Double-entry Bookkeeping

Dines Bjørner[*]
Technical University of Denmark
Fredsvej 11, DK-2840 Holte, Danmark
E–Mail: bjorner@gmail.com, **URL:** www.imm.dtu.dk/~db

April 30, 2024, 16:06

77

Abstract

We step-wise unfold a formal model of a double-entry bookkeeping "system". We develop the model is stages: from very simplistic, to reasonably "full-blown" realistic. First we develop a model in the traditional abstract software specification style. Then we embed a final stage of the traditional model in a "prototype" domain model.

Contents

[*]This report is under copyright protection: © Dines Bjørner, April 5, 2024

1 Introduction

1.1 What is This All About ?

We shall present a description of certain aspects of double-entry bookkeeping.[1] The description, in Sects. 3 and 3, focus on the "classical" issue of single- and double-entry bookkeeping. Whereas the description, in Sect. 5, focus on the domain modelling, that is, of embedding bookkeeping in models of such domains as road-pricing, shipping, retailing, manufacturing, etc. We refer to the books [6, 11] for introductions to domain modelling, and to the `Internet` document [9, *Domain Models – A Compendium*] for a compendium on some 15 [such] domains.[2] Double-entry bookkeeping, per se, is not [really] a domain issue. But its relation to domains is obvious !

1.2 A Background – A Context

There are two issues at play here.

[1] https://en.wikipedia.org/wiki/Double-entry_bookkeeping

[2] It is the intention, eventually, to include this document's model of double-entry bookkeeping into that compendium.

1.2.1 Background

The working-out of this model of *double-entry bookkeeping* takes place on/in the following background.

1.2.1.1 Actual Double-entry Bookkeeping Systems.

Having first learned basic skills of *double-entry bookkeeping* and passed an examn during my MSc studies, 1956–1962. Having realized that *double-entry bookkeeping* represents an example of *intentional pull*, in recent years, cf. Sect. 5.6 of [11], 2020. Having a neighbour, "up the road", who has made a first fortune on *double-entry bookkeeping* software. But, having "studied" commercial, on the market *double-entry bookkeeping* software packages[3], never been quiet content with their explanation of these software systems.

1.2.1.2 Formal Software Development.

Since 1973, i.e., since my work at the IBM Vienna Laboratory, Austria, it has been clear to me that programs, their specification, and hence also now, domain descriptions and requirements prescriptions are mathematical object. And that the development of software can, and, to me, thus should be orderly developed: in phases from domain descriptions via requirements prescriptions to software and its code. All this is presented in [5, 6, 11].

1.2.2 Context

The Triptych Dogma

In order to *specify* **software**,
we must understand its requirements.
In order to *prescribe* **requirements**,
we must understand the **domain**.
So we must **study, analyze** and **describe** domains.

The specific context in which this report, on what may seem a rather "low-level" topic, is then conceived is this. First: the above, the **The Triptych Dogma**. Then fact that each human artifact domain — such as those described in [9] — somehow or other include a [double-entry] bookkeeping element.

The general context is that of *the specific view of software development* as represented by the *Software Engineering Terminology* of Appendix A.

1.3 Terminologies

In any construction project, in any domain, whether for software or other, it is vitally important to agree on all professional terms. It seems that this is especially important in the software business. The domain description that we shall unfold, later, is one such registration of all the relevant professional terms of the field of double-entry bookkeeping. But before any attempt at modelling the domain, as an element of its study and analysis we urge the domain describer to first establish appropriate terminologies.

There are basically two terminologies. One, in Sect. 2, for the *financial management* terms related to *double-entry bookkeeping.* And another, Appendix 2, related to *domain, requirements* and *software engineering*.

[3]– but, it must be said: never personally used such software

1.4 A Caveat

The present, Spring 2024, report is a torso. It sketches while also presenting the essential facets: the updating of double-entry bookkeeping debit/credit and asset/liability accounts. We leave it to the reader to complete possibly "dangling" descriptions: narratives and formalizations; to tie the various description elements together, and "embed" the result in a specific [road pricing, container shipping, retailer, banking, or pipeline domain.

1.5 Structure of Report

- In Sect. 2 we present, mostly from/courtesy `Wikipadia`, a vocabulary of terms relevant to bookkeeping.
- Section 3 then presents, in the style of [5, *Software Engineering, vols. 1–3* 2005/2006] a series of from very simple to reasonably realistic single-entry bookkeeping models.
- Section 4 then "generalizes" this to a double-entry bookkeeping model.
- Section 5 finally "embeds" the double-entry bookkeeping model into a model f the domain of accountancy.

2 Financial Management Terminology

I expect to insert more term explanations.

- **Account:** In bookkeeping, an account refers to assets, liabilities, income, expenses, and equity, as represented by individual ledger pages, to which changes in value are chronologically recorded with debit and credit entries. These entries, referred to as postings, become part of a book of final entry or ledger. Examples of common financial accounts are sales, accounts receivable, mortgages, loans, PP&E (Property, Plant, and Equipment), common stock, sales, services, wages and payroll.

  A chart of accounts provides a listing of all financial accounts used by particular business, organization, or government agency.

  The system of recording, verifying, and reporting such information is called accounting. Practitioners of accounting are called accountants.

- **Asset:** An asset is any resource owned or controlled by a business or an economic entity. It is anything (tangible or intangible) that can be used to produce positive economic value. Assets represent value of ownership that can be converted into cash (although cash itself is also considered an asset). The balance sheet of a firm records the monetary value of the assets owned by that firm. It covers money and other valuables belonging to an individual or to a business.[

  Assets can be grouped into two major classes: tangible assets and intangible assets. Tangible assets contain various sub-classes, including current assets and fixed assets. Current assets include cash, inventory, accounts receivable, while fixed assets include land, buildings and equipment. Intangible assets are non-physical resources and rights that have a value to the firm because they give the firm an advantage in the marketplace. Intangible assets include goodwill, intellectual property (such as copyrights, trademarks, patents, computer programs), and financial assets, including financial investments, bonds, and companies' shares.

IFRS (International Financial Reporting Standards), the most widely used financial reporting system, defines: “An asset is a present economic resource controlled by the entity as a result of past events. An economic resource is a right that has the potential to produce economic benefits.”

- **Audit:** An audit is an *independent examination of financial information of any entity, whether profit oriented or not, irrespective of its size or legal form when such an examination is conducted with a view to express an opinion thereon.* Auditing also attempts to ensure that the books of accounts are properly maintained by the concern as required by law. Auditors consider the propositions before them, obtain evidence, roll forward prior year working papers, and evaluate the propositions in their auditing report.

  Audits provide third-party assurance to various stakeholders that the subject matter is free from material misstatement The term is most frequently applied to audits of the financial information relating to a legal person. Other commonly audited areas include: secretarial and compliance, internal controls, quality management, project management, water management, and energy conservation. As a result of an audit, stakeholders may evaluate and improve the effectiveness of risk management, control, and governance over the subject matter.

- **Auditor:** An auditor is a person or a firm appointed by a company to execute an audit. To act as an auditor, a person should be certified by the regulatory authority of accounting and auditing or possess certain specified qualifications. Generally, to act as an external auditor of the company, a person should have a certificate of practice from the regulatory authority.

- **Balance:** In banking and accounting, the balance is the amount of money owed (or due) on an account.

  In bookkeeping, “balance” is the difference between the sum of debit entries and the sum of credit entries entered into an account during a financial period. When total debits exceed the total credits, the account indicates a debit balance. The opposite is true when the total credit exceeds total debits, the account indicates a credit balance. If the debit/credit totals are equal, the balances are considered zeroed out. In an accounting period, “balance” reflects the net value of assets and liabilities to better understand balance in the accounting equation.

- **Credits and Debits:** Credits and debits in double-entry bookkeeping are entries made in account ledgers to record changes in value resulting from business transactions. A debit entry in an account represents a transfer of value to that account, and a credit entry represents a transfer from the account. Each transaction transfers value from credited accounts to debited accounts. For example, a tenant who writes a rent cheque to a landlord would enter a credit for the bank account on which the cheque is drawn, and a debit in a rent expense account. Similarly, the landlord would enter a credit in the rent income account associated with the tenant and a debit for the bank account where the cheque is deposited.

  Debits and credits are traditionally distinguished by writing the transfer amounts in separate columns of an account book. This practice simplified the manual calculation of net balances before the introduction of computers; each column was added separately, and then the smaller total was subtracted from the larger. Alternately, debits and credits can be listed in one column, indicating debits with the suffix ”Dr” or writing them plain, and indicating credits with the suffix ”Cr” or a minus sign. Debits and credits do not, however, correspond in a fixed way to positive and negative numbers. Instead the correspondence depends on the normal balance convention of the particular account.

- **Double-entry accounting:** See Double-entry bookkeeping.
- **Double-entry bookkeeping:** Double-entry bookkeeping, also known as double-entry accounting, is a method of bookkeeping that relies on a two-sided accounting entry to maintain financial information. Every entry to an account requires a corresponding and opposite entry to a different account. The double-entry system has two equal and corresponding sides, known as debit and credit[4]; this is based on the fundamental accounting principle that for every debit, there must be an equal and opposite credit. A transaction in double-entry bookkeeping always affects at least two accounts, always includes at least one debit and one credit, and always has total debits and total credits that are equal.

  A Complete Transaction: In our model "the two sides" are instead modelled as a pair of pairs: A *debit/credit* pair and an *asset/liability* pair. Thus a "completed" transaction[5] in our double-entry bookkeeping should always affects at least two accounts, always includes a *debit/credit* and an *asset/liability*, and always has total *debit/credits* and total *asset/liability* that should be equal.

- **Equity:** Ownership of assets that have liabilities attached to them:
  - **Stock:** equity based on original contributions of cash or other value to a business.
  - **Home equity:** the difference between the market value and unpaid mortgage balance on a home.
  - **Private equity:** stock in a privately held company.
  - **Equity Method:** Equity method in accounting is the process of treating investments in associate companies. Equity accounting is usually applied where an investor entity holds 2050% of the voting stock of the associate company, and therefore has significant influence on the latter's management. Under International Financial Reporting Standards, equity method is also required in accounting for joint ventures.[1] The investor records such investments as an asset on its balance sheet. The investor's proportional share of the associate company's net income increases the investment (and a net loss decreases the investment), and proportional payments of dividends decrease it. In the investors income statement Equity accounting may also be appropriate where the investor has a smaller interest, depending on the nature of the actual relationship between the investor and investee. Control of the investee, usually through ownership of more than 50% of voting stock, results in recognition of a subsidiary, whose financial statements must be consolidated with the parent's. The ownership of less than 20% creates an investment position, carried at historic book or fair market value (if available for sale or held for trading) in the investor's balance sheet.[6]
- **Ledger:**A ledger[1] is a book or collection of accounts in which accounting transactions are recorded. Each account has: (1) an opening or brought-forward balance; (2) a list of transactions, each recorded as either a debit or credit in separate columns (usually with a counter-entry on another page) and (3) an ending or closing, or carry-forward, balance.

[4]This is strange: I must check this. In the model of this paper one of the two accounts is a, or the, debit/credit account, the other the asset/liability account.

[5]By a "complete" transaction we shall understand a set of two or more *writes* (*updates*): a *debit/credit* account update and one or more *asset/liability* account updates – cf. Sect. 5.2.1 on page 29.

[6]https://ifrscommunity.com/knowledge-base/equity-method/

- **Liability:** Liability: a current obligation of an entity arising from past transactions or events.

  In accounting, **contingent liabilities** are liabilities that may be incurred by an entity depending on the outcome of an uncertain future event[1] such as the outcome of a pending lawsuit. These liabilities are not recorded in a company's accounts and shown in the balance sheet when both probable and reasonably estimable as 'contingency' or 'worst case' financial outcome. A footnote to the balance sheet may describe the nature and extent of the contingent liabilities. The likelihood of loss is described as probable, reasonably possible, or remote. The ability to estimate a loss is described as known, reasonably estimable, or not reasonably estimable. It may or may not occur.

  Current liability, or **short-term liabilities** are obligations that will be settled by current assets or by the creation of new current liabilities.

  Non-current, or **Long-term liabilities**, are liabilities with a future benefit over a certain period of time (e.g. longer than one year)

3 A Sequence of Models of Single-entry Bookkeeping

3.1 A Simplest Single Entry Model

The simplest possible accounting just records the *budget* and the *debit/credit balance*. There is no recording of the earnings and expenditure transactions.

3.1.1 A Formal Type Model

1. An simplest account is just a pair of a *budget* and what has been accumulated: *debit [income]* and *credit [expense]*.
2. The budget is a natural number of [currency] units allocated.
3. *Debit [income]* & *Credit [expense]* entry is an integer number of [currency] units that has been *earned* or *spent*.

type
1. ACCOUNT_0 = BUDGET_0 × DEB_CRE_0
2. BUDGET_0 = **Nat**
3. DEB_CRE_0 = **Int**

3.1.2 A Formal 'Semantics' Model.

There is, basically, no bookkeeping to be associated with this model. Expenses result in the debit/credit being lowered. Income result in the debit/credit being lowered. No record is made (i.e., "written down") of these "transactions.

4. There is an *account* value.
5. There are two kinds of transactions: expenses and incomes.
6. It's debit/credit element is being decreased by expenses.

