To Every Manifest Domain a CSP Expression
A Rôle for Mereology in Computer Science

Dines Bjørner: December 9, 2016, 15:37

Fredsvej 11, DK-2840 Holte, Denmark.
DTU Compute, Technical University of Denmark, DK-2800 Kgs. Lyngby, Denmark.
E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~dibj

Abstract

We give an abstract model[1] of parts and part-hood relations, of Stansław Leśniewski's *mereology* [2]. Mereology applies to software application domains such as the financial service industry, railway systems, road transport systems, health care, oil pipelines, secure [IT] systems, etcetera. We relate this model to axiom systems for mereology, showing satisfiability, and show that for every mereology there corresponds a class of `Communicating Sequential Processes` [3], that is: a λ–expression.

Keywords: Mereology, Manifest Domain, Domain Description

In memory of Douglas T. Ross 1929–2007

1. Introduction

The term 'mereology' is accredited to the Polish mathematician, philosopher and logician Stansław Leśniewski (1886–1939). In this contribution we shall be concerned with only certain aspects of mereology, namely those that appears most immediately relevant to domain science (a relatively new part of current computer science). Our knowledge of 'mereology' has been through studying, amongst others, [2].

1.1. Computing Science Mereology

"Mereology (from the Greek $\mu\varepsilon\rho o\varsigma$ 'part') is the theory of parthood relations: of the relations of part to whole and the relations of part to part within a whole"[2].

[1] This paper is a complete rewrite of [1]. Sections 3 and 6 are thus completely rewritten.
[2] Achille Varzi: Mereology, http://plato.stanford.edu/entries/mereology/ 2009 and [2]

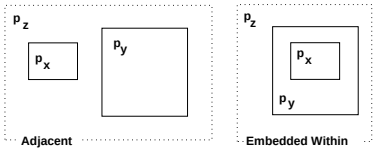

Figure 1: Immediately 'Adjacent' and 'Embedded Within' Parts

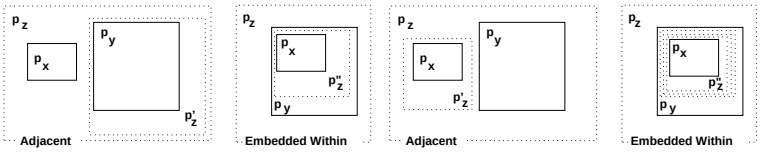

Figure 2: Transitively 'Adjacent' and 'Embedded Within' Parts

In this contribution we restrict 'parts' to be those that, firstly, are spatially distinguishable, then, secondly, while "being based" on such spatially distinguishable parts, are conceptually related. We use the term 'part' in a more general sense than in [4]. The relation: "being based", shall be made clear in this paper. Accordingly two parts, p_x and p_y, (of a same "whole") are are either "adjacent", or are "embedded within", one within the other, as loosely indicated in Fig. 1. 'Adjacent' parts are direct parts of a same third part, p_z, i.e., p_x and p_y are "embedded within" p_z; or one (p_x) or the other (p_y) or both (p_x and p_y) are parts of a same third part, p'_z "embedded within" p_z; etcetera; as loosely indicated in Fig. 2, or one is "embedded within" the other — etc. as loosely indicated in Fig. 2. Parts, whether 'adjacent' or 'embedded within', can share properties. For adjacent parts this sharing seems, in the literature, to be diagrammatically expressed by letting the part rectangles "intersect". Usually properties are not spatial hence 'intersection' seems confusing. We refer to Fig. 3. Instead of depicting parts sharing properties as in Fig. 3[L]eft, where shaded, dashed rounded-edge rectangles stands for 'sharing', we shall (eventually) show parts sharing properties as in Fig. 3[R]ight where •—• connections connect those parts.

