15 Channels: A Final View

We can now summarize the types of the various channel messages first referred to in Items 120, 121, 122 and 123.

type
120 MA = Create (Item 140 on Page 43) | Shred (Item 140d on Page 43) | NewDocID (Item 147 on Page 44)
121 MH = Grant (Item 140c on Page 43) | Copy (Item 151 on the preceding page) |
122 AD = Shred (Item 148 on Page 44)
123 HD = Edit (Item 149 on Page 44) | Read (Item 150 on the previous page) | Copy (Item 151 on the preceding page)

A.16 An Informal Summary of Behaviours

A.16.1 The Create Behaviour: Left Fig. 5 on the next page

152 [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. A.13.1 on Page 43, Items 140–140e.)

153 [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.

154 [3] The archive behaviour then "spawns off" document behaviour docu_k – here shown by the "dash–dotted" rounded edge square.

155 [4] The archive behaviour then offers the document identifier k message to the management behaviour.

(We refer to Sect. A.13.4 on Page 44, Item 147.)

156 [5] The management behaviour then

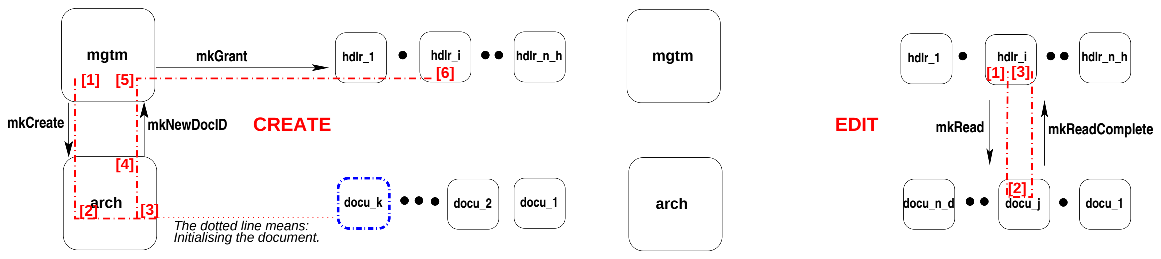


Figure 5: Informal Snapshots of Create and Edit Document Behaviours

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. A.13.2 on Page 43, Item 144 on Page 44.)

157 [6] Handler behaviour i records that it now has certain access rights to doccument i.

A.16.2 The Edit Behaviour: Right Fig. 5

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.

A.16.3 The Read Behaviour: Left Fig. 6 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.

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.

A.16.4 The Copy Behaviour: Right Fig. 6 on the following page

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.

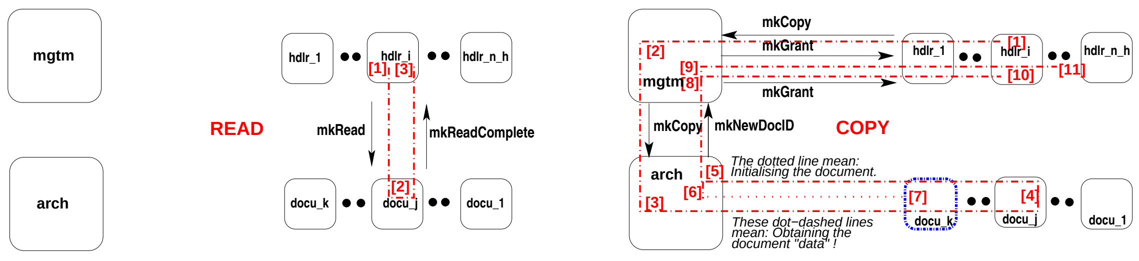


Figure 6: Informal Snapshots of Read and Copy Document Behaviours

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,

10 while also informing the master copy of the copy identity (etcetera).

11 The handlers granted access to the copy record this fact.

A.16.5 The Grant Behaviour: Left Fig. 7 on the next page

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.