7. And increased by income.

value
4. (budget,deb_cre):ACCOUNT_0
type
5. Transaction = Expense | Income
6. Expense = **Nat**
7. Income = **Nat**
value
6. expense: Expense → ACCOUNT_0 → ACCOUNT_0
6. expense(n)(budget)(deb_cre) ≡ (budget,deb_cre − n)
7. income: Income → ACCOUNT_0 → ACCOUNT_0
7. income(n)(budget)(deb_cre) ≡ (budget,deb_cre + n)

3.2 Two Simple Single Entry Semantic Type Models

3.2.1 Simple Account Lists

3.2.1.1 A Formal Model.

8. A simple *account* is a pair of an *debit [income]* and *credit [expense] accounts*.
9. *Debit [income] accounts* are *account triplets*
10. *Credit [expense] accounts* are *account triplets*
11. *Account triplets* are triplets of a *budget*, an *entry list* and the sum total of what has been *earned* or *spent*.[7]
12. A *budget* is as defined in Item 2 on the preceding page.
13. An *entry list* is a list of entries.
14. An *entry*[8] is a triplet of a time-stamp, some [explanatory] text, and an *amount earned* or *spent*.
15. A *time* stamp is further unspecified.
16. The explanatory *text* is further unspecified.
17. The *amount* is a natural number of [currency] units that has been *earned* or *spent*.

The *simple account lists* model thus has both the income and the expense accounts being lists of time-stamped, text-explained transactions.

type
8. ACCOUNT_1 = DEBIT_ACCOUNT_1 × CREDIT_ACCOUNT_1
9. DEBIT_ACCOUNT_1 = ACCOUNT_TRIPLE_1

[7] The term 'earned' is used in connection with *income accounts*, and the term 'spent' in connection with *expense accounts*.

[8] An *entry* is the recorded evidence of a *transaction*. A *transaction* is an action, i.e., something that changes a state.

10. CREDIT_ACCOUNT_1 = ACCOUNT_TRIPLE_1
11. ACCOUNT_TRIPLE_1 = BUDGET_1 × s_entries:ENTRY_LIST_1 × s_amount:AMOUNT_1
12. BUDGET_1 = BUDGET_0
13. ENTRY_LIST_1 = ENTRY_1*
14. ENTRY_1 = s_time:$\mathbb{TIME}$ × s_text:E_Text_1 × s_amount:AMOUNT_1
15. $\mathbb{TIME}$ = ...
16. E_Text_1 = ...
17. AMOUNT_1 = **Nat**

3.2.1.2 Wellformedness.

18. *Entries* in a *list of entries* are ordered *time*-wise in ascending order – with adjacent entries possibly have same time stamps.
19. The sum total of all *amounts* in an *account entry list* must equal the *spent* entry of the *account*.

axiom [Time-ordering]
18. ∀ el:ENTRIES_1 • ∀ i,j:**Nat** • {i,j}⊆**inds** el ∧ i<j ≡ s_time(el(i))≤s_time(el(j))

19. ∀ (inc_acct_1,eps_acct_1):ACCOUNT_1 •
19. **let** total_inc = s_amount(inc_acct_1), total_exp = s_amount(exp_acct_1) **in**
19. **let** income = sum(s_entries(inc_acct_1)), expenses = sum(s_entries(exp_acct_1)) **in**
19. total_inc= income ∧ total_exp = expenses **end end**

value
19.‘ sum: ENTRIES_1 → Amount_1
19.′ sum(el) ≡ **case** el **of** ⟨⟩ → 0, ⟨(_,_,a)⟩⌢el′ → a + sum_amounts(el′) **end**

3.2.2 Simple Account Maps.

3.2.2.1 A Formal Model.

The *simple account map* model introduces separate account name lists of time-stamped, text-explained transactions.

20. [ι 8 π 9] A simple *account* is a pair of an *debit* and *credit accounts*.
21. [ι 9 π 9] *Debit accounts* are *account triplets*
22. [ι 10 π 9] *credit accounts* are *account triplets*
23. [*] *Account triplets* are triplets of a *budget*, an *entry map* and the sum total of what has been *earned* or *spent.*
24. [ι 12 π 9] A *budget* is as defined in Item 2 on page 8.
25. [*] An *entry map* is a map of *account named entry lists*.
26. [ι 8 π 9] An *entry list* is a list of entries.

27. [ι 13 π 9] An *entry* is a triplet of a time-stamp, some [explanatory] text, and an *amount earned* or *spent*.

28. [ι 14 π 9] A *time* stamp is further unspecified.

29. [ι 15 π 9] The explanatory *text* is further unspecified.

30. [ι 16 π 9] The *amount* is a natural number of [currency] units that has been *earned* or *spent*.

The [ι#π#] refers to ι*tem*/π*age* entries. The [*]-marked items represent the changes wrt. the simple account lists model 3.2.1 on page 9.

type
20. ACCOUNT_2 = DEBIT_ACCOUNT_2 × CREDIT_ACCOUNT_2
21. DEBIT_ACCOUNT_2 = ACCOUNT_TRIPLE_2
22. CREDIT_ACCOUNT_2 = ACCOUNT_TRIPLE_2
23. ACCOUNT_TRIPLE_2 = BUDGET_2 × s_entries:ENTRY_MAP_2 × s_amount:AMOUNT_2
24. BUDGET_2 = BUDGET_0
25. ENTRY_MAP_2 = Acc_Name $\overrightarrow{m}$ ENTRY_LIST_2
26. ENTRY_LIST_2 = ENTRY_2*
27. ENTRY_2 = s_time:$\mathbb{TIME}$ × s_text:E_Text_2 × s_amount:AMOUNT_2
[ι 15 π 9]. $\mathbb{TIME}$ = ...
29. E_Text_2 = ...
30. AMOUNT_2 = **Int**

3.2.2.2 Wellformedness.

31. [ι 18 π 10] *Entries* in a *list of entries* are ordered *time*-wise in ascending order – with adjacent entries possibly have same time stamps.

32. [ι 19 π 10] The sum total of all *amounts* in an *account entry list* must equal the *earned* or *spent* entry of the *account*.

axiom [Time-ordering]
18. ∀ el:ENTRIY_LIST_2 • ∀ i,j:**Nat** • {i,j}⊆**inds** el ∧ i<j ≡ s_time(el(i))≤s_time(el(j))

19. ∀ (inc_acct_2,eps_acct_2):ACCOUNT_2 •
19. **let** total_inc = s_amount(inc_acct_2), total_exp = s_amount(exp_acct_2) **in**
19. **let** income = sum(s_entries(inc_acct_2)), expenses = sum(s_entries(exp_acct_2)) **in**
19. total_inc= income ∧ total_exp = expenses **end end**

value
sum: ENTRIES_2 → Amount_1
sum(el) ≡ **case** el **of** ⟨⟩ → 0, ⟨(_,_,a)⟩^el′ → a + sum_amounts(el′) **end**

3.3 A General Single Entry Model

3.3.1 A Formal Model

3.3.1.1 A Type Model.

33. [ι 8 π 9] An *account* is a pair of an *debit [income]* and *credit [expense] accounts*.
34. [ι 9 π 9] *Debit [Income] accounts* are *account triplets*
35. [ι 10 π 9] *Debit [Expense accounts* are *account triplets*
36. [ι 23 π 10] *Account triplets* are triplets of a *budget*, an *entries component* and a sum total of what has been *earned* or *spent.*
37. [ι 12 π 9] A *budget* is as defined in Item 2 on page 8.
38. [*] An *entries component* is a map from [sub-]*account names* to either an *entry list* or an *entry map.*
39. [ι 8 π 9] An *entry list* is a triple of a *budget*, a list of simple entries, and a sum total of what has been *earned* or *spent.*
40. [*] An *entry map* is a triplet of a *budget*, a map, and a sum total of what has been *earned* or *spent.*
41. The *map* is from [sub]*account names* to *account triplets*
42. [ι 8 π 9] A *simple entry* is a triplet of a time-stamp, some [explanatory] text, and an *amount spent*
43. [ι 15 π 9] A *time* stamp is further unspecified.
44. [ι 16 π 9] The explanatory *text* is further unspecified.
45. [ι 17 π 9] The *amount* is a natural number of [currency] units that has been *earned* or *spent.*

The [*]-marked items represent the changes wrt. the simple account lists model 3.2.2 on page 10.

type
33. ACCOUNT_3 = DEBIT_ACCOUNT_3 × CREDIT_ACCOUNT_3
34. DEBIT_ACCOUNT_3 = ACCOUNT_TRIPLET_3
35. CREDIT_ACCOUNT_3 = ACCOUNT_TRIPLET_3
36. ACCOUNT_TRIPLET_3 = BUDGET_3 × s_entries:ENTRIES_3 × s_total:AMOUNT_3
37. BUDGET_3 = BUDGET_0
38. [*] ENTRIES_3 = Account_Name $\overrightarrow{m}$ s_entries:(ENTRY_LIST_3 | ENTRY_MAP_3
39. ENTRY_LIST_3 = BUDGET_3 × s_entry_list:ENTRY_3* × s_sub_total:AMOUNT_3
40. [*] ENTRY_MAP_3 = s_budget:BUDGET_3 × MAP_3 × s_total:AMOUNT_3
41. [*] MAP_3 = Acc_Name $\overrightarrow{m}$ ACCOUNT_TRIPLET_3
42. ENTRY_3 = s_time:$\mathbb{TIME}$ × s_text:E_Text_3 × s_amount:AMOUNT_3
[ι 15 π 9]. $\mathbb{TIME}$ = ...
44. E_Text_3 = ...
45. AMOUNT_3 = **Nat**

Figure 1 intends to graphically + textually illustrate a specific [ACCOUNT_3] account.

```
      ┌                                                                         ┐
      │ _->(#,<(_,_,*),(_,_,*),...,(_,_,*)>,*),                                 │          [1]
      │ _->(#,<(_,_,*),(_,_,*),...,(_,_,*)>,*),                                 │          [2]
      │            ┌ _->(#,<(_,_,*),...,(_,_,*)>,*),┐                           │          [3]
(#,   │ _->(#,     │ ... ,                          │ ,*),                      │ ,*)      [4]
      │            └ _->(#,<(_,_,*),...,(_,_,*)>,*) ┘                           │          [5]
      │ ... ,                                                                   │
      │ _->(#,<(_,_,*),(_,_,*),...,(_,_,*)>,*)                                  │          [6]
      └                                                                         ┘
```

Figure 1: **An Account.**

0. Slanted, bracketed numerals, e.g., *[1,3,5]*, refer to text lines of the figure.
1. Texts after →s in lines *[1,2,6]* stand 2. for ENTRY_LIST_3s.
3. Text of leftmost →s in line *[4]* stands for an ENTRY_MAP_3.
4. Text lines *[3,5]* within the ENTRY_MAP_3 stand for ENTRY_LIST_3s.
5. The '#'s *[1,2,3,4,4,5,6]* stands for a *budget*s.
6. The '_'s immediately to the right of the opening square brackets *[1,2,3,4,5,6]* stand for *account names*.
7. The '_'s right after the '→ parentheses '_'s *[1,2,3,4,4,6]* stands for *budgets*.
8. All other '_'s stands for *time*, resp. *texts*.
9. The '*'s stands for *amounts*.

Constraints:

10. The first three '*'s in the first two and the last text lines *[1,2,6]* must sum up to the last '*' in those lines.
11. Similarly for the first two '*'s in the 3rd and the 4th *[3,4]* text lines: they must sum up to the last '*' in those lines.
12. The last '*'s in the arrowed (→) lines *[1,2,3,5,6]* must sum up to the last two '*', respectively, in the rightmost text lines *[4]*.
14. Similar constraints apply to *budget* entries:
    15. The sum of the first #s in line *[1,2,6]* and the second # in line *[4]* must equal the first # in line *[1]*.
    16. The sum of the first #s in lines *[3,5]* must equal the second # in line *[4]*.

3.3.1.2 Access Paths.

We define an auxiliary function: access_paths. An *access path* is a sequence of *account names* such that the first element of the path applies to a [root] account and selects either an entry list or an entry map of either an income or an expense account. And the first of a possible tail of the path accesses an entry list or an entry map of the selected former such entry. Et cetera. Thus *debit* and *credit accounts* define each their sets of *access paths*.

46. An *access path* is a sequence of *account names*.
47. access_paths applies to either *debit* and *credit accounts* and yields a set of *access paths*.
48. Since *debit* and *credit accounts* are *account triplets* one can select their *entries* component.

    [38. An *entries component* is a map to either *entry_lists* or *entry_maps*.]

49. If *entry_lists*, then a set of *singleton access paths*, ⟨an⟩, for each of the account names of the *entry_lists* is yielded.

50. If *entry_maps*, then a set, map_access_paths(map), of all the *access paths* reachable from, and including the *map access path*, ⟨an⟩ is yielded.

type
46. Acces_Path = Account_Name*
value
47. access_paths: (DEBIT_ACCOUNT_3|CREDIT_ACCOUNT_3) → Access_Path-**set**
47. access_paths(acc_trip) ≡
48. **let** entries = s_entries(acc_trip) **in**
49. is_ENTRY_LIST_3(entries(an))
49. → { ⟨an⟩ | an:Acc_Name • an ∈ **dom** entries }
50. is_ENTRY_MAP_3(entries(an))
50. → ∪ {map_access_paths(entries(an)) | an:Acc_Name • an ∈ **dom** entries }
47. **end**
47. access_paths: ACCOUNT_3 → Access_Path-**set**
47. access_paths(debit_account,_) ≡ access_paths(debit_account)

51. The map_access_paths function applies to map:ENTRY_MAP_3s and yields a set of *access paths*.

    [36 [ι 23 π 10]. Each map range element is an *account triplet* and these are triplets of a *budget*, an *entries component* and the sum total of what has been *earned* or *spent*.]

52. So for each *account name*, an, in the map that *account name* is prefixed each of the the *access paths* from that *account triple*.

value
51. map_access_paths: ENTRY_MAP_3 → Access_Path-**set**
51. map_access_paths(entry_map) ≡
52. { ⟨an⟩⌢ap | an:Account_Name • an ∈ **dom** entry_map,
52. ap:Access_Path • ap ∈ access_paths(entry_map(an)) }

3.3.1.3 Well-formed Access Paths.

We aim at expressing that all *account names* are distinct. To to so we build up that well-formedness criterion in two stages.