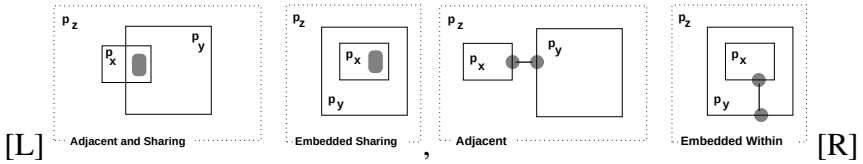

Figure 3: Two models, [L,R], of parts sharing properties

1.2. From Domains via Requirements to Software

One reason for our interest in mereology is that we find that concept relevant to the modeling of domains. A derived reason is that we find the modeling of domains relevant to the development of software. Conventionally a first phase of software development is that of requirements engineering. To us domain engineering is (also) a prerequisite for requirements engineering [5]. Thus to properly **design** Software we need to **understand** its or their Requirements; and to properly **prescribe** Requirements one must **understand** its Domain. To **argue** correctness of Software with respect to Requirements one must usually **make assumptions** about the Domain: $\mathbb{D}, \mathbb{S} \models \mathbb{R}$. Thus **description** of Domains become an indispensable part of Software development.

1.3. Domains: Science and Engineering

Domain Science is the study and knowledge of domains. **Domain Engineering** is the practice of **"walking the bridge"** from domain science to domain descriptions: to **create domain descriptions** on the background of scientific knowledge of domains, the specific domain "at hand", or domains in general; and to **study domain descriptions** with a view to broaden and deepen scientific results about domain descriptions. This contribution is based on the engineering and study of many descriptions, of air traffic, banking, commerce (the consumer/retailer/wholesaler/producer supply chain), container lines, health care, logistics, pipelines, railway systems, secure [IT] systems, stock exchanges, etcetera.

1.4. Contributions of This Paper

A general contribution of this paper is that of providing elements of a domain science. Three specific contributions are those of (i) giving a model that satisfies published formal, axiomatic characterisations of mereology; (ii) showing that to every (such modeled) mereology there corresponds a CSP [3] program; and (iii) suggesting complementing **syntactic** and **semantic** theories of mereology.

1.5. Structure of This Paper

We briefly overview the structure of this contribution. First, in Sect. 2, **we loosely characterise how we look at mereologies: "what they are to us !"**. Then, in Sect. 3, **we give an abstract, model-oriented specification of a class of mereologies** in the form of composite parts and composite and atomic subparts and their possible connections. The abstract model as well as the axiom system (Sect. 5) focuses on the **syntax of mereologies.** Following that, in Sect. 5, **we indicate how the model** of Sect. 3 **satisfies the axiom system of that section.** In preparation for Sect. 6 we **present characterisations of attributes of parts, whether atomic or composite.** Finally Sect. 6 presents **a**

semantic model of mereologies, one of a wide variety of such possible models. This one emphasizes the possibility of considering parts and subparts as processes and hence a mereology as a system of processes. Section 7 concludes with some remarks on what we have achieved.