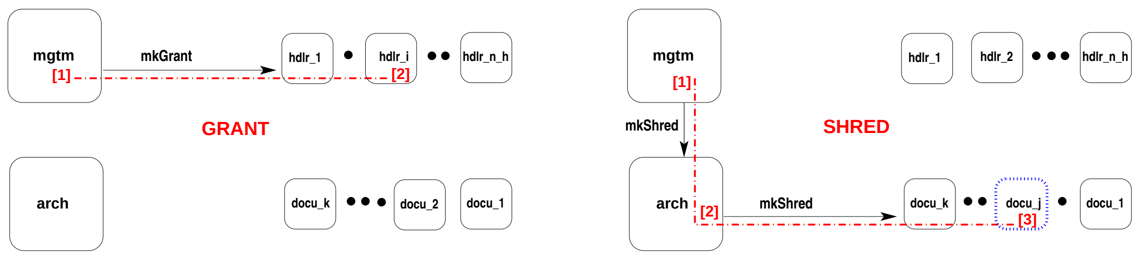


Figure 7: Informal Snapshots of Grant and Shred Document Behaviours

A.16.6 The Shred Behaviour: Right Fig. 7

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).

A.17 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. A.16.1–A.16.6. 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 $\beta_{\tau_1}, \beta_{\tau_2}, ..., \beta_{\tau_\tau}$.

A.17.1 Management Behaviour

158 The management behaviour is involved in the following action traces:

a	**create**	Fig. 5 on Page 47 Left
b	**copy**	Fig. 6 on the facing page Right
c	**grant**	Fig. 7 Left
d	**shred**	Fig. 7 Right

```
value
158   mgtm: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
158   mgtm(mdir) ≡
158a        mgtm_create(mdir)
158b     ⌈⌉ mgtm_copy(mdir)
158c     ⌈⌉ mgtm_grant(mdir)
158d     ⌈⌉ mgtm_shred(mdir)
```

Management Create Behaviour: Left Fig. 5 on Page 47

159 The **management create** behaviour

160 initiates a create document behaviour (i.e., a request to the archive behaviour),

161 and then awaits its response.

```
value
159  mgtm_create: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
159  mgtm_create(mdir) ≡
160  [1]   let hi = mgtm_create_initiation(mdir) ;         [Left Fig. 5 on Page 47]
161  [5]   mgtm_create_awaits_response(mdir)(hi) end  [Left Fig. 5 on Page 47]
```

The **management create initiation** behaviour

162 selects a handler on behalf of which it requests the document creation,

163 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",

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

165 yields the identifier of the chosen handler.

```
value
160  mgtm_create_initiation: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} HI
160  mgtm_create_initiation(mdir) ≡
162            let hi:HI • hi ∈ dom mdir,
163  [1.2−.4]      (dis,info):(DI-set×Info),dn:DN • is_meaningful(embed_DIs_in_DD(dis,info))(mdir) in
164  [1.1]        mgtm_arch_ch ! mkCreate(embed_DIs_in_DD(ds,info),dn,″empty_DC″)
165             hi end

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

The **management create awaits response** behaviour

166 starts by awaiting a reply from the archive behaviour with the identity, *di*, of the document (that that behaviour has created).

167 It then selects suitable access rights,

168 with which it updates its handler/document directory

169 and offers to the chosen handler

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

```
value
161 mgtm_create_awaits_response: MDIR → HI → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
161 mgtm_create_awaits_response(mdir) ≡
166 [5]      let mkNewDocID(di) = mgtm_arch_ch ? in
167 [5.1]    let acrs:ANm-set in
168 [5.2]    let mdir′ = mdir † [hi ↦ [di ↦ acrs]] in
169 [5.3]    mgtm_hdlr_ch[hi] ! mkGrant(di,acrs)
170          mgtm(mdir′) end end end
```

Management Copy Behaviour: Right Fig. 6 on Page 48

171 The **management copy** behaviour

172 accepts a copy document request from a handler behaviour (i.e., a request to the archive behaviour),

173 and then awaits a response from the archive behaviour;

174 after which it grants access rights to handlers to the document copy.

```
value
171 mgtm_copy: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
171 mgtm_copy(mdir) ≡
172 [2] let hi = mgtm_accept_copy_request(mdir) in
173 [8] let di = mgtm_awaits_copy_response(mdir)(hi) in
174 [9] mgtm_grant_access_rights(mdir)(di) end end
```

175 The **management accept copy** behaviour non-deterministically externally (⌷) awaits a copy request from a[ny] handler (i) behaviour –

176 with the request identifying the master document, j, to be copied.

177 The management accept copy behaviour forwards (!) this request to the archive behaviour –

178 while yielding the identity of the requesting handler.

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

The **management awaits copy response** behaviour

179 awaits a reply from the archive behaviour as to the identity of the newly created copy (di) of master document j.

180 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) –

181 while yielding the identity, di, of the newly created copy.