53. The *account names* of any access path are distinct.

axiom
53. ∀ ap:Access_Path • **card elems** ap = le ap

54. Any two distinct *access paths*, if they share an *account name* then it is the first element of these *access paths*.

axiom [Access Paths]
54. $\forall$ ap,ap$'$:Access_Path •
54. **elems** ap $\cap$ **elems** **elems** ap$'$ $\neq$ {}
54. $\Rightarrow$ **hd** ap=**hd** ap$'$ $\wedge$ {**hd** ap}=**elems** ap $\cap$ **elems** **elems** ap$'$

We can choose to let the *debit* and the *credit accounts* be "identically structured", that is have exactly the same access paths:

55. The *debit* and the *credit accounts* have exactly the same *access paths:*

axiom [Identical Access Paths]
55. $\forall$ (inc_acc,exp_acc):ACCOUNT_3 • access_paths(inc_acc) = access_paths(exp_acc)

Or we could choose otherwise, cf. Sect. 3.3.1.5. That should suffice. [Prove that !]

3.3.1.4 Account Access.

An *access path* "points" to an *entry list*. A proper prefix, i.e., if the *access path* is of **len**gth 2 or more, "points" to an *entry map*.

56. The function access takes as `argument` an *access path* or a proper prefix thereof and `applies` to either an *debit* or an *credit* account and `yields` either an *entry list* or an *entry map*.
57. If the *access path*
58. is of length 1, i.e., $\langle$an$\rangle$, then *select* the *entries* of the *account* as the result.
59. If the *access path* is of length more than 1, i.e., $\langle$an$\rangle$$^\frown$ap$'$, then access the account obtained from *access path* $\langle$an$\rangle$ with *access path* ap$'$.

value
56. access: Access_Path×(DEBIT_ACCOUNT_3|CREDIT_ACCOUNT_3)
56. $\rightarrow$ (ENTRY_LIST_3|ENTRY_MAP_3)
56. access(ap,account) $\equiv$
57. **case** ap **of**
58. $\langle$an$\rangle$ $\rightarrow$ s_entries(account),
59. $\langle$an$\rangle$$^\frown$ap$'$ $\rightarrow$ access(ap$'$,s_entries(account))
56. **end**
56. **pre** ap $\in$ access_paths(account) $\vee$ $\exists$ ap$'$• $\in$ access_paths(account) $\wedge$ ap $\in$ prefix_paths(ap$'$)

prefix_paths: Access_Path $\rightarrow$ Access_Path-**set**
prefix_paths(ap) $\equiv$ { $\langle$an(i)| i:**Nat** • 1$\leq$i$\leq$**len** ap $\rangle$ }

3.3.1.5 Summary Expense Accounts.

There are at least two other possibilities of distinguishing between income and expenses.

3.3.1.5.1 Paired Debit/Credit Entries

- The ACCOUNT_3 model, cf. Item 33 on page 12,
  - has the ENTRY_LIST_3s cf. Item 39 on page 12,
  - be simple triplets:
  - ENTRY_LIST_3 = BUDGET_3 × s_entry_list:ENTRY_3* × s_total:AMOUNT_3.
- Instead we could avoid the distinction
  - at the top level of the ACCOUNT_3 model
  - between INCOME_ACCOUNT_3s and EXPENSE_ACCOUNT_3s.
    - Instead ACCOUNT_3s are now just ACCOUNT_TRIPLEs.
    - But ENTRY_LIST_3s now make the distinction between *debit* and *credit*:
    - BUDGET_3×s_entry_lists(s_inc:ENTRY_3*,s_exp:ENTRY_3*)×s_total:AMOUNT_3.

type
33. ACCOUNT_4 = ACCOUNT_TRIPLET_4
36. ACCOUNT_TRIPLET_4 = BUDGET_4×s_entries:ENTRIES_4×s_total:AMOUNT_4
37. BUDGET_4 = BUDGET_0
38. ENTRIES_4 = Account_Name $\overrightarrow{m}$ (ENTRY_LIST_4|ENTRY_MAP_4)
39. [*] ENTRY_LIST_4 = BUDGET_4×s_debit:ENTRY_4*,s_credit:ENTRY_4*×s_total:AMOUNT_4
40. [*] ENTRY_MAP_4 = s_budget:BUDGET_4 × MAP_4 × s_total:AMOUNT_4
41. [*] MAP_4 = Acc_Name $\overrightarrow{m}$ ACCOUNT_TRIPLET_4
42. ENTRY_4 = s_time:$\mathbb{TIME}$×s_text:E_Text_4×s_amount:AMOUNT_4
[ι 15 π 9]. $\mathbb{TIME}$ = ...
44. E_Text_4 = ...
45. AMOUNT_4 = **Nat**

3.3.1.5.2 Summary Credit Entries

Instead of pairing, as in Sect. 3.3.1.5.1, *debit* and *credit* entries, one could summarize *expenses* in “earlier” entries, that is, in entries with whose *access path* is a prefix of the the *access path*, ap, to the *debit* entry, however with an account name ⟨an⟩, suffixed to ap,

We leave the formalization to the reader !

4 A Double-entry Bookkeeping Model

We present the *double-entry bookkeeping* as a pair of pairs ! That is: a pair of *debit/credit accounts* and a pair of *asset/liability accounts.*

4.1 A Type Model

Each of the pairs are type-structured as were the accounts in Sect. 3.3.1.1 on page 12.

4.1.1 Types

We repeat most of the type formulas from Sect. 3.3.1.1 on page 12.

60. *Double-entry Bookkeeping Accounts* are pairs of *debit/credit accounts* and *asset/liability accounts*.
61. *Asset/Liability Accounts* are pairs of *Asset Accounts* and *Liability Accounts*.
62. *Asset Accounts* are *account triplets*.
63. *Liability Accounts* are *account triplets*.

type
60. DBL_ENTRY_ACCOUNT = DC_ACCOUNT × AL_ACCOUNT
[ι 33 π 12]. DC_ACCOUNT = DEBIT_ACCOUNT × CREDIT_ACCOUNT
[ι 34 π 12]. DEBIT_ACCOUNT = ACCOUNT_TRIPLET
[ι 35 π 12]. CREDIT_ACCOUNT = ACCOUNT_TRIPLET
61. AL_ACCOUNT = ASSET_ACCOUNT × LIABILITY_ACCOUNT
62. ASSET_ACCOUNT = ACCOUNT_TRIPLET
63. LIABILITY_ACCOUNT = ACCOUNT_TRIPLET
[ι 36 π 12]. ACCOUNT_TRIPLET = BUDGET × s_entries:ENTRIES × s_total:AMOUNT
[ι 37 π 12]. BUDGET = **Nat**
[ι 38 π 12]. ENTRIES = Account_Name $\overrightarrow{m}$ s_entries:(ENTRY_LIST | ENTRY_MAP)
[ι 39 π 12]. ENTRY_LIST = BUDGET × s_entry_list:ENTRY* × s_sub_total:AMOUNT
[ι 40 π 12]. ENTRY_MAP = s_budget:BUDGET × MAP × s_total:AMOUNT
[ι 40 π 12]. MAP = Acc_Name $\overrightarrow{m}$ ACCOUNT_TRIPLET
[ι 42 π 12]. ENTRY = s_time:$\mathbb{TIME}$ × s_text:E_Text × s_amount:AMOUNT
[ι 15 π 9]. $\mathbb{TIME}$ = ...
[ι 44 π 12]. E_Text = ...
[ι 45 π 12]. AMOUNT = **Nat**

Please observe the recursion in formula [ι 41 π 12] "back to" formula [ι 36 π 12] above.

4.1.2 Wellformedness

We refer to Sects. 3.3.1.2 on page 13 and 3.3.1.3 on page 14 The signature of the function *access paths* need be adjusted:

4.1.2.1 Access Paths

4.1.2.1.1 Common Constraints

value
47.′ access_paths:
47.′ (DEBIT_ACCOUNT|CREDIT_ACCOUNT|ASSET_ACCOUNT|LIABILITY_ACCOUNT)
47.′ → Access_Path-**set**

[ι 53 π 14] The *account names* of any access path are distinct.

axiom [Distinctness of Account Names, I]
[ι 53 π 14]. $\forall$ ap:Access_Path • **card elems** ap = **len** ap

[ι 54 π 15] Any two distinct *access paths*, if they share an *account name* then it is the first element of these *access paths*.

axiom [Distinctness of Account Names, II]
[ι 54 π 15]. $\forall$ ap,ap′:Access_Path •
[ι 54 π 15]. **elems** ap $\cap$ **elems elems** ap′ $\neq$ {}
[ι 54 π 15]. $\Rightarrow$ **hd** ap=**hd** ap′ $\wedge$ {**hd** ap}=**elems** ap $\cap$ **elems elems** ap′

The *debit* and the *credit accounts* have exactly the same *access paths* [ι 55 π 15], and *debit/credit account paths* are "fully distinct"[9] from *asset/liability account paths*, informally:

axiom [Distinctness of Account Names, III]
[ι 55 π 15]. $\forall$ deb_acc:DEBIT_ACCOUNT,cre_acc:CREDIT_ACCOUNT
[ι 55 π 15]. • access_paths(deb_acc) = access_paths(cre_acc) $\wedge$
[ι 55 π 15]. $\forall$ ass_acc:ASSET_ACCOUNT,lia_acc:LIABILITY_ACCOUNT
[ι 55 π 15]. • access_paths(ass_acc) = access_paths(lia_acc) $\wedge$
[ι 55 π 15]. access_paths(deb_acc) $\cap$ access_paths(ass_acc) = {}

4.1.2.1.2 Double-entry Constraints

64. The *access paths* of *debit/credit* and of *asset/liability* accounts are identical.[10]

axiom [Sameness of Debit/Credit and Asset/Liability Access Paths]
64. $\forall$ ((deb_acc,cre_acc),(ass_acc,lai_acc)):
64. ((DEBET_ACCOUNT×CREDIT_ACCOUNT)×(ASSET_ACCOUNT×LIABILIY_ACCOUNT))
64. • access_paths(deb_acc) = access_paths(cre_acc)
64. $\wedge$ access_paths(ass_acc) = access_paths(lia_acc)

65. The set of *account names* of *debit/credit* and of *asset/liability* accounts are distinct.[11]

66. We define the auxiliary function: account_names.

value
66. account_names: ACCOUNT_TRIPLET → Acc_Name-**set**
66. account_names(acc_trip) ≡
66. **let** acc_pths = access_paths(s_entries(acc_trip)) **in**
66. $\cup$ { $\cup$ { **elems** pth | pth:Acc_Path • pth $\in$ acc_pths } }
66. **end**
axiom [Distinctness of Debit/Credit and Asset/Liability Account Names]
65. $\forall$ ((deb_acc_trip,cre_acc_trip))((ass_acc_trip,lia_acc_trip))

[9] — must be made more clear
[10] Cf. 3.3.1.3 on page 14
[11] Cf. 3.3.1.3 on page 14

65. • (DEBIT_ACCOUNT×DEBIT_ACCOUNT)×(DEBIT_ACCOUNT×DEBIT_ACCOUNT)
65. (account_names(deb_acc_trip) = account_names(cre_acc_trip)
65. ∧ account_names(ass_acc_trip) = account_names(lia_acc_trip))
65. ∧ (account_names(deb_acc_trip) ∪ account_names(cre_acc_trip))
65. ∩ (account_names(ass_acc_trip) ∪ account_names(lia_acc_trip)) = {}

4.1.2.2 Budgets

We refer to lines [10–16] of the caption of Fig. 1 on page 13.

67. The *budget* of an ACCOUNT_TRIPLET must equal the summation of the *budget*s of the BUDGETs of the ENTRY_LIST or the *ENTRY_MAP*.