2. Our Concept of Mereology

2.1. Informal Characterisation

Mereology, to us, is the study and knowledge about how physical and conceptual parts relate and what it means for a part to be related to another part: *being disjoint, being adjacent, being neighbours, being contained properly within, being properly overlapped with,* etcetera. By physical parts we mean such spatial individuals which can be pointed to. **Examples:** *a road net (consisting of street segments and street intersections); a street segment (between two intersections); a street intersection; a road (of sequentially neigbouring street segments of the same name); a vehicle; and a platoon (of sequentially neigbouring vehicles).* By a conceptual part we mean an abstraction with no physical extent, which is either present or not. **Examples:** *a bus timetable (not as a piece or booklet of paper, or as an electronic device, but) as an image in the minds of potential bus passengers; and routes of a pipeline, that is, neighbouring sequences of pipes, valves, pumps, forks and joins, for example referred to in discourse: "the gas flows through "such-and-such" a route".* The tricky thing here is that a route may be thought of as being both a concept or being a physical part — in which case one ought give them different names: a planned route and an actual road, for example. The mereological notion of subpart, that is: *contained within* can be illustrated by **examples:** *the intersections and street segments are subparts of the road net; vehicles are subparts of a platoon; and pipes, valves, pumps, forks and joins are subparts of pipelines.* The mereological notion of adjacency can be illustrated by **examples.** We consider *the various controls of an air traffic system, cf. Fig. 4 on the following page, as well as its aircraft, as adjacent within the air traffic system; the pipes, valves, forks, joins and pumps of a pipeline, cf. Fig. 9 on Page 9, as adjacent within the pipeline system; two or more banks of a banking system, cf. Fig. 6 on Page 7, as being adjacent.* The mereo-topological notion of neighbouring can be illustrated by **examples:** *Some adjacent pipes of a pipeline are neighbouring (connected) to other pipes or valves or pumps or forks or joins, etcetera; two immediately adjacent vehicles of a platoon are neighbouring.* The mereological notion of proper overlap can be illustrated by **examples** some of which are of a general kind: *two routes of a pipelines may overlap; and two conceptual bus timetables may overlap with some, but not all bus line entries being the same;* and some really reflect adjacency: *two adjacent pipe overlap in their connection, a wall between two rooms overlap each of these rooms — that is, the rooms overlap each other "in the wall".*

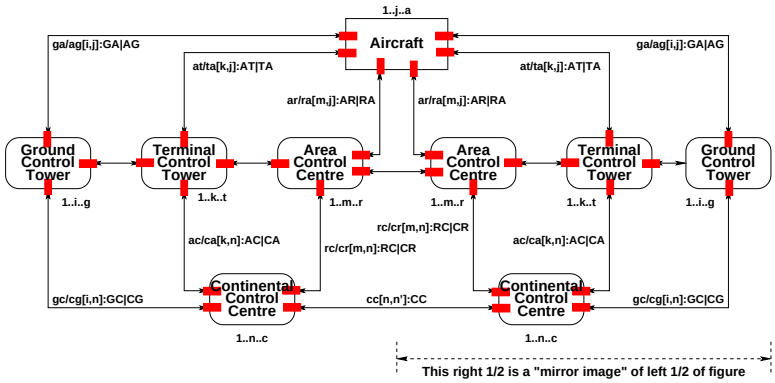

Figure 4: A schematic air traffic system

2.2. Six Examples

We shall, in Sect. 3, present a model that is claimed to abstract essential mereological properties of air traffic, buildings and their installations, machine assemblies, financial service industry, the oil industry and oil pipelines, and railway nets.

2.2.1. Air Traffic

Figure 4 shows nine adjacent (9) boxes and eighteen adjacent (18) lines. Boxes and lines are parts. The line parts "neighbours" the box parts they "connect". Individually boxes and lines represent adjacent parts of the composite air traffic "whole". The rounded corner boxes denote buildings. The sharp corner box denote aircraft. Lines denote radio telecommunication. The "overlap" between neigbouring line and box parts are indicated by "connectors". Connectors are shown as small filled, narrow, either horisontal or vertical "filled" rectangle[3] at both ends of the double-headed-arrows lines, overlapping both the line arrows and the boxes. The index ranges shown attached to, i.e., labeling each unit, shall indicate that there are a multiple of the "single" (thus representative) box or line unit shown. These index annotations are what makes the diagram of Fig. 4 schematic. Notice that the 'box' parts are fixed installations and that the double-headed arrows designate the ether where radio waves may propagate. We could, for example, assume that each such line is characterised by a combination of location and (possibly encrypted) radio communication frequency. That would allow us to consider all lines for not overlapping. And if they were overlapping, then that must have been a decision of the air traffic system.

[3]There are 38 such rectangles in Fig. 4.