```
158b.    mgtm_awaits_copy_response: MDIR → HI → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} DI
158b.    mgtm_awaits_copy_response(mdir)(hi) ≡
179.   [8]  let mkNewDocID(di) = mgtm_arch_ch ? in
180.        mgtm_hdlr_ch[hi] ! mkCopy(di) ;
181.        di end
```

The **management grants access rights** behaviour

182 selects suitable access rights for a suitable number of selected handlers.

183 It then offers these to the selected handlers.

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

Management Grant Behaviour: Left Fig. 7 on Page 49 The **management grant** behaviour

184 is a variant of the mgtm_grant_access_rights function, Items 182–183.

185 The management behaviour selects a suitable subset of known handler identifiers, and

186 for these a suitable subset of document identifiers from which

187 it then constructs a map from handler identifiers to subsets of access rights.

188 With this the management behaviour then issues appropriate grants to the chosen handlers.

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

Management Shred Behaviour: Right Fig. 7 on Page 49 The **management shred** behaviour

189 initiates a request to the archive behaviour.

190 First the management shred behaviour selects a document identifier (from its directory).

191 Then it communicates a shred document message to the archive behaviour;

192 then it notes the (to be shredded) document in its directory

193 whereupon the management shred behaviour resumes being the management behaviour.

```
value
189   mgtm_shred: MDIR → out mgtm_arch_ch  Unit
189   mgtm_shred(mdir) ≡
190         let di:DI • is_suitable(di)(mdir) in
191   [1]  mgtm_arch_ch ! mkShred(time_ch?,di) ;
192         let mdir′ = [hi↦mdir(hi)\{di}|hi:HI•hi ∈ dom mdir] in
193         mgtm(mdir′) end end
```

A.17.2 Archive Behaviour

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

a **create**	Fig. 5 on Page 47 Left
b **copy**	Fig. 6 on Page 48 Right
c **shred**	Fig. 7 on Page 49 Right

```
type
137   ADIR = avail:DI-set × used:DI-set × gone:DI-set
axiom
137   ∀ (avail,used,gone):ADIR • avail ∩ used = {} ∧ gone ⊆ used
value
194   arch: ADIR → in,out mgmt_arch_ch, {arch_docu_ch[di]|di:DI•di ∈ dis} Unit
194a   arch(adir) ≡
194a          arch_create(adir)
194b        ⌈⌉ arch_copy(adir)
194c        ⌈⌉ arch_shred(adir)
```

The Archive Create Behaviour: Left Fig. 5 on Page 47 The **archive create** behaviour

195 accepts a request, from the management behaviour to create a document;

196 it then selects an available document identifier;

197 communicates this new document identifier to the management behaviour;

198 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

199 in parallel with resuming the archive behaviour with updated programmable attributes.

```
194a.   arch_create: AATTR → in,out mgmt_arch_ch, {arch_docu_ch[di]|di:DI•di ∈ dis} Unit
194a.   arch_create(avail,used,gone) ≡
195.    [2]  let mkCreate((hi,t),dd,an,dc) = mgmt_arch_ch ? in
196.          let di:DI•di ∈ avail in
197.   [4]  mgmt_arch_ch ! mkNewDocID(di) ;
198.   [3]  docu_di(dd)(⟨an⟩,dc,<(date_of_creation)>)
199.          ∥ arch(avail\{di},used∪{di},gone)
194a.          end end
```

The Archive Copy Behaviour: Right Fig. 6 on Page 48 The **archive copy** behaviour

200 accepts a copy document request from the management behaviour with the identity, j, of the master document;

201 it communicates (the request to obtain all the attribute values of the master document, j) to that document behaviour;

202 whereupon it awaits their communication (i.e., (dd,da,dc,dh));

203 (meanwhile) it obtains an available document identifier,

204 which it communicates to the management behaviour,

205 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),

206 in parallel with resuming the archive behaviour with updated programmable attributes.