This constraint looks "innocent", at first. But since it applies to recursively embedded ACCOUNT_TRIPLETs it is quite powerful. So we express it as a universal predicate over ACCOUNT_TRIPLETs rather than trying to figure out a recursively, first descending, then ascending, re-tracking, function. [Try formulate such a function !]

axiom [Budgets]
67. ∀ (b,e,_):ACCOUNT_TRIPLET • b = budget_sum(e)
value
67. budget_sum: ENTRIES → AMOUNT
67. budget_sum(e) ≡
67. **case** e **of**
67. [ ] → 0,
67. [a↦elom]∪ e' → entry_sum(elom)+budget_sum(e')
67. **end**

67.' entry_sum: (ENTRY_LIST|ENTRY_MAP) → AMOUNT
67.' entry_sum(elom) ≡
67.' is_ENTRY_LIST(elom) → list_sum(s_entry_list(elom)),
67.' is_ENTRY_MAP(elom) → map_sum(s_entry_list(elom))

67." list_sum: ENTRY_LIST → AMOUNT
67." list_sum(el) ≡ sum(el) [cf.[ι 19 π 10].$'$]

67."' map_sum: ENTRY_MAP → AMOUNT
67."' map_sum(em) ≡
67."' **case** em **of**
67."' [ ] → 0,
67."' [a↦(b,_,_)]∪ em' → b + map_sum(em')
67."' **end**

4.1.2.3 Amounts

68. The *amount* of an ACCOUNT_TRIPLET must equal the summation of the *amounts*s of the BUDGETs of the ENTRY_LIST or the *ENTRY_MAP*.

axiom [Amounts]
68. ∀ (_,e,a):ACCOUNT_TRIPLET • a = amount_sum(e)
value
68. amount_sum: ENTRIES → AMOUNT
68. amount_sum(e) ≡
68. **case** e **of**
68. [ ] → 0,
68. [a↦elom]∪ e‘ → amount_entry_sum(elom)+amount_sum(e‘)
68. **end**

68.‘ amount_entry_sum: (ENTRY_LIST|ENTRY_MAP) → AMOUNT
68.‘ amount_entry_sum(elom) ≡
68.‘ is_ENTRY_LIST(elom) → amount_list_sum(s_entry_list(elom)),
68.‘ is_ENTRY_MAP(elom) → amount_map_sum(s_entry_list(elom))

68.“ amount_list_sum: ENTRY_LIST → AMOUNT
68.“ amount_list_sum(el) ≡ sum(el) [cf.[ι 19 π 10].′]

68.“‘ amount_map_sum: ENTRY_MAP → AMOUNT
68.“‘ amount_map_sum(em) ≡
68.“‘ **case** em **of**
68.“‘ [ ] → 0,
68.“‘ [a↦(_,_,a)]∪ em‘ → a + amount_map_sum(em‘)
68.“‘ **end**

4.1.2.4 Balance

69. By a *balance* of DC_ACCOUNT or a AL_ACCOUNT

70. we shall mean the difference between their *budgets* and *amounts*.

value
69. balance: ACCOUNT_TRIPLET → **Int**
70. balance(budget,_,amount) ≡ budget −− amount

4.1.2.5 Intentional Pull

71. The balances of the DC_ACCOUNT and the AL_ACCOUNT of a *double-entry bookkeeping* system must equal !

Well, there is no guarantee that the accounts balance ! Only *proper accountancy* and *audit* might secure that !

value
71. proper_accountancy: ENTRY_ACCOUNT → **Bool**
71. proper_accountancy(dc_acc,al_acc) ≡ balance(dc_acc)=balance(al_acc)

This constraint is the "hall-mark" of *double-entry bookkeeping* systems !

4.2 Transactions

4.2.1 Read

72. To *read*, is to [screen] "display" an *account entry* of a *double-entry bookkeeping* system - given an *access path* to either a *debit/credit account* or an *access/liability account* for that system.

value
72. view: DBL_ENTRY_ACCOUNT × (DCorAL × Access_Path) → (ENTRY_LIST | ENTRY_MAP)
type
72. DCorAL = ″dc″ | ″al″
value
72. read((dca,ala),(dcoral,ap)) ≡
72. **case** dcoral **of**
72. ″dc″→access(ap,dca), cf. [ι 56 π 15]
72. ″al″→access(ap,ala) cf. [ι 56 π 15]
72. **end**
72. **pre**: dcoral=″dc″→ap ∈ access_paths(dca),_→ap ∈ access_paths(ala)

4.2.2 Write

To *write* is to *insert a new entry* is an ENTRY_LIST, that is, at the end of the *view*ed entry.

Writes can occur to either a *debit/credit account* or to an *asset/liability* account. *Updating* a *debit/credit account* usually requires a corresponding one or more *updates* to the *asset/liability account*.

This is required in order to maintain the *intentional pull* of the *double-entry bookkeeping* system. Cf. Sect. 4.1.2.5 on the facing page.

We model *writes* follows:

73. To *write* syntactically takes (i) an indication as to whether the update is to that of a debit account, to a credit account, to an asset or to a liability account, (ii) an access path and (iii) the text and (iv) amount with which to update the accessed entry.

74. Semantically the *write* occurs in the context of a *double-entry bookkeeping* system and yields such a system.

75. We express the effect of a *write* to a *double-entry bookkeeping* system (dca,ala) **as** that of yielding a changed *double-entry bookkeeping* system (dca′,ala′).

76. The "difference" between (dca,ala) and (dca′,ala′) is expressed in the **where** predicate.

77. The *access paths* are unchanged.

78. A time, τ, is recorded.[12]
79. Either the *write* is to a *debit/credit account* or it is to an *asset/liability account*.
    (a) If to a *debit/credit account* then the *asset/liability account* is unchanged.
    (b) For all *accesses*, ap′,
    (c) to the *debit/credit account* other than the prescribed (to be updated) entry,
    (d) the entries are unchanged.
    (e) For the accessed *entry list* their sub-entries differ as follows:
    (f)
    - the budget is unchanged;
    - the entry list extended with suffix triplet of
        - the time stamp;
        - a text; and
        - an amount;
    - and the entire entry list amount is adjusted accordingly.
80. A similar [**where**] predicate applies to *asset/liability accounts*

type
73. Write :: mkWrite(D_C_A_L,Access_Path,E_Text,AMOUNT)
73. D_C_A_L = ″`da`″ | ″`ca`″ | ″`aa`″ | ″`la`″
value
74. write: Write $\to$ DBL_ENTRY_ACCOUNT $\to$ DBL_ENTRY_ACCOUNT
75. write(dcal,ap,txt,a)(dca,ala) **as** (dca′,ala′)
76. **where**
77. access_paths(dca)=access_paths(dca′) $\wedge$ access_paths(ala)=access_paths(ala′)
78. $\wedge$ **let** τ = **record**_$\mathbb{TIME}$() **in**
79. **case** dcal **of**
79a. ″`da`″ $\to$ ala′ = ala $\wedge$
79b. $\forall$ ap′ •
79c. ap′ $\in$ Access_Path(read((dca,_),(″`dc`″,ap)))$\setminus$\{ap\}
79d. $\Rightarrow$ read((dca,_),(″`da`″,ap′)) = read((dca,_),(″`da`″,ap))
79e. $\wedge$ **let** (b,el,am) =read((dca,_),(″`da`″,ap)), (b′,el′,am′) = read((dca,_),(dcoral,ap)) **in**
79f. b=b′ $\wedge$ el′=el$^\frown\langle$(τ,txt,a)$\rangle$ $\wedge$ am′ = am + a **end**
80. ″`ca`″ $\to$ [similarly !]
80. ″`aa`″ $\to$ [similarly !]
80. ″`la`″ $\to$ [similarly !]
79. **end**
78. **end**
74. **pre**: dcal $\in$ \{″`da`″,″`ca`″\} $\to$ ap $\in$ access_paths(dca), _ $\to$ ap $\in$ access_paths(ala)

The above model is inspired by the storage model – for such languages as `PL/I`, `Algol 68`, `CHILL` and `Ada` [20, 3, 1, 4] – put forward by Hans Bekič and Kurt Walk [2].

[12] **record**_$\mathbb{TIME}$() is a “built-in” primitive of the description language.

4.2.3 Establish Accounts

So far the bookkeeping operations were concerned with established, fixed access path accounts. In this section we shall suggest an *establish accounts* command. Account structures, their composition of *entry lists* and *entry maps*, can be fully described by their *access paths*. Additionally, by supplying, for each access path a *budget*, one have said all there is to say about any "freshly opened" year accounts book !

4.2.3.1 Command Syntax.

81. Syntactically the *establish accounts* command consists of a pair: *establish debit/credit accounts* and *establish asset/liability accounts*
82. An *establish debit/credit accounts*, respectively
83. an *establish asset/liability accounts*.
84. Each of these map *access paths* to *budgets*.
85. The implied sets of *access paths* form distinct sets as outlined above.

type
81. Estab_Accounts :: mkEstablish(dc_accs:Estab_DC_Accounts,al_accs:Estab_AL_Accounts)
82. Estab_DC_Accounts = Access_Path $\overrightarrow{m}$ Budget
83. Estab_AL_Accounts = Access_Path $\overrightarrow{m}$ Budget
value
85. access_paths: Estab_Accounts → Access_Path-**set** × Access_Path-**set**
85. ∀ (dc_accs,al_accs):Establish_Accounts •
85. **dom** dc_accs ∩ **dom** al_accs = {}
85. ∧ { ans(p) | p:Access_Path • p ∈ **dom** dc_accs } ∩ { ans(p) | p:Access_Path • p ∈ al_accs } = {}

85. ans: Access−Path → Account_Name-**set**
85. ans(p) ≡ **elems** p, **pre**: **len** p = **card** p

The *establish account budgets*, thus, are only ascribed to *entry list accounts*.

4.2.3.2 Command Semantics.

The establish_accounts command can be narrated as follows:

86. It takes an Estab_Accounts command
87. and yields (**as**) a pair of (pairs of) *debit/credit* and *asset/liability accounts*. These four accounts
88. have their access paths "select" *account triplets*
    - all of whose *budgets* are those of the command,
    - all of whose *entry lists* are empty, and
    - all of whose *entry list amounts* are zero (0).

89. The intermediate budgets of entry maps conform to the constraints expressed in Sect. 4.1.2.3 on page 20.

value
86. establish_accounts: Estab_Accounts
87. $\quad\rightarrow$ (DEBIT_ACCOUNT$\times$CREDIT_ACCOUNT)$\times$(ASSET_ACCOUNT$\times$LIABILITY_ACCOUNT)
87. establish_accounts(mkEstablish(dc_accs,al_accs)) **as** ((dacc,cacc),(aacc,lacc))
where:
88. $\forall$ p:Access_Path • p $\in$ **dom** dacc $\Rightarrow$ access(p,dacc) = (daac(p),$\langle\rangle$,0)
88. $\wedge$ $\forall$ p:Access_Path • p $\in$ **dom** cacc $\Rightarrow$ access(p,cacc) = (caac(p),$\langle\rangle$,0)
88. $\wedge$ $\forall$ p:Access_Path • p $\in$ **dom** aacc $\Rightarrow$ access(p,aacc) = (aacc(p),$\langle\rangle$,0)
88. $\wedge$ $\forall$ p:Access_Path • p $\in$ **dom** aacc $\Rightarrow$ access(p,lacc) = (lacc(p),$\langle\rangle$,0)
89. $\wedge$ [ι 68 π 20]

4.2.4 Save Accounts

5 A Financial Management Prototype Domain

We refer to [10, *Domain Modelling*].

There are three main subsections of this section.

In Sect. 5.1 we "embed" the model of *double-entry bookkeeping*, of Sect. 4, in "a[ny]" domain, focusing on *domain endurants*. In Sect. 5.2 we present the *domain facet* of *transaction scripts* – cf. [6, *Chapter 8: Domain Facets*]. And in Sect. 5.3 we focus on *domain perdurants*: especially the behaviours that can be *transcendentally deduced* from *endurant parts*.

5.1 Endurants

Endurants can be considered in two stages. The *external qualities* and the *internal qualities* stages. The latter can be considered in three sub-stages. The *unique identification*, the *mereologies*, and the *attributes* stages.

5.1.1 External Qualities

External qualities will be considered in two steps. The endurant sorts, and the endurant values steps.

5.1.1.1 Endurant Sorts

90. From any domain, cf. [9, *Domain Models A Compendium*], we can, besides the "core" of the domain, observe:

91. its *management*.

From this *management* we can observe:

92. the *double-entry account* and

93. its *accountancy*.

From the *double-entry account* we can observe the

94. *debit/credit account*, and the
95. *asset/liability account*.

From the *accountancy* we can observe:

96. a set of zero, one or more *accountants*.

We leave the *audit[or]* further undefined.

type
90. DOMAIN
91. MGT
92. DEBK = DBL_ENTRY_ACCOUNT
94. DC_ACCOUNT
95. AL_ACCOUNT
93. ACCOUNTANCY
96. ACCOUNTANT
value
91. **obs**_MGT: DOMAIN → MGT
92. **obs**_DEBK: MGT → DEBK
93. **obs**_ACCOUNTANCY: MGT → ACCOUNTANCY
96. **obs**_ACCOUNTANTs: ACCOUNTANCY → ACCOUNTANT-**set**

5.1.1.2 Endurant Values

For use in later descriptions we introduce some relevant endurant values.

97. There is given a *domain*.
98. From its *management* we observe its *double-entry account*.
99. From the *double-entry account* we observe its *debit/credit account*.
100. From the *double-entry account* we observe its *asset/liability account*.
101. From the *management* we can observe the *debit/credit to asset/liability relation*, an *attribute*, DB_AL_REL.
102. From the *management* we can observe observe the *accountancy*.
103. From the *accountancy* a set of *accountants*.

value
97. *domain*:DOMAIN
98. *mgt*:MGT = **obs**_MGT(*domain*)
98. *debk*:DEBK = **obs**_DEBK(*mgt*)
99. *dc_acc*:DC_ACCOUNT = **obs**_DC_ACCOUNT(*debk*)
100. *al_acc*:AL_ACCOUNT = **obs**_AL_ACCOUNT(*debk*)
102. *accountancy*:ACCOUNTANCY = **obs**_ACCOUNTANCY(*mgt*)
103. *accountants*:ACCOUNTANTs = **obs**_ACCOUNTANTs(*accountancy*)
type
120. DB_AL_REL = Access_Path $\xrightarrow[m]{}$ REL

5.1.2 Internal Qualities.

Internal qualities will e considered in three, sequential, sub-stages.[13] The unique identification, the mereologies, and the attributes sub-stages.

5.1.2.1 Unique Identifiers.

Behaviours are uniquely distinguished by he Unique Identifiers of "their parts": $p : P$: **uid**_$P(p)$. So the unique identifier π:UI of p is a static, constant, argument of behaviour behaviour$_P$.