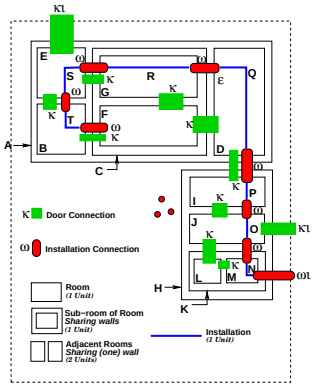

Figure 5: A building plan with installation

2.2.2. Buildings

Figure 5 shows a building plan — as a composite part. The building consists of two buildings, A and H. The buildings A and H are neighbours, i.e., shares a common wall. Building A has rooms B, C, D and E, Building H has roomsl, J and K; Rooms L and M are within K. Rooms F and G are within C. The thick lines labeled N, O, P, Q, R, S, and T models either electric cabling, water supply, air conditioning, or some such "flow" of gases or liquids. Connection $\kappa\iota o$ provides means of a connection between an environment, shown by dashed lines, and B or J, i.e. "models", for example, a door. Connections κ provides "access" between neighbouring rooms. Note that 'neighbouring' is a transitive relation. Connection $\omega\iota o$ allows electricity (or water, or oil) to be conducted between an environment and a room. Connection ω allows electricity (or water, or oil) to be conducted through a wall. Etcetera. Thus "the whole" consists of A and B. Immediate subparts of A are B, C, D and E. Immediate subparts of C are G and F. Etcetera.

2.2.3. Financial Service Industry

Figure 6 on the next page is rather rough-sketchy! It shows seven (7) larger boxes [6 of which are shown by dashed lines], six [6] thin lined "distribution" boxes, and twelve (12) double-arrowed lines. Boxes and lines are parts. (We do not described what is meant by "distribution".) Where double-arrowed lines touch upon (dashed) boxes we have connections. Six (6) of the boxes, the dashed line boxes, are composite parts, five (5) of them consisting of a variable number of atomic parts; five (5) are here shown as having three atomic parts each with bullets "between" them to designate "variability". Clients, not shown, access the outermost (and hence the "innermost" boxes, but the latter is not shown) through connections, shown by bullets, •.

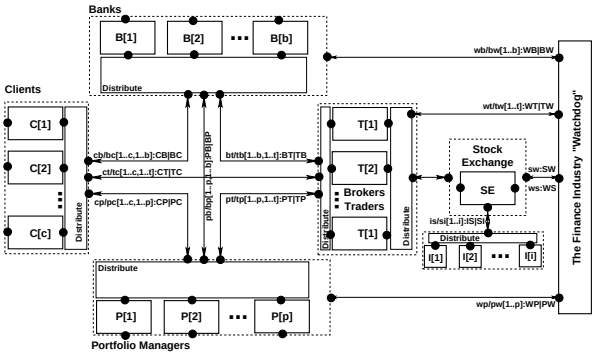

Figure 6: A Financial Service Industry

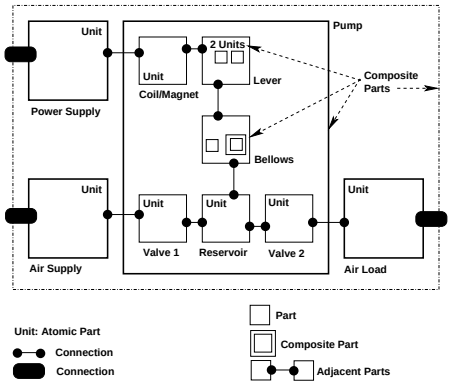

Figure 7: An air pump, i.e., a physical mechanical system

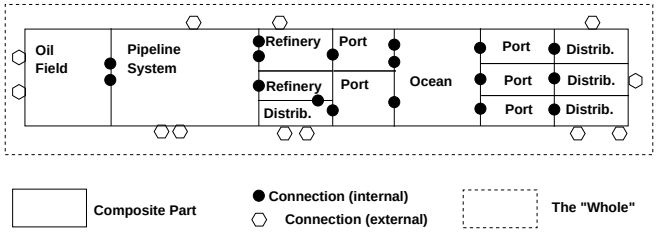

Figure 8: A Schematic of an Oil Industry

2.2.4. Machine Assemblies

Figure 7 on the preceding page shows a machine assembly. Square boxes designate either composite or atomic parts. Black circles or ovals show connections. The full, i.e., the level 0, composite part consists of four immediate parts and three internal and three external connections. The Pump is an assembly of six (6) immediate parts, five (5) internal connections and three (3) external connectors. Etcetera. Some connections afford "transmission" of electrical power. Other connections convey torque. Two connections convey input air, respectively output air.