```
194b.  arch_copy: AATTR → in,out mgmt_arch_ch, {arch_docu_ch[di]|di:DI•di ∈ dis} Unit
194b.  arch_copy(avail,used,gone) ≡
200.    [3] let mkDocID(j,hi) = mgtm_arch_ch ? in
201.        arch_docu_ch[j] ! mkReqAttrs() ;
202.        let mkAttrs(dd,da,dc,dh) = arch_docu_ch[j] ? in
203.        let di:DI • di ∈ avail in
204.        mgtm_arch_ch ! mkCopyDocID(di) ;
205.   [6,7] docu_di(augment(dd,″copy″,j,hi),augment(da,″copy″,hi),dc,augment(dh,(″copy″,date_and_time,j,hi)))
206.      ∥ arch(avail\{di},used∪{di},gone)
194b.      end end end
```

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

The Archive Shred Behaviour: Right Fig. 7 on Page 49 The **archive shred** behaviour

207 accepts a shred request from the management behaviour.

208 It communicates this request to the identified document behaviour.

209 And then resumes being the archive behaviour, noting however, that the shredded document has been shredded.

```
194c.  arch_shred: AATTR → in,out mgmt_arch_ch, {arch_docu_ch[di]|di:DI•di ∈ dis} Unit
194c.  arch_shred(avail,used,gone) ≡
207.   [2] let mkShred(j) = mgmt_arch_ch ? in
208.       arch_docu_ch[j] ! mkShred() ;
209.       arch(avail,used,gone∪{j})
194c.       end
```

A.17.3 Handler Behaviours

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

a	**create**	Fig. 5 on Page 47 Left
b	**edit**	Fig. 5 on Page 47 Right
c	**read**	Fig. 6 on Page 48 Left
d	**copy**	Fig. 6 on Page 48 Right
e	**grant**	Fig. 7 on Page 49 Left

```
value
210   hdlr_hi: HATTRS → in,out mgtm_hdlr_ch[hi],{hdlr_docu_ch[hi,di]|di:DI•di∈dis} Unit
210   hdlr_hi(hattrs) ≡
210a          hdlr_create_hi(hattrs)
210b        ⌈⌉ hdlr_edit_hi(hattrs)
210c        ⌈⌉ hdlr_read_hi(hattrs)
210d        ⌈⌉ hdlr_copy_hi(hattrs)
210e        ⌈⌉ hdlr_grant_hi(hattrs)
```

The Handler Create Behaviour: Left Fig. 5 on Page 47

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

212 It according updates its programmable hattrs attribute;

213 and resumes being a handler behaviour with that update.

```
210a  hdlr_create_hi: HATTRS × HHIST → in,out mgtm_hdlr_ch[hi] Unit
210a  hdlr_create_hi(hattrs,hhist) ≡
211     let mkGrant(di,acrs) = mgtm_hdlr_ch[hi] ? in
212     let hattrs′ = hattrs † [hi ↦ acrs] in
213     hdlr_create_hi(hattrs′,augment(hhist,mkGrant(di,acrs))) end end
```

The Handler Edit Behaviour: Right Fig. 5 on Page 47

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

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

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

217 Editing take some time, *ti*.

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

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

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

```
210b  hdlr_edit_hi: HATTRS × HHIST → in,out {hdlr_docu_ch[hi,di]|di:DI•di∈dis} Unit
210b  hdlr_edit_hi(hattrs,hhist) ≡
214 [1]  let di:DI • di ∈ dom hattrs ∧ ″edit″ ∈ hattrs(di) in
215 [1]  let (e,u):(EDIT×UNDO) • ... , n:AN • ... in
216 [1]  hdlr_docu_ch[hi,di] ! mkEdit(hi,t=time_ch?,e,u,n) ;
217 [2]  let ti:TIME_INTERVAL • ... in
218 [2]  wait ti ; assert: time_ch? = t+ti
219 [3]  let mkEditComplete(ti′,...) = hdlr_docu_ch[hi,di] ? in assert ti′ ≅ ti
220      hdlr_hi(hattrs,augment(hhist,(di,mkEdit(hi,t,ti,e,u))))
210b     end end end end
```

The Handler Read Behaviour: Left Fig. 6 on Page 48

221 The **handler behaviour**, on its own volition, decides to read a document, *di*, for which it has reading rights.

222 It then communicates the desire to read document *di* with at time t=time_ch? – with an annotation note (n).

223 Reading take some time, *ti*.

224 We can therefore assert that the time at which reading has completed is $t+ti$.

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

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