5.1.2.1.1 Unique Identifier Observers and Values

104. There is the type of [all] unique identifiers.
105. There is the unique identifier of the debit/credit account.
106. There is the unique identifier of the asset/liability account.
107. The are the unique identifiers of each of the accountants of the set of accountants.

type
104. UI
value
105. **uid**_DC_ACCOUNT: DC_ACCOUNT → UI
106. **uid**_AL_ACCOUNT: AL_ACCOUNT $->$UI
107. **uid**_ACCOUNTANT: ACCOUNTANT → UI

105. *dci*:UI = **uid**_DC_ACCOUNT(*dc_acc*)
106. *ali*:UI = **uid**_AL_ACCOUNT(*al_acc*)
107. *ais*:UI-**set** = { **uid**_ACCOUNTANT(acc)|acc:ACCOUNTANT•acc∈*accountants* }

5.1.2.1.2 Wellformedness.

All parts of a domain have distinct identification. That is:

108. The number of accountants equals the number of their unique identifiers.
109. And these are distinct from the *debit/credit* and *asset/liability account* identifiers
110. – which are distinct.

axiom [Uniqueness of Parts]
108. **card** $accountants$ = **card** ais ∧
109. $ais \cap \{dci, ali\} = \{\}$ ∧
110. $dci \neq ali$

[13] A usual fourth sub-stage, *'Intentional Pull'* was already considered in Sect. 4.1.2.5 on page 20.

5.1.2.2 Mereologies.

Behaviours communicate with other behaviours. So the mereology of part p indicates with which other behaviours behaviour p interacts. So the mereology **mereo**_P(p), usually modelled as a set of unique identifies, is a [usually] static argument of behaviour behaviour$_P$.

5.1.2.2.1 Mereology Observers

111. The *double-entry debit/credit bookkeeping* behaviour, dbl_dc_book, communicates with a the set of all accountants [a *mereology* argument], and has the *debit/credit account*, dc_acc, as its *programmable* argument.
112. The *double-entry asset/liability bookkeeping* behaviour, dbl_al_book, communicates with just the *asset/liability account*, al_acc [a *mereology* argument], and has the *asset/liability account*, al_acc, as its *programmable* argument.
113. The accountant behaviour communicates with just the *double-entry asset/liability bookkeeping* behaviour [a *mereology* argument], dbl_al_book.
114. The *debit/credit account* relates to the *asset/liability account* and the whole set of all accountants.
115. The *asset/liability account* relates only to the *debit/credit account*.
116. Accountants relate, in this [abbreviated] model, only to the [one] *debit/credit account*.

114. **mereo**_DC_ACCOUNT: DC_ACCOUNT $\rightarrow$ UI $\times$ UI-**set**
115. **mereo**_AL_ACCOUNT: AL_ACCOUNT $\rightarrow$ UI
116. **mereo**_ACCOUNTANT: ACCOUNTANT $\rightarrow$ UI

5.1.2.2.2 Mereology Wellformedness

117. The mereology of *debit/credit account* dc_acc is the pair of the unique identifier of the *asset/liability account* al_acc and the the set of unique identifiers of all the *accountants*.
118. The mereology of the *asset/liability account* al_acc is the pair of the unique identifier of the *debit/credit account*.
119. The mereology of each *accountant* is just that of the *debit/credit account* dc_acc.

axiom [Mereology Constraints]
117. **mereo**_DC_ACCOUNT(dc_acc) $= (dci, ais)$
118. **mereo**_AL_ACCOUNT(al_acc) $= dci$
119. $\forall$ acc:ACCOUNTANT • acc $\in$ $accountants \Rightarrow$ **mereo**_ACCOUNTANT(acc) $= dci$

5.1.2.3 Attributes.

We shall focus on a very few endurant attributes. Attributes [also] become behaviour arguments. Some are *static*, cannot change value. Others are *programmable*, does, indeed, change value.

5.1.2.3.1 Debit/Credit Accounts

pp:Debit Credit Accounts

> [ι 33 π 12]. The [foremost] *debit/credit account endurant* attribute is that of the *debit/credit account*. It is a *programmable attribute.*
>
> [ι 120 π 28] Besides this, the *debit/credit account endurant* has the *static attribute* of the *debit/credit to asset/liability relation*, DB_AL_REL. See *wellformedness* below.

type
[ι 120 π 28]. DB_AL_REL = ... [see below]
value
[ι 33 π 12]. **attr**_DC_ACCOUNT: DC_ACCOUNT → DC_ACCOUNT
[ι 33 π 12]. *dc_acc* = **attr**_DC_ACCOUNT(dc_acc)

120. The *debit/credit to asset/liability relation*, DB_AL_REL, maps *debit/credit access paths* to
121. a map, REL, from *access/liability access paths* to a rational lying properly between 0 and 1,
122. and such that these sum up to 1 !

type
[ι 120 π 28]. DB_AL_REL = Access_Path $\overrightarrow{m}$ REL
121. REL = Access_Path $\overrightarrow{m}$ Rat
value
121. **attr**_DB_AL_REL: MGT → DB_AL_REL
121. *db_al_rel* = **attr**_DB_AL_REL(mgt)
axiom [Proper Management]
120. ∀ db_al_rel:DB_AL_REL • **dom** db_al_rel = ...
122. ∀ rel:REL • **dom** rel = ... ∧ rng_rel_sum(rel)=1.
value
122. rng_rel_sum(rel) ≡ + {r|ap:Access_Path,r:Rat•ap ∈ **dom** rel ∧ r=rel(ap)}
122. **pre**: ∀ r:Rat • r ∈ **rng** rel ⇒ 0<r≤1 ∧ + is the distributed-fix addition operator

The idea behind the DB_AL_REL is explained in Sect. 5.2.1 on the next page.

5.1.2.3.2 Asset/Liability Accounts

> [ι 61 π 17] The [foremost, well only] *asset/liability account endurant* attribute is that of the *asset/liability account*. It is a *programmable attribute.*

value
[ι 61 π 17]. **attr**_AL_ACCOUNT: AL_ACCOUNT → AL_ACCOUNT

5.1.2.3.3 Accountants

123. Each accountant has, in our somewhat "reduced" model, just one *static attribute:* a set og *access paths* partitioning. It lists the debit/credit account access paths that this accountant can read [view] and write.

type
123. Access_Rights = Access_Path-**set**
value
123. **attr**_Access_Rights: ACCOUNTANT → Access_Rights

124. No two accountants share access paths.

125. The set of all accountants possess all access path of the debit/credit account attribute.

axiom [Distinct Access Rights]
124. ∀ acc1,acc2:ACCOUNTANT•{acc1,acc2} ⊆ *accountants*
124. • acc1 ≠ acc2 ⇒ **attr**_Access_Rights(acc1) ∩ **attr**_Access_Rights(acc1) = {}
125. ∪{**attr**_Access_Rights(acc) acc:ACCOUNTANT • acc ∈ *accountants*} = access_paths(*dc_acc*)

5.2 Some Domain Facets.

We refer to [6, *Chapter 8*]. Usually, in the many models of [9], we have not illustrated the concept of *domain facets*. Among *domain facets* we can list

- *support technologies*,
- *rules & regulations*,
- *scripts*,
- *script languages*,
- *management & organization*,
- and *human behaviour*.

The *domain facet* that we shall illustrate is one of *scripts*.

5.2.1 A Complete Transaction.

We refer to the *A Complete Transaction* comment on Page 7.

The idea behind the DB_AL_REL is the following: When a *debit* [or *credit*] entry is posted, for a certain amount, it should be followed by one (or more) *liability* [resp., *asset*] posting(s). For any given *debit* [or *credit*] posting there is one or more specific *liability* [resp., *asset*] posting(s) to be made, each such posting being in the amount of a fraction of the *debit* [or *credit*] posting, with their sum being equal to the *debit* [or *credit*] posting amount. The rational number fractions do not necessarily result in a natural number liability [resp., asset] posting. Hence these must be suitably *"rounded"*.

5.2.1.1 Transaction Syntax, Syntactic Types.

126. A *transaction* is a pair commands: an *debit/credit enter* and a of set of *liability/asset enter* one or more commands —

127. such that these latter conform to the constraints expresses in [ι 128 π 30].

type
126. Transaction = DC_Enter × LA−Enter-**set**
axiom [Well-formed Transaction]
127. [ι 128 π 30] ...

5.2.1.2 Transaction Syntax, Semantic Types.

128. The *actual posting* is thus a map from *debit* [or *credit*] *access path*s to maps from *liability* respectively [*asset*] *access path*s to natural number *amounts*.

type
128. ACT_A_POST = Deb_AccessPath $\overrightarrow{m}$ Lia_A_A_REL
128. ACT_L_POST = Cre_AccessPath $\overrightarrow{m}$ Ass_A_L_REL
128. Lia_A_A_REL = Lia_AccessPath $\overrightarrow{m}$ Amount
128. Ass_A_L_REL = Ass_AccessPath $\overrightarrow{m}$ Amount
128. Deb_AccessPath,Cre_AccessPath,Lia_AccessPath,Ass_AccessPath = AccessPath

5.3 Perdurants

5.3.1 Bookkeeping Channels.

Behaviours interact. *Accountants* communicate *read*, *write* and other commands to the *debit/credit account behaviour*. The *debit/credit account behaviour* communicates *read* and *write* commands to the *asset/liability behaviour*. To "effect" so, in our CSP [16] model, we introduce the abstract notion of *channels*.

129. The abstract notion of bookkeeping channels is here a CSP **channel** *array* whose indices are un-ordered pairs of unique identifiers of *unique accountant*, *debit/credit account* and/or *asset/liability account identifiers* – and

130. a message, MSG – which is either a *read*, a *write*, or some other command.

type
130. MSG = Read | Write | ...
channel
129. { ch[{ui,uj}] | ui,uj:UI • {ui,uj}⊆*uis* } : MSG

5.3.2 Bookkeeping Behaviours.

5.3.2.1 Bookkeeping Perdurants.

We refer to Sect. 5.1.1.2. We shall consider the following domain perdurants to be transcendentally deduced into domain behaviours.

131. a *double-entry debit/credit bookkeeping account* behaviour,
132. a *double-entry asset/liability bookkeeping account* behaviour, and
133. a set of *accountant* behaviours.

131. dc_account [basd on] DC_ACCOUNT [i.e.,] dc_acc
132. al_account: [basd on] AL_ACCOUNT [i.e.,] al_acc
133. accountant: [basd on] ACCOUNTANTs [i.e.,]accountants

5.3.2.2 Bookkeeping Domain Behaviour Signatures.

We shall not follow the *'doctrine"* of expressing the *domain behaviour* signatures strictly according to [6]. That is: We omit a "full treatment" of all attributes. But to remind you:

134. The *debit/credit account* behaviour, dc_account, communicates with a the set of all accountants [a *mereology* argument], and has the *debit/credit account*, dc_acc, as its *programmable* argument.
135. The *asset/liability account* behaviour, al_account, communicates with just the *asset/liability account*, al_acc [a *mereology* argument], and has the *asset/liability account*, al_acc, as its *programmable* argument.
136. The accountant behaviour communicates with just the *double-entry asset/liability bookkeeping* behaviour [a *mereology* argument], dbl_al_book.

value
134. dc_account: UI $\rightarrow$ UI-**set** $\rightarrow$ ... $\rightarrow$ DC_ACCOUNT ... **Unit**
135. al_account: UI $\rightarrow$ UI-**set** $\rightarrow$ ... $\rightarrow$ AL_ACCOUNT ... **Unit**
136. accountant: UI $\rightarrow$ UI $\rightarrow$ (Acces_Path-**set** $\times$ ...) $\rightarrow$ ... **Unit**

5.3.2.3 Bookkeeping Behaviour Definitions.

5.3.2.3.1 The Debit/Credit Account Behaviour

[ι 75 π 21]. We remind the reader of the definition of the write function.

137. The dc_account *behaviour* is here defined without detailing possible [*static* and *monitorable*] arguments (...).
138. The dc_account *behaviour* external non-deterministically, ⎕, awaits write commands from either of the accountant behaviours (cf. [ι 139d π 32]).

These commands are either debit/credit, i.e., write, commands, or a *establish "fresh, new" accounts*, or are ... .

139. If *write* commands
   (a) the dbl_dc_book *behaviour* then performs the write function on the *double-entry bookkeeping's debit/credit account* dc_acc.
   (b) After which it then performs the "corresponding" updates, at least one, possible ["a few"] more, on the *double-entry bookkeeping's asset/liability account* "al".
   (c) After which it "reverts" to being the The dbl_dc_book *behaviour* –
   (d) [with this external non-deterministic actions "ranging" over all accounts]
140. If *view* commands the dbl_dc_book *behaviour* then ...
141. If *errata* commands the dbl_dc_book *behaviour* then ...
142. If *establish* commands the dbl_dc_book *behaviour* then ...
143. If *audit* commands the dbl_dc_book *behaviour* then ...

```
value
137.  dc_account: UI → UI-set → ... → DC_ACCOUNT → Unit
137.  dc_account(dci)(auis)(...)(dc_acc) ≡
139.     ⌈⌉ { let mkWrite(daorca,ap,txt,a) = ch[{dci,aui}] ? in
139a.          let dci_acc′ = write(daorca,ap,txt,a)(dc_acc) in
139b.          update_asset_liability_accounts(daorca)(ap,txt,a);
139c.          dc_account(dci)(auis)(...)(dc_acc′)
139d.          end end | aui:Acc_UI • aui ∈ auis
139.        | aui:Acc_UI • aui ∈ auis }
140.     ⌈⌉ { let mkView(...) = ch[{dci,aui}] ? in
140.        ... end | aui:Acc_UI • aui ∈ auis }
141.     ⌈⌉ { let Errata(...) = ch[{dci,aui}] ? in
141.        ... end | aui:Acc_UI • aui ∈ auis }
142.     ⌈⌉ { let mkEstablish() = ch[{dci,aui}] ? in
142.        ... end | aui:Acc_UI • aui ∈ auis }
143.     ⌈⌉ { let mkAudit(...) = ch[{dci,aui}] ? in
143.        ... end | audit_ii:Aaudit_UI_UI • audit_ui ∈ audit_uis }
```

[ι 121 π 28]. We remind the reader of the **value** definition of *db_al_rel*,

144. To *update the asset/liability account*
145. is to provide the *debit or credit account* "marker": "da","ca", the *debit/credit account path*, some *entry text*, and the *amount* with which the *debit/credit account* was updated and a pair of the *debit or credit access path* and an *amount*.[14]

[14] The time stamp is "provided" at the time point when the actual *asset/liability account* is updated, cf. [ι 78 π 22].