2.2.5. Oil Industry

"The" Overall Assembly. Figure 8 shows a composite part consisting of fourteen (14) composite parts, left-to-right: one oil field, a crude oil pipeline system, two refineries and one, say, gasoline distribution network, two seaports, an ocean (with oil and ethanol tankers and their sea lanes), three (more) seaports, and three, say gasoline and ethanol distribution networks. Between all of the neighbouring composite parts there are connections, and from some of these composite parts there are connections (to an external environment). The crude oil pipeline system composite part will be concretised next.

A Concretised Composite Pipeline. Figure 9 on the following page shows a pipeline system. It consists of 32 atomic parts: fifteen (15) pipe units (shown as directed arrows and labeled p1–p15), four (4) input node units (shown as small circles, ○, and labeled in$i$–inℓ), four (4) flow pump units (shown as small circles, ○, and labeled fp$a$–fp$d$), five (5) valve units (shown as small circles, ○, and labeled vx–vw), three (3) join units (shown as small circles, ○, and labeled jb–jc), two (2) fork units (shown as small circles, ○, and labeled fb–fc), one (1) combined join & fork unit (shown as small circles, ○, and labeled jafa), and four (4) output node units (shown as small circles, ○, and labeled on$p$–on$s$). In this example the routes through the pipeline system start with node units and end with node units, alternates between node units and pipe units, and are connected as shown

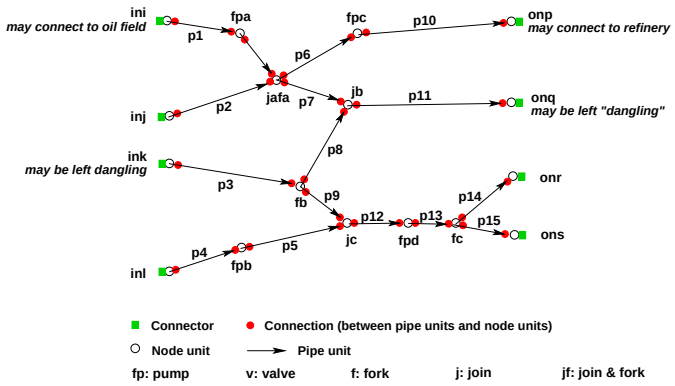

Figure 9: A Pipeline System

by fully filled-out dark coloured disc connections. Input and output nodes have input, respectively output connections, one each, and shown as lighter coloured connections. In [6] we present a description of a class of abstracted pipeline systems.

2.2.6. Railway Nets

The left of Fig. 10 on the next page [L] diagrams four rail units, each with two, three or four connectors shown as narrow, somewhat "longish" rectangles. Multiple instances of these rail units can be assembled (i.e., composed) by their connectors as shown on Fig. 10 on the following page [L] into proper rail nets. The right of Fig. 10 on the next page [R] diagrams an example of a proper rail net. It is assembled from the kind of units shown in Fig. 10 [L]. In Fig. 10 [R] consider just the four dashed boxes: The dashed boxes are assembly units. Two designate stations, two designate lines (tracks) between stations. We refer to to the caption four line text of Fig. 10 on the following page for more "statistics". We could have chosen to show, instead, for each of the four "dangling' connectors, a composition of a connection, a special "end block" rail unit and a connector.

2.2.7. Discussion

We have brought these examples only to indicate the issues of a "whole" and atomic and composite parts, adjacency, within, neighbour and overlap relations, and the ideas of attributes and connections. We shall make the notion of 'connection' more precise in the next section.