```
210c  hdlr_edit_hi: HATTRS × HHIST → in,out {hdlr_docu_ch[hi,di]|di:DI•di∈dis} Unit
210c  hdlr_edit_hi(hattrs,hhist) ≡
221 [1]  let di:DI • di ∈ dom hattrs ∧ ″read″ ∈ hattrs(di), n:N • ... in
222 [1]  hdlr_docu_ch[hi,di] ! mkRead(hi,t=time_ch?,n) ;
223 [2]  let ti:TIME_INTERVAL • ... in
224 [2]  wait ti ; assert: time_ch? = t+ti
225 [3]  let mkReadComplete(ti,...) = hdlr_docu_ch[hi,di] ? in
226      hdlr_hi(hattrs,augment(hhist,(di,mkRead(di,t,ti))))
210c     end end end
```

The Handler Copy Behaviour: Right Fig. 6 on Page 48

227 The **handler [copy] behaviour**, on its own volition, decides to copy a document, *di*, for which it has copying rights.

228 It communicates this copy request to the management behaviour.

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

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

```
210d    hdlr_copy$_{hi}$: HATTRS × HHIST → in,out mgtm_hdlr_ch[hi] Unit
210d    hdlr_copy$_{hi}$(hattrs,hhist) ≡
227    [1] let di:DI • di ∈ dom hattrs ∧ ″copy″ ∈ hattrs(di) in
228    [1] mgtm_hdlr_ch[hi] ! mkCopy(di,hi,t=time_ch?) ;
229   [10] let mkCopyComplete(di′,di) = mgtm_hdlr_ch[hi] ? in
230   [10] hdlr$_{hi}$(hattrs,augment(hhist,time_ch?,(mkCopy(di,hi,,t),mkCopyComplete(di′))))
210d         end end
```

The Handler Grant Behaviour: Left Fig. 7 on Page 49

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

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

```
210e    hdlr_grant$_{hi}$: HATTRS × HHIST → in,out mgtm_hdlr_ch[hi] Unit
210e    hdlr_grant$_{hi}$(hattrs,hhist) ≡
231    [2] let mkGrant(di,acrs) = mgtm_hdlr_ch[hi] ? in
232    [2] hdlr$_{hi}$(hattrs†[di↦acrs],augment(hhist,time_ch?,mkGrant(di,acrs)))
210e        end
```

A.17.4 Document Behaviours

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

a **edit**	Fig. 5 on Page 47 Right
b **read**	Fig. 6 on Page 48 Left
c **shred**	Fig. 7 on Page 49 Right

```
value
233   docu$_{di}$: DD × (DA × DC × DH) →  in,out arch_docu_ch[di], {hdlr_docu_ch[hi,di]|hi:HI•hi∈his} Unit
233   docu$_{di}$(dattrs) ≡
233a          docu_edit$_{di}$(dd)(da,dc,dh)
233b       ⌈⌉ docu_read$_{di}$(dd)(da,dc,dh)
233c       ⌈⌉ docu_shred$_{di}$(dd)(da,dc,dh)
```

The Document Edit Behaviour: Right Fig. 5 on Page 47

234 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.

235 The edit (etc.) function(s) take some time, ti, to do.

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

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

```
value
233a   docu_edit_di: DD × (DA × DC × DH) → in,out {hdlr_docu_ch[hi,di]|hi:HI•hi∈his} Unit
233a   docu_edit_di(dd)(da,dc,dh) ≡
234    [2] let mkEdit(hi,t,e,u,n) = ⌈⌉{hdlr_docu_ch[hi,di]?|hi:HI•hi∈his} in
234a   [2] let dc′ = e(dc),
234b           da′ = augment(da,((hi,t),(″edit″,e,u),n)),
234c           dh′ = augment(dh,((hi,t),(″edit″,e,u))) in
235        let ti = time_ch? − t in
236        hdlr_docu_ch[hi,di] ! mkEditComplete(ti,...) ;
237        docu_di(dd)(da′,dc′,dh′)
233a       end end end
```

The Document Read Behaviour: Left Fig. 6 on Page 48

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

239 The reading takes some time to do.

240 The handler behaviour is advised on completion.

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