146. When a *debit or credit account* is updated then one or more corresponding *liability*, respectively *asset account*s must be "balanced". The DB_AL_REL "table", *db_al_rel*, serves to indicate with which fractions of the *debit or credit account amount* respective *liability*, respectively *asset account*s shall be "balanced". These fractions may result in non-natural, rational amounts. These are *rounded off* "by" the round function – once and for all,

147. before the *liability*, or *asset account*s are updated.

value
[ι 121 π 28]. *db_al_rel* = **attr**_DB_AL_REL(*mgt*)

144. update_asset_liability_accounts: ("dc"|"ca") → Access_Path×Txt×Amount → **Unit**
145. update_asset_liability_accounts(dcorca)(ap,txt,a) ≡
146. **let** dbalrel = round(*db_al_rel*(ap),a) **in**
147. upd_ass_lia_acc(dcorca)(dbalrel)(ap,txt,a) **end**

148. The *asset/liability account*s update provides a marker, "dc" or "al", as to whether a *liability* or an *asset account* is to be updated – and for that update it provides the *rounded* overall amounts dbalrel, a *liability/asset access path* dc_ap, a suitable entry text, and the amount, a, with which the *debit/credit account* was updated.

149. Either the *rounded* overall amounts dbalrel

150. is "empty", i.e., [], and the updates have been done,

151. or there is an *access path*, ac_ap, for which a *fraction*, f, is to be updated –

152. in which case a *write* command is communicated to the *asset/liability account behaviour* as a *asset* or a *liability update* with the *access path* of the *debit/credit* account that was updated, the update text, and the rounded update amount –

153. whereupon the *update asset/liability account behaviour* resumes being so.

value
148. upd_ass_lia_acc: DCorAL → DBALREL → Access_Path×Txt×Amount → **Unit**
148. upd_ass_lia_acc(dcorca)(dbalrel)(dc_ap,txt,a) ≡
149. **case** dbalrel **of**
150. [ ] → **skip**,
151. [ac_ap↦f] ∪ dbalrel′ →
152. ch[{aci,aui}] ! mkWrite(**if** dcorca = "dc" **then** "al" **else** "la" **end**,dc_ap,txt,dbalrel(ac_ap)) ;
153. upd_ass_lia_acc(dcorca)(dbalrel′)(dc_ap,txt,a)
149. **end**

154. There is a type, Amounts, whose values record the rounded *amount*s that specified *access path* entries are to be updated with.

155. The [auxiliary] round function takes a *debit/credit to asset/liability relation*, an *access path*, and an *amount* and yields the rounded *amounts* for that *access path* in the *debit/credit to asset/liability relation*.

156. The definition sets of the *debit/credit to asset/liability relation* and the *amounts* shall be identical.

157. The sum of *amount* entries in *amounts* shall match the *debit/credit* update amount, a, and

158. the *amounts* range entries must be suitably rounded up or down to a "whole", natural number value.[15]

type
154. Amounts = Access_Path $\overrightarrow{m}$ Amount
value
155. round: REL × Access_Path × Amount → Amounts
155. round(rel,ap,a) **as** amounts
where:
156. **dom** rel = **dom** amounts
157. ∧ a = sum(amounts)
158. ∧ ∀ ap:Access_Path • ap ∈ **dom** rel ⇒ amounts(ap) ∈ $\{\lfloor(\text{rel(ap)}*\text{a})\rfloor,\lceil(\text{rel(ap)}*\text{a})\rceil\}$

159. The $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ are the *floor*, respectively *ceiling* distributed-fix operators.

value
159. $\lfloor\cdot\rfloor,\lceil\cdot\rceil$: Rat → **Nat**

160. We leave it to the reader to "decipher" the sum function !

value
160. sum: Amounts → **Nat**
160. sum(am) ≡ **case** am **of** [ ] → 0, [ap↦a]∪am′→a+sum(am′) **end**

The dbl_al_book behaviour is much like the dbl_dc_book behaviour: a few renamings and item [ι 139b π 32] omitted !

5.3.2.3.2 The Asset/Liability Account Behaviour

5.3.2.3.3 The Accountant Behaviour.

The accountant behaviour is one amongst a definite set of one or more accountants. Each accountant has access to the debit/credit account and, within it, to a distinct set of debit/credit sub-accounts. Each accountant receives copies of debit/credit messages from, as we shall call them, *agents*, and "passes" these on, in the form of *debit* or *credit* "writes" to the dc_account behaviour.

161.

162.

163.

[15] Now, this "rounding" operation is somewhat "doubtful". It must be subject to some statistical distribution, etc., etc. !

164.

165.

166.

```
value
161. accountant: UI → (UI×UI-set) → DC_Paths → Accountant_History → Unit
161. accountant(aci)(dci,agent_uis)(dc_paths)(ahist) ≡
162.    ⌈⌉ { let ag_msg = ch[{dci,aui}] in
163.       let daorca = debit_or_credit(agent_msg,ahist),
163.           ap = access_path(agent_msg,dc_paths),
163.           txt = text(agent_msg,ahist),
163.           a = cost(agent_msg) in
164.       let a_hist′ = update_Accountant_History(ag_msg,aui,record_TIME,daorca)(ahist) in
165.       ch[{aci,dci}] ! mkWrite(daorca,ap,txt,a) ;
166.       accountant(aci)(dci,agent_uis)(dc_paths)(ahist′) end end end
162.      | aui:UI • aui ∈ agent_uis }
```

5.3.2.3.4 The Audit Behaviour.

5.3.3 Initialize System

6 Summing Up

7 Bibliography

References

[1] Anon. *C.C.I.T.T. High Level Language (CHILL), Recommendation Z.200, Red Book Fascicle VI.12*. See [13]. ITU (Intl. Telecmm. Union), Geneva, Switzerland, 1980 – 1985.

[2] Hans Bekič and Kurt Walk. Formalization of Storage Properties. In *Symposium on Semantics of Algorithmic Languages*, volume LNM 188. Springer, 1971.

[3] B.J. Mailloux and J.E.L Peck and C.H.A. Koster and Aad van Wijngaarden. *Report on the Algorithmic Language ALGOL 68*. Springer, Berlin, Heidelberg, 1969.

[4] D. Bjørner and O. Oest. *Towards a Formal Description of Ada*, volume 98 of *LNCS*. Springer-Verlag, 1980.

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

[6] Dines Bjørner. *Domain Science & Engineering – A Foundation for Software Development*. EATCS Monographs in Theoretical Computer Science. Springer, Heidelberg, Germany, 2021. A revised version of this book is [8].

[7] Dines Bjørner. Domain Modelling – A Primer. A short version of [8]. xii+202 pages[16], May 2023.

[8] Dines Bjørner. Domain Science & Engineering – A Foundation for Software Development. Revised edition of [6]. xii+346 pages[17], January 2023.

[9] Dines Bjørner. Domain Models – A Compendium. Internet: `http://www.imm.dtu.dk/~dibj/2024/models/domain-models.pdf`, March 2024. This is a very early draft. 19 domain models are presented.

[10] Dines Bjørner. Domain Models – A Compendium. Internet: `http://www.imm.dtu.dk/~dibj/2024/models/domain-models.pdf`, March 2024. This is a very early draft. 19 domain models are presented.

[11] Dines Bjørner and Yang ShaoFa. Domain Modelling. Technical University of Denmark. Revised edition of [10]. xii+208 pages. https://www.imm.dtu.dk/ dibj/2023/dommod/dommod.pdf, May 2023.

[12] O.-J. Dahl, E.W. Dijkstra, and Charles Anthony Richard Hoare. *Structured Programming*. Academic Press, 1972.

[13] P.L. Haff, editor. *The Formal Definition of CHILL*. ITU (Intl. Telecmm. Union), Geneva, Switzerland, 1981.

[14] Charles Anthony Richard Hoare. Notes on Data Structuring. In *[12]*, pages 83–174, 1972.

[15] Charles Anthony Richard Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985.

[16] Charles Anthony Richard Hoare. Communicating Sequential Processes. Published electronically: `usingcsp.com/cspbook.pdf`, 2004. Second edition of [15]. See also `usingcsp.com/`.

[17] Michael A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley, Reading, England, 1995.

[18] W. Little, H.W. Fowler, J. Coulson, and C.T. Onions. *The Shorter Oxford English Dictionary on Historical Principles*. Clarendon Press, Oxford, England, 1973, 1987. Two vols.

[19] E.C. Luschei. *The Logical Systems of Leśniewksi*. North Holland, Amsterdam, The Netherlands, 1962.

[20] ANSI X3.53-1976. The PL/I programming language. Technical report, American National Standards Institute, Standards on Computers and Information Processing, 1976.

[21] Achille C. Varzi. *On the Boundary between Mereology and Topology*, pages 419–438. Hölder-Pichler-Tempsky, Vienna, 1994.

[22] George Wilson and Samuel Shpall. Action. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, summer 2012 edition, 2012.

[16]This book is currently being translated into Chinese by Dr. Yang ShaoFa, IoS/CAS, Beijing and into Russian by Dr. Mikhail Chupilko, ISP/RAS, Moscow

[17]Due to copyright reasons no URL is given to this document's possible Internet location. A primer version, omitting certain chapters, is [7]

A Software Engineering Terminology

I still need to fill in the below entries — from other documents !

The field of computing science is still youngIts lexicon is not yet firmly established – though [17, *Michael A. Jackson*] is a seminal example. Over the last $\frac{1}{2}$ century I have developed and adhered to the terminology of this section.

The explication of the **highlighted terms** in this appendix are mine in the sense that I adhere, systematically, to these explications. You, the reader, may have some previously conceived understanding of these terms. Please, temporarily, in the context of the present report, forget about Your "previous" understanding. It is not a matter of whether my explications are right or wrong. They are the ones I adhere to. So in the present context they are right !

1. **Abstraction.** By an *abstraction* we shall understand a formulation of some phenomenon or concept of some universe of discourse such that some aspects of the phenomenon or concept are emphasized (i.e., considered important or relevant) while others are left out of consideration (i.e., considered unimportant or irrelevant)

   > Abstraction relates to conquering complexity of systems description through the judicious use of abstraction, where abstraction, briefly, is the act and result of omitting consideration of (what would then be called) details while, instead, focusing on (what would therefore be called) important facets.
   >
   > *Conception, my boy, fundamental brain-work,*
   > *is what makes the difference in all art*
   > D.G. Rossetti[18]: letter to H. Caine[19]
   >
   > In the natural sciences one observes phenomena — and then one abstracts. In programming we create universes, but first abstractly.
   >
   > The following is from the opening paragraphs of C.A.R. Hoare's: *Notes on Data Structuring* [14].
   >
   > > *Abstraction is a tool, used by the human mind, and to be applied in the process of describing (understanding) complex phenomena. Abstraction is the most powerful such tool available to the human intellect. Science proceeds by simplifying reality. The first step in simplification is abstraction. Abstraction (in the context of science) means leaving out of account all those empirical data which do not fit the particular, conceptual framework within which science at the moment happens to be working. Abstraction (in the process of specification) arises from a conscious decision to advocate certain desired objects, situations and processes as being fundamental; by exposing, in a first, or higher, level of description, their similarities and — at that level — ignoring possible differences.*

[18] Gabriel Charles Dante Rossetti, generally known as Dante Gabriel Rossetti, was an English poet, illustrator, painter, translator, and member of the Rossetti family. He founded the Pre-Raphaelite Brotherhood in 1848 with William Holman Hunt and John Everett Millais. Born: May 12, 1828, London, United Kingdom Died: April 9, 1882 (age 53 years), Birchington-on-Sea, United Kingdom

[19] Sir Thomas Henry Hall Caine CH KBE (14 May 1853 31 August 1931), usually known as Hall Caine, was a British novelist, dramatist, short story writer, poet and critic of the late nineteenth and early twentieth century.

2. **Informatics.** Informatics is the confluence of mathematics, computer and computing science.

3. **Mathematics.** Mathematics is the science and study of quality, structure, space, and change. Mathematicians seek out patterns, formulate new conjectures, and establish truth by rigorous deduction from appropriately chosen axioms and definition.

4. **Computing.** Computing is the act of calculating something – adding it up, multiplying it, or doing more complex mathematical, including logical, functions. The verb compute comes from a Latin word for pruning.

5. **Computer.** A computer is a mechanical, electro-mechanical, electrical or electronic device that can be so-called 'programmed' to automatically carry out sequences of arithmetic and logical operations (computation).

6. **Computer Science.** Computer science is the mathematical study and knowledge of the phenomena that "exists inside" computers: data& processes.

7. **Computing Science.** Computing science is the study and knowledge of how to construct the phenomena that "exists inside" computers: data& processes, including methodologies for domain descriptions, requirements prescriptions, software designs and program codes.

Informatics is in contrast to IT, we think: where informatics is a "more-or-less intellectual world" of its "products" being more-or-less appropriate, pleasing, correct, ...; IT is a "more-or-less material world" of its "products" being more-or-less bigger, smaller, faster, cheaper, etc.

8. **Information.** Information is an abstract concept that refers to something which has the power to inform. At the most fundamental level, it pertains to the interpretation (perhaps formally) of that which may be sensed, or their abstractions. Any natural process that is not completely random and any observable pattern in any medium can be said to convey some amount of information. Whereas digital signals and other data use discrete signs to convey information, other phenomena and artifacts such as analogue signals, poems, pictures, music or other sounds, and currents convey information in a more continuous form. Information is not knowledge itself, but the meaning that may be derived from a representation through interpretation. `Wikipedia`

9. **Information Technology.** Hardware and software systems to manage, process, protect, and exchange information.