```
value
233b  docu_read_di: DD × (DA × DC × DH) → in,out {hdlr_docu_ch[hi,di]|hi:HI•hi∈his} Unit
233b  docu_read_di(dd)(da,dc,dh) ≡
238   [2] let mkRead(hi,t,n) = {hdlr_docu_ch[hi,di]?|hi:HI•hi∈his} in
239   [2] let ti:TIME_INTERVAL • ... in
239   [2] wait ti ;
240   [2] hdlr_docu_ch[hi,di] ! mkReadComplete(ti,...) ;
241   [2] docu_di(dd)(augment(da,n),dc,augment(dh,(hi,t,ti,″read″)))
233b      end end
```

The Document Shred Behaviour: Right Fig. 7 on Page 49

242 The **document [shred] behaviour** offers to accept a document shred request from the archive behaviour –

243 whereupon it **stops**!

```
value
233c  docu_shred_di: DD × (DA × DC × DH) → in,out arch_docu_ch[di] Unit
233c  docu_shred_di(dd)(da,dc,dh) ≡
242   [3] let mkShred(...) = arch_docu_ch[di] ? in
243       stop
233c  [3] end
```

A.18 Conclusion

This completes a first draft version of this document. The date time is: January 3, 2018: 09:32 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 158 on Page 49, 194 on Page 53, 210 on Page 55 and 233 on Page 57, 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.

B RSL: The RAISE Specification Language – A Primer

B.1 Type Expressions

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

B.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"

B.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$^\omega$
[12] A $\underset{m}{\rightarrow}$ 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., $(A \underset{m}{\overrightarrow{\;}} B)$, or (A*)-**set**, or (A-**set**)list, or $(A|B) \underset{m}{\overrightarrow{\;}} (C|D|(E \underset{m}{\overrightarrow{\;}} F))$, 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

B.2 Type Definitions

B.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
```

B.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) |}
```

B.2.3 Sorts — Abstract Types

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

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

B.3 The RSL Predicate Calculus

B.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 $\sim$a, a$\wedge$b, a$\vee$b, a$\Rightarrow$b, a=b, a$\neq$b

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

B.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
$\sim$a, a$\wedge$b, a$\vee$b, a$\Rightarrow$b, a=b, a$\neq$b
x=y, x$\neq$y,
i<j, i$\leq$j, i$\geq$j, i$\neq$j, i$\geq$j, i>j

are simple predicate expressions.

B.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:

$\forall$ x:X • $\mathcal{P}(x)$
$\exists$ y:Y • $\mathcal{Q}(y)$
$\exists$! 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.

B.5 Concrete RSL Types: Values and Operations

B.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)
```

B.5.2 Set Expressions

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

$$\{\{\},\ \{a\},\ \{e_1,e_2,...,e_n\},\ ...\} \in A\textbf{-set}$$
$$\{\{\},\ \{a\},\ \{e_1,e_2,...,e_n\},\ ...,\ \{e_1,e_2,...\}\} \in A\textbf{-infset}$$

Set Comprehension The expression, last line below, to the right of the $\equiv$, 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)}
```

B.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)
```

B.5.4 List Expressions

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

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

$$\langle\ a\_i\ ..\ a\_j\ \rangle$$

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 ~→ B
value
   comprehend: Aω × P × Q ~→ Bω
   comprehend(l,P,Q) ≡ ⟨ Q(l(i)) | i in ⟨1..len l⟩ • P(l(i))⟩
```

B.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 →m 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 →m V
   F = U ~→ X
   G = V ~→ Y
   P = U → Bool
value
   comprehend: M×F×G×P → (X →m Y)
   comprehend(m,F,G,P) ≡ [ F(u) ↦ G(m(u)) | u:U • u ∈ dom m ∧ P(u)]
```