   (a) **Hardware.** By IT hardware we shall understand the mechanical, electro-mechanical, electric and electronic systems that facilitate computations.

   (b) **Software.** By IT software we shall understand the full set of documents that record the full computing science development of domain descriptions, requirements prescriptions, software design and program code: management plans, including budgets and accounts, manpower resources and their deployment, all validation documents: testing, model checking and theorem proofs, all maintenance records: corrective, preventive, perfective, etc., etc.

10. **IT.** Same as information technology.

11. **The Triptych Dogma.** In order to *specify* $\mathbb{S}$oftware, we must understand its $\mathbb{R}$*equirements*. In order to *prescribe* $\mathbb{R}$*equirements* we must understand the $\mathbb{D}$*omain*. So we must study, analyze and describe $\mathbb{D}$omains. $\mathbb{D},\mathbb{S} \models \mathbb{R}$[20]

12. **Method.** By a **method** we shall understand a set of **principles** and **procedures** for selecting and applying a set of **techniques** and **tools** to a problem in order to achieve an orderly construction of a **solution**, i.e., an **artifact**.

13. **Methodology.** By **methodology** we shall understand the *study & application* of one or more methods.

14. **Formal Method.** By a formal method we shall mean a method whose techniques and tools can be given a mathematical meaning.

15. **Principle:** By a **principle** we mean: *a principle is a proposition or value that is a guide for behavior or evaluation* `Wikipedia`*, i.e., code of conduct.*

16. **Procedure.** By a **procedure** we mean: *instructions or recipes, a set of commands that show how to achieve some result, such as to prepare or make something* `Wikipedia`*, i.e., an established way of doing something*.

17. **Technique.** By a **technique** we mean: *a technique, or skill, is the learned ability to perform an action with determined results with good execution often within a given amount of time, energy, or both* `Wikipedia`*, i.e., a way of carrying out a particular task*.

18. **Tool.** By a **tool** we mean: *a tool is an object that can extend an individual's ability to modify features of the surrounding environment.* `Wikipedia`

19. **Language.** The principal method of human communication, consisting of words used in a structured and conventional way and conveyed by speech, writing, or gesture. More specifically, in this document, the set of words (terms) structured in some form of syntax, adhering, more-or-less to some form of semantics, and formed and communicated with some form of pragmatics in mind.

    Animals with higher social interaction uses *sign*s, eventually developing a *language*. These languages adhere to the same system of defined concepts which are a prerequisite for any description of any world: namely the system that philosophy lays bare from a basis of transcendental deductions and the *principle of contradiction* and its *implicit meaning theory*. A *human* is an animal which has a *language*.

    Homo sapiens, in early forms, have existed, some estimate, for millions of years. And, apparently, according to some archaeologists/linguists, did communicate by means of language, but with no abstractions, no metaphors, no concepts. These archaeologists/linguists think that abstractions first came into human language some 15.000 to 35.000 years ago. And that this marks humans from other animals and explains why humans effected societal development in its broadest terms.

20. **Formal Language.** By a formal language we shall understand a language whose syntax and semantics can be expressed a formal, mathematical manner.

[20] In proofs of $\mathbb{S}$oftware correctness, with respect to $\mathbb{R}$equirements, assumptions are made with respect to the $\mathbb{D}$omain.

21. **Semiotics.** By **semiotics** we understand the study of the *Pragmatics*, the *Semantics* and the *Syntax* of languages.

22. **Syntax.** By **syntax** we understand the rules for and form of structures, be they sentential or otherwise.

    By a **formal syntax** we understand a syntax such that we can also analyse sentential structures wrt. their possibly ambiguous composition.

    (a) **Discussion:**

    By sentential structures we mean sequences of characters such as you are reading right now, and such as those of the formulas: Expressions and statements of specification and programming languages.

    By 'other' structures we mean atomic and composite values such as those of **RSL, Java** and other specification or programming languages, or of other mathematical systems: Algebras, logics, etc.

    Syntax is about form, not content: "Appearance", not meaning. I can express the number seven in many different ways:

    $$7, \text{seven}, \text{vii}, ııııııı, 00111, 13,$$

    that is: As an arabic-like numeral, spelled out in letters, as a roman numeral, as a sequence of seven "strokes", as a binary numeral, or as a radix four numeral !

    There may be many syntactic instances signifying the "same thing" (as here the number seven), but one may say that there is exactly one (instance of the) number (that we name) seven !

    (b) **Concrete Syntax:** By a concrete syntax we mean a grammar for specifying strings of sentences (sequences of characters), or for specifying layout of two-dimensional diagrams (pictures) — both as communicated between people, or for specifying the bit and byte-wise layout of storage cells for structured values, etc.

    By the "etcetera" we are appealing to your intuition.

    Natural languages do not have precise means of specifying the exact set of (syntactically) "correct" sentences. But programming and specification languages have. In fact: A formal language is a language which has a precise way of delineating all, and only its correct, ie. allowable sentences.

    We speak of the concrete forms of communicating between humans, of a humans presenting to computers, such mathematical formulas, respectively programs and specifications, as sentences subject to concrete syntax, ie. a grammar. We shall use the term grammar to mean a syntax for a concrete representation.

    (c) **BNF Grammar** BackusNaur form (BNF or Backus normal form) is a notation used to describe the syntax of programming languages or other formal languages. It was developed by John W. Backus and Peter Naur. BNF can be described as a metasyntax notation for context-free grammars. BackusNaur form is applied wherever exact descriptions of languages are needed, such as in official language specifications, in manuals, and in textbooks on programming language theory. BNF can be used to describe document formats, instruction sets, and communication protocols.

23. **Semantics.** The branch of linguistics and logic concerned with meaning. The two main areas are logical semantics, concerned with matters such as sense and reference and presupposition and implication, and lexical semantics, concerned with the analysis of word meanings and relations between them [`Oxford Languages`]. We shall be concerned with lexical semantics in this paper.

24. **Pragmatics.** In linguistics and related fields, pragmatics is the study of how context contributes to meaning. The field of study evaluates how human language is utilized in social interactions, as well as the relationship between the interpreter and the interpreted [Wikipedia]. We shall not really be concerned with pragmatics in this paper.

25. **Transcendense.** is the basic ground concept from the word's literal meaning (from Latin), of climbing or going beyond, albeit with varying connotations in its different historical and cultural stages. [`Wikipedia`]
    (a) **Transcendental.** By *transcendental* we shall understand the philosophical notion: *the a priori or intuitive basis of knowledge, independent of experience*.
    (b) **Transcendental Deduction.** By a *transcendental deduction* we shall understand the philosophical notion: *a transcendental "conversion" of one kind of knowledge into a seemingly different kind of knowledge.*

26. **Science.** Science is the pursuit and application of knowledge and understanding of the natural and social world following a systematic methodology based on evidence [Science Council, UK].

27. **Engineering.** Engineering is the practice of using natural science, mathematics, and the engineering design process to solve technical problems, increase efficiency and productivity, and improve systems [Wikipedia].

28. **Domain Engineering.** The [computing science "inspired"] engineering of constructing *domain descriptions.* See [6, 11].

29. **Requirements Engineering.** The [computing science "inspired"] engineering of deriving *requirements prescriptions* from *domain descriptions.* See [6, *Chapter 9*].

30. **Program Engineering.** The [computing science "inspired"] engineering of deriving *program code* from *requirements prescriptions.* See [5].

31. **Domain.** By a *domain* we shall understand a *rationally describable* segment of a *discrete dynamics* fragment of a *human assisted* reality: the world that we daily observe – in which we work and act, a reality made significant by human-created entities. The domain embody *endurants* and *perdurants*

32. **Domain Analysis.** The analysis of domains according, as we see it, to the following scheme:

33. **Ontology.** A set of concepts and categories applicable to a suitable subject area or domain that shows their properties and the relations between them [`Oxford Languages`].

34. **Taxonomy** is the practice and science of categorization or classification of of specific domain instance `Wikipedia`.

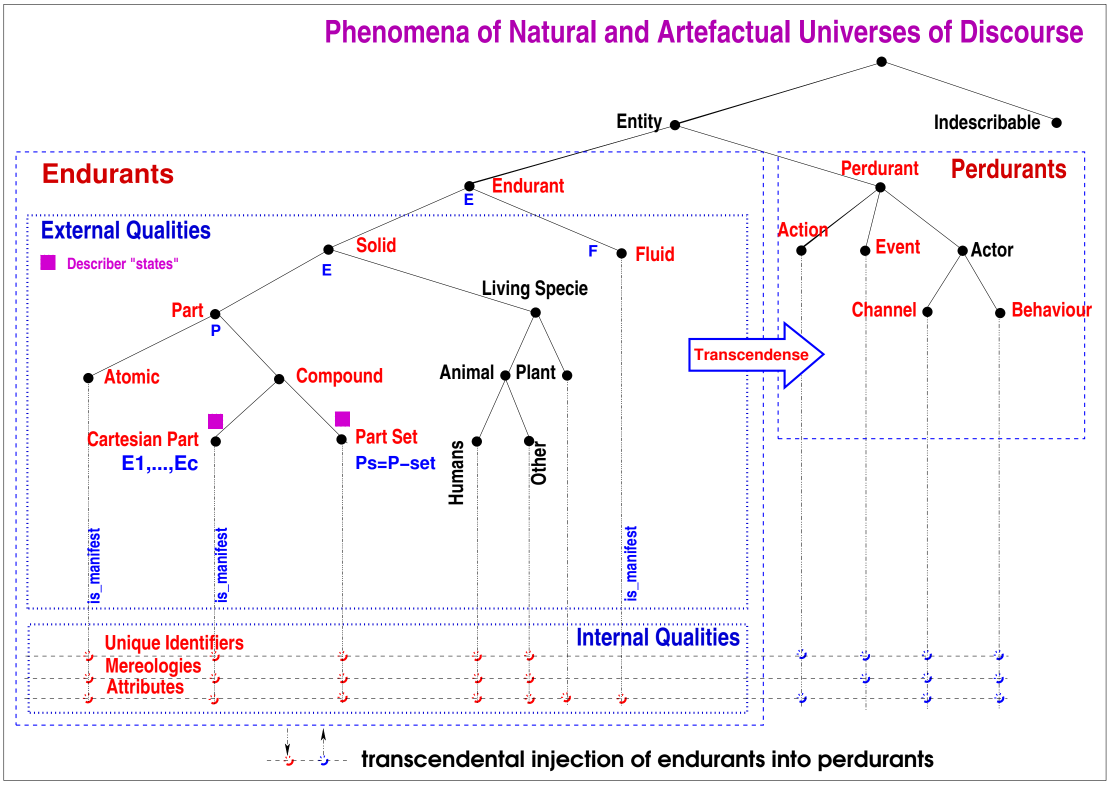

Figure 2: A Domain Analysis & Description Ontology

35. **Phenomenon.** By a *phenomenon* we shall understand a fact that is observed to exist or happen.

36. **Entity.** By an *entity* we shall understand a more-or-less rationally describable phenomenon.

    [**is**_`entity` is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *phenomenon*, ϕ, yields the Boolean truth value **true** if the *phenomenon* is a more-or-less rationally describable.].

37. **Domain Endurants** are those quantities of domains that we can observe (see and touch), in *space*, as "complete" entities at no matter which point in *time* – "material" entities that persists, endures – capable of enduring adversity, severity, or hardship `[Merriam Webster]`.

    [**is**_`endurant` is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *entity, e*, yields the Boolean truth value **true** if the *entity* is a an *endurant*.]

    Endurants may be either *solid* (discrete) or *fluid*, and solid endurants, called *parts*, may be considered *atomic* or *compound* parts. or solid endurants may be further analyzed *living species: plants* and *animals* – including *humans*.

38. **External Qualities** External qualities of endurants of a manifest domain are, in a simplifying sense, those we can see, touch and have spatial extent. They, so to speak, take form.

39. **Internal Qualities** are those properties [of endurants] hat do not occupy *space* but can be measured or spoken about, that is properties which we cannot see but can measure,

mechanically, chemically, electrically, electronically, or otherwise.

40. **Solid** By a *solid* [or *discrete*] endurant we shall understand an endurant which is separate, individual or distinct in form or concept, or, rephrasing: have 'body' [or magnitude] of three-dimensions: length, breadth and depth [18].

    [**is**_`solid` is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *endurant,* e*,* yields the Boolean truth value **true** if the *endurant* is a a *solid*.]

41. **Fluid** By a *fluid endurant* we shall understand an endurant which is prolonged, without interruption, in an unbroken series or pattern; or, rephrasing: a substance (liquid, gas or plasma) having the property of flowing, consisting of particles that move among themselves [18, Vol. I, pg. 774].

    [**is**_`fluid` is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *endurant,* e*,* yields the Boolean truth value **true** if the *endurant* is a a *fluid*.]

42. **Part.** By a [physical] *part* we shall understand a discrete endurant existing in time and subject to laws of physics, including the *causality principle* and *gravitational pull*

    [**is**_`part` is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *solid,* s*,* yields the Boolean truth value **true** if the *solid* is a part.]

    (a) **Manifest Part.** A manifest part is a part, a discrete endurant, which the domain engineer chooses to describe as consisting of one or more endurants, whether discrete or continuous, but to **indeed** endow with internal qualities: unique identifiers, mereology or attributes.

    [**is**_`manifest` is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *part,* p*,* yields the Boolean truth value **true** if the *part* is a manifest.]

    (b) **Structure Part.** A structure part is a part, a discrete endurant, which the domain engineer chooses to describe as consisting of one or more endurants, whether discrete or continuous, but to **not** endow with internal qualities: unique identifiers, mereology or attributes.