B.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 ~→ Nat
```

Set Examples

```
examples
   a ∈ {a,b,c}
   a ∉ {}, a ∉ {b,c}
   {a,b,c} ∪ {a,b,d,e} = {a,b,c,d,e}
   ∪{{a},{a,bb},{a,d}} = {a,b,d}
   {a,b,c} ∩ {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 $s \equiv$

```
        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 ∗/
```

B.5.7 Cartesian Operations

```
type
    A, B, C
    g0: G0 = A × B × C
    g1: G1 = ( A × B × C )
    g2: G2 = ( A × B ) × C
    g3: G3 = A × ( B × C )

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

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

B.5.8 List Operations

List Operator Signatures

```
value
    hd: A^ω ~→ A
    tl: A^ω ~→ A^ω
    len: A^ω ~→ Nat
    inds: A^ω → Nat-infset
    elems: A^ω → A-infset
    .(.): A^ω × Nat ~→ A
    ⌢: A* ≠A: A^ωA^ω A* A^ω → BoBbol
```

List Operation Examples

```
examples
    hd⟨a1,a2,...,am⟩=a1
    tl⟨a1,a2,...,am⟩=⟨a2,...,am⟩
    len⟨a1,a2,...,am⟩=m
    inds⟨a1,a2,...,am⟩={1,2,...,m}
    elems⟨a1,a2,...,am⟩={a1,a2,...,am}
    ⟨a1,a2,...,am⟩(i)=ai
    ⟨a,b,c⟩⌢⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
    ⟨a,b,c⟩=⟨a,b,c⟩
    ⟨a,b,c⟩ ≠ ⟨a,b,d⟩
```

Informal Explication

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.

- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i, gives the ith element of the list.
- $\widehat{\ }$: 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″)
```

B.5.9 Map Operations

Map Operator Signatures and Map Operation Examples

value

$\mathsf{m(a)}$: $\mathsf{M} \rightarrow \mathsf{A} \overset{\sim}{\rightarrow} \mathsf{B}$, $\mathsf{m(a)} = \mathsf{b}$

dom: $\mathsf{M} \rightarrow \mathsf{A}$-**infset** [domain of map]
  dom $[\mathsf{a1}\mapsto\mathsf{b1},\mathsf{a2}\mapsto\mathsf{b2},...,\mathsf{an}\mapsto\mathsf{bn}] = \{\mathsf{a1},\mathsf{a2},...,\mathsf{an}\}$

rng: $\mathsf{M} \rightarrow \mathsf{B}$-**infset** [range of map]
  rng $[\mathsf{a1}\mapsto\mathsf{b1},\mathsf{a2}\mapsto\mathsf{b2},...,\mathsf{an}\mapsto\mathsf{bn}] = \{\mathsf{b1},\mathsf{b2},...,\mathsf{bn}\}$

$\dagger$: $\mathsf{M} \times \mathsf{M} \rightarrow \mathsf{M}$ [override extension]
  $[\mathsf{a}\mapsto\mathsf{b},\mathsf{a}'\mapsto\mathsf{bb}',\mathsf{a}''\mapsto\mathsf{bb}''] \dagger [\mathsf{a}'\mapsto\mathsf{bb}'',\mathsf{a}''\mapsto\mathsf{bb}'] = [\mathsf{a}\mapsto\mathsf{b},\mathsf{a}'\mapsto\mathsf{bb}'',\mathsf{a}''\mapsto\mathsf{bb}']$

$\cup$: $\mathsf{M} \times \mathsf{M} \rightarrow \mathsf{M}$ [merge $\cup$]
  $[\mathsf{a}\mapsto\mathsf{b},\mathsf{a}'\mapsto\mathsf{bb}',\mathsf{a}''\mapsto\mathsf{bb}''] \cup [\mathsf{a}'''\mapsto\mathsf{bb}'''] = [\mathsf{a}\mapsto\mathsf{b},\mathsf{a}'\mapsto\mathsf{bb}',\mathsf{a}''\mapsto\mathsf{bb}'',\mathsf{a}'''\mapsto\mathsf{bb}''']$