    [**is**_`structure` is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *part,* p*,* yields the Boolean truth value **true** if the *part* is a a structure.]

43. **Living Species.** By a *living species* we shall understand a discrete endurant, subject to laws of physics, and additionally subject to *causality of purpose.*

    [**is**_`living_species` is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *solid,* s*,* yields the Boolean truth value **true** if the *solid* is a living species.]

    (a) **Plant.** We refer to the initial definition of *living species* above – while emphasizing the following traits: (i) *form animals can be developed to reach*; (ii) *causally determined to maintain*. (iii) *development and maintenance* in an *exchange of matter with an environment*, and (iv) *ability to purposeful movement*.

    [**is**_`plant` is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *living species,* ℓ*,* yields the Boolean truth value **true** if the *living species* is a plant.]

    (b) **Animal.** We refer to the definition of *living species* above – while emphasizing the following traits: (i) *form animals can be developed to reach*; (ii) *causally determined* *to maintain*. (iii) *development and maintenance* in an *exchange of matter with an environment*, and (iv) *ability to purposeful movement*.

[**is**_animal is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a yields the Boolean truth value **true** if the is a .]

(c) **Human.** A *human* (a *person*) is an *animal*, see above, with the additional properties of having *language*, being *conscious* of *having knowledge* (of its own situation), and *responsibility*.

[**is**_human is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *living species, ℓ,* yields the Boolean truth value **true** if the *living species* is a human.]

44. **Atomic Part.** Atomic parts are those which, in a given context, are deemed to **not** consist of meaningful, separately observable proper *sub-part*s. A *sub-part* is a *part*.

    [**is**_atomic is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *part, p,* yields the Boolean truth value **true** if the *part* is a atomic.]

45. **Compound Part.** Compound parts are those which are observed to [potentially] consist of several parts.

    [**is**_compound is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *part, p,* yields the Boolean truth value **true** if the *part* is a compound.]

46. **Cartesian** Cartesian parts are those compound parts which are observed to consist of a definite number of (two or more) distinctly sort-named endurants (solids or fluids).

    [**is**_Cartesian is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *compound part, p,* yields the Boolean truth value **true** if the *compound* is a Cartesian.]

47. **Set Part** Part sets are those compound parts which are observed to consist of an indefinite number of zero, one or more "similar" parts.

    [**is**_part_set is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *compound part, p,* yields the Boolean truth value **true** if the *compound* is a set of parts.]

48. **Unique Identifiers.** A unique identity is an immaterial property that distinguishes any two *spatially* distinct solids.

    [**uid**_P is an informal function prompt, i.e., a function which when applied, by the domain analyzer, to a *manifest part*, yields *the unique identifier of that part*]

49. **Mereologies.** Mereology is a theory of [endurant] part-hood relations: of the relations of an [endurant] parts to a whole and the relations of [endurant] parts to [endurant] parts within that whole.[21]

    [**mereo**_P is an informal function prompt, i.e., a function which when applied, by the domain analyzer, to a *manifest part*, yields *yields the mereology of that part*]

50. **Attributes.** Attributes are properties of endurants that can be measured either physically (by means of length (ruler) and spatial quantity measuring equipment, electronically, chemically, or otherwise) or can be objectively spoken about.

    [**attributes**_P is an informal function prompt, i.e., a function which when applied, by the domain analyzer,

[21] Mereology in this sense was first studied by the Polish mathematician and philosopher Stanisław Leśniewski (1886–1939) [19, 21].

to a *part*, yields *a set of one or more attribute names* (ηA_1,ηA_2,...,ηA_n)]

[**attr**_A is an informal function prompt, i.e., a function which when applied, by the domain analyzer, to a *manifest part*, yields *the value of attribute* A *for that part*]

Michael A. Jackson [17] has suggested a hierarchy of attribute categories: from *static* (is_static) to *dynamic* (is_dynamic) values – and within the dynamic value category: *inert* values (is_inert), *reactive* values (is_reactive), *active* values (is_active) – and within the dynamic active value category: *autonomous* values (is_autonomous), *biddable* values (is_biddable), and *programmable* values (is_programmable) .

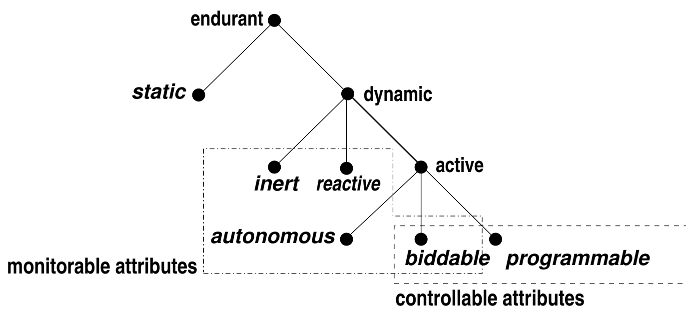

Figure 3: Michael Jackson's Attribute Categories

We elaborate, informally, on the domain analysis attribute predicates, "performed" by the domain analyzer:

(a) **Static.** By a static attribute we shall understand an attribute whose values are constants, i.e., cannot change.

[**is**_static is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *an attribute value* yields the Boolean truth value **true** if the *value* is a static attribute.]

(b) **Dynamic.** By a dynamic attribute we shall understand an attribute whose values are variable, i.e., can change.

[**is**_dynamic is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *an attribute value* yields the Boolean truth value **true** if the *value* is a dynamic attribute.]

(c) **Inert.** By an inert attribute we shall understand a dynamic attribute whose values only change as the result of external stimuli where these stimuli prescribe new values.

[**is**_inert is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *a dynamic attribute value* yields the Boolean truth value **true** if the *value* is an inert attribute.]

(d) **Reactive.**

By a reactive attribute we shall understand a dynamic attribute whose values, if they vary, change in response to external stimuli, where these stimuli come from outside the domain of interest.

[**is**_reactive is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *a dynamic attribute value* yields the Boolean truth value **true** if the is a .]

(e) **Active.** By an active attribute we shall understand a dynamic attribute whose values change (also) of its own volition.

[**is**_active is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *a dynamic attribute value* yields the Boolean truth value **true** if the *value* is an active attribute.]

(f) **Autonomous.** By an autonomous attribute we shall understand a dynamic active attribute whose values change only "on their own volition". The values of an autonomous attributes are a "law onto themselves and their surroundings".

[**is**_autonomous is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *an active attribute value* yields the Boolean truth value **true** if the *value* is an autonomous attribute.]

(g) **Biddable.** By a biddable attribute we shall understand a dynamic active attribute whose values *are prescribed but may fail to be observed as such.*

[**is**_biddable is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *an active attribute value* yields the Boolean truth value **true** if the *value* is a biddable attribute.]

(h) **Programmable.** By a programmable attribute we shall understand a dynamic active attribute whose values can be prescribed.

[**is**_programmable is an informal predicate prompt, i.e., a function which when applied, by the domain analyzer, to a *an active attribute value* yields the Boolean truth value **true** if the *value* is a programmable attribute.]

Figure 3 on the preceding page hints at two major categories of dynamic attributes: *monitorable* and *controllable* attributes.

(a) **Monitorable Attribute.** By a monitorable attribute we shall understand a dynamic active attribute which is either *inert* or *reactive* or *autonomous* or *biddable*. That is:

$\texttt{is\_monitorable}(e) \equiv \texttt{is\_inert}(e) \vee \texttt{is\_reactive}(e) \vee \texttt{is\_autonomous}(e) \vee \texttt{is\_biddable}(e)$.

(a) **Controllable Attribute.**
By a controllable attribute we shall understand a dynamic active attribute which is either *biddable* or *programmable*. That is:

$\texttt{is\_controllable}(e) \equiv \texttt{is\_biddable}(e) \vee \texttt{is\_programmable}(e)$.

51. **Perdurant.** Perdurants are those quantities of domains for which only a fragment exists, in *space*, if we look at or touch them at any given snapshot in *time*.

52. **State.** By a state [of a domain] we shall understand a[ny] set of manifest parts.

53. **Actor.** By an *actor* we shall understand something that is capable of initiating and/or *carrying out* actions, events or behaviours.

    Actors will be described as behaviours. These behaviours evolve around a state. The state is the set of qualities, in particular the dynamic attributes, of the associated parts and/or any possible components or materials of the parts.

54. **Discrete Action.** By a discrete action [22, *Wilson and Shpall*] we shall understand a foreseeable thing which deliberately and potentially changes a well-formed state, in one step, usually into another, still well-formed state, for which an actor can be made responsible.

55. **Discrete Event.** By a *discrete event* we shall understand some unforeseen thing, that is, some 'not-planned-for' "action", one which surreptitiously, non-deterministically changes a well-formed state into another, but usually not a well-formed state, and for which no particular domain actor can be made responsible.

56. **Discrete Behaviour.** By a *discrete behaviour* we shall understand a set of sequences of potentially interacting sets of discrete actions, discrete events and discrete behaviours.

57. **Channels:** Behaviours sometimes synchronise and usually communicate. Synchronisation and communication is abstracted as the sending (ch ! m) and receipt (ch ?) of messages, m:M, over channels, ch. Channels are abstractions. They are abstractions of the 'medium' in which synchronizaion and communication takes place.

58. **Domain Description.** A domain description consists of a set of one or more description units. There are six kinds of description units.

    (a) **Type Description Unit.** A type description unit specifies one or more types.

    (b) **Value Description Unit.** A value description unit specifies one or more values of specified types.

    (c) **Function Description Unit.** A function description unit [is a value description unit and] specifies one or more functions: their signatures and their corresponding [body] definition. [Behaviours and actions are described by such units.]

    (d) **Variable Description Unit.** A variable description unit declares one or more variables of specified types.

    (e) **Axiom Description Unit.** An axiom description unit specifies properties of types and values (incl. funtions).

    (f) **Channel Description Unit.** A channel description unit declares a channel [array].

59. **Domain Initialization.** Domain initialization specifies the initial argument values for all part behaviours and "starts" their behaviour.

60. **Domain Engineering.** Domain engineering is the engineering of studying, analyzing and decribing domains.

61. **Domain Science.** Domain science is the scietific, i.., mathematical study of how to describe domains and of the general properties of domains.[22].

62. **Machine.** By machine we shall understand a, or the, combination of hardware and software that is the target for, or result of the required computing systems development.

63. **Requirements.** We present three complementary characterizations:

    (a) By a *requirements* we understand (cf., [**?**, IEEE Standard610.12]): *"A condition or capability needed by a user to solve a problem or achieve an objective"*.

[22]Typical studies could be studies of how to describe time-continous, i.e., non-discrete behaviours and studies of *intentional pulls*

(b) By *requirements* we shall understand a document which prescribes desired properties of a machine: what endurants the machine shall "maintain", and what the machine shall (must; not should) offer of functions and of behaviours while also expressing which events the machine shall "handle".

(c) By *requirements* we shll mean: to specify the/a machine.

Domain Requirements can be analyzed and prescribed in three stages: domain, interface and machine requirements.

(a) **Domain Requirements.** Domain requirements are those requirements which can be expressed withou any reference to the machine.

Domain requirements can be prescribed in a number of stages.

i. **Domain Projection.** By a domain projection is meant *a subset of the domain description, one which* **projects out** *all those endurants: parts, materials and components, as well as perdurants: actions, events and behaviours that the stakeholders do not wish represented or relied upon by the machine.*

ii. **Domain Instantiation.** By domaininstantiation we mean *a* **refinement** *of the partial domain requirements prescription (resulting from the projection step) in which the refinements aim at rendering the endurants: parts, materials and components, as well as the perdurants: actions, events and behaviours of the domain requirements prescription more concrete, more specific.*

iii. **Domain Determination.** By *domain* determination we mean *a* **refinement** *of the partial domain requirements prescription, resulting from the instantiation step, in which the refinements aim at rendering the endurants: parts, materials and components, as well as the perdurants: functions, events and behaviours of the partial domain requirements prescription less non-determinate, more determinate.*

iv. **Domain Extension.** By *domain* extension we understand *the introduction of endurants and perdurants that were not feasible in the original domain, but for which, with computing and communication, and with new, emerging technologies, for example, sensors, actuators and satellites, there is the possibility of feasible implementations, hence the requirements, that what is introduced becomes part of the unfolding requirements prescription.*

(b) **Interface Requirements.** Interface requirements are those requirements which can be expressed with reference to bot the domain and the machine.

(c) **Machine Requirements.** Machine requirements are those requirements which can be expressed without any reference to the domain, solely to the machine.

Concepts such as *shared* and *derived requirements* and *requirements fitting* are also relevant, but not defined here; cf. [6, *Chapter 9*].

64. **Software Design.**

65. **Validation.**

(a) **Verification.**

(b) **Testing.**

(c) **Model Checking.**

The are 102 Software Engineering terms defined in this section.

B Indexes

B.1 Financial Management Terminology

I expect there to be several more terms to be defined, hence to be indexed.

B.2 Software Engineering Terminology

There are some 100 terms indexed here.

B.3 Domain Description Formula

Entry ιs refer to Items. Some functions are so-called *overloaded*, i.e., same function name for different signatures: different in definition set types, but same in range set type.

B.4 "Statistics"

There are

- 29 financial management terminology (Sect. 2),
- 194 software engineering terminology (Appendix A),

  and

- 189 formal, double-entry bookkeeping description (Sects. 3-5):
  - 22 axiom,
  - 8 behaviour,
  - 1 channel,
  - 45 function,
  - 98 type, and
  - 15 value

index entries – for a current total of 412 index entries.

I expect to add several more financial management terms. A few more software engineering terms. And more domain description formula entries as I finalize and correct my domain description.