$\backslash$: $\mathsf{M} \times \mathsf{A}$-**infset** $\rightarrow \mathsf{M}$ [restriction by]
  $[\mathsf{a}\mapsto\mathsf{b},\mathsf{a}'\mapsto\mathsf{bb}',\mathsf{a}''\mapsto\mathsf{bb}'']\backslash\{\mathsf{a}\} = [\mathsf{a}'\mapsto\mathsf{bb}',\mathsf{a}''\mapsto\mathsf{bb}'']$

$/$: $\mathsf{M} \times \mathsf{A}$-**infset** $\rightarrow \mathsf{M}$ [restriction to]
  $[\mathsf{a}\mapsto\mathsf{b},\mathsf{a}'\mapsto\mathsf{bb}',\mathsf{a}''\mapsto\mathsf{bb}'']/\{\mathsf{a}',\mathsf{a}''\} = [\mathsf{a}'\mapsto\mathsf{bb}',\mathsf{a}''\mapsto\mathsf{bb}'']$

$=,\neq$: $\mathsf{M} \times \mathsf{M} \rightarrow$ **Bool**

$^\circ$: $(\mathsf{A} \underset{m}{\overrightarrow{\rightarrow}} \mathsf{B}) \times (\mathsf{B} \underset{m}{\overrightarrow{\rightarrow}} \mathsf{C}) \rightarrow (\mathsf{A} \underset{m}{\overrightarrow{\rightarrow}} \mathsf{C})$ [composition]
  $[\mathsf{a}\mapsto\mathsf{b},\mathsf{a}'\mapsto\mathsf{bb}'] \circ [\mathsf{bb}\mapsto\mathsf{c},\mathsf{bb}'\mapsto\mathsf{c}',\mathsf{bb}''\mapsto\mathsf{c}''] = [\mathsf{a}\mapsto\mathsf{c},\mathsf{a}'\mapsto\mathsf{c}']$

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.
- $\dagger$: Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some "pairings" of the right operand map.
- $\cup$: Merge. When applied to two operand maps, it gives a merge of these maps.
- $\backslash$: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements that are not in the right operand set.
- $/$: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.
- $=$: The equal operator expresses that the two operand maps are identical.
- $\neq$: The nonequal operator expresses that the two operand maps are *not* identical.

- $\circ$: Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1, to the range elements of the right operand map, m_2, such that if a is in the definition set of m_1 and maps into b, and if b is in the definition set of m_2 and maps into c, then a, in the composition, maps into c.

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

```
value
  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
```

B.6 λ-Calculus + Functions

B.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), ...
```

B.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 $\lambda y \bullet e$ if $x \neq y$ and x is free in e.
- ⟨A⟩: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

B.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)).

B.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)

B.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

B.6.6 Function Definitions

Functions can be defined explicitly:

value
  f: Arguments $\rightarrow$ Result
  f(args) $\equiv$ DValueExpr

  g: Arguments $\overset{\sim}{\rightarrow}$ Result
  g(args) $\equiv$ ValueAndStateChangeClause
  pre P(args)

Or functions can be defined implicitly:

value
  f: Arguments $\rightarrow$ Result
  f(args) **as** result
  post P1(args,result)

  g: Arguments $\overset{\sim}{\rightarrow}$ Result
  g(args) **as** result
  pre P2(args)
  post P3(args,result)

The symbol $\overset{\sim}{\rightarrow}$ 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.

B.7 Other Applicative Expressions

B.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))($\mathcal{E}_d$)

B.7.2 Recursive let Expressions

Recursive **let** expressions are written as:

  let f = λa:A • E(f) **in** B(f,a) **end**

is "the same" as:

  let f = **YF in** B(f,a) **end**

where:

  F $\equiv$ λg•λa•(E(g)) and YF = F(YF)

B.7.3 Predicative let Expressions

Predicative **let** expressions:

let a:A • $\mathcal{P}$(a) **in** $\mathcal{B}$(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).

B.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
```

B.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
```

B.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⟩ ::= !
```

B.8 Imperative Constructs

B.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**.

B.8.2 Variables and Assignment

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

B.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
```

B.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**

B.8.5 Iterative Conditionals

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

B.8.6 Iterative Sequencing

8. **for** e **in** list_expr • P(b) **do** S(b) **end**

B.9 Process Constructs

B.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).

B.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.

B.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.

B.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.

B.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.[33]

[33] For schemes, classes and objects we refer to [2, Chap. 10]