3. An Abstract, Syntactic Model of Mereologies

3.1. Parts and Subparts

1 We distinguish between **atomic** and **composite parts.**

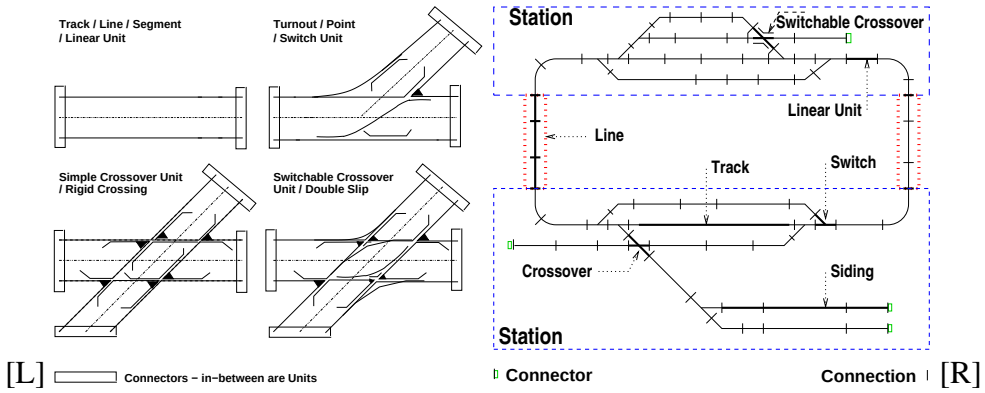

Figure 10: To the left: Four rail units.

To the right: A "model" railway net. An Assembly of four Assemblies: two stations and two lines.

Lines here consist of linear rail units; stations of all the kinds of units shown in to the left. There are 66 connections and four "dangling" connectors

2 Atomic parts do not contain separately distinguishable parts.

3 Composite parts contain at least one separately distinguishable part.

type

1. P == AP | CP
2. AP :: mkA(...)
3. CP :: mkC(...,s_sps:P-**set**) **axiom** ∀ mkC(_,ps):CP • ps≠{}

It is the domain analyser who decides what constitutes "the whole", that is, how parts relate to one another, what constitutes parts, and whether a part is atomic or composite. We refer to the proper parts of a composite part as subparts. Figure 11 on the next page illustrates composite and atomic parts. The *slanted sans serif* uppercase identifiers of Fig. 11 *A1, A2, A3, A4, A5, A6* and *C1, C2, C3* are meta-linguistic, that is. they stand for the parts they "decorate"; they are not identifiers of "our system".

3.2. No "Infinitely" Embedded Parts

The above syntax, Items 1–3, does not prevent composite parts, p, to contain composite parts, p', "ad-infinitum ! But we do not wish such "recursively" contained parts !

4 To express the property that parts are finite we introduce a notion of part derivation.

5 The part derivation of an atomic part is the empty set.

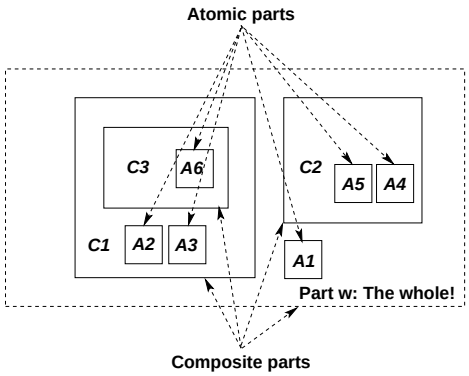

Figure 11: Atomic and Composite Parts

6 The part derivation of a composite part, p, mkC(pq,ps) where pq is that composite part's quality, is the set ps of subparts of p.

value
4. pt_der: P → P-**set**
5. pt_der(mkA(pq)) ≡ {}
6. pt_der(mkC(pq,ps)) ≡ ps

7 We can also express the part derivation, pt_der(ps) of a set, ps, of parts.
8 If the set is is empty then pt_der({}) is the empty set, {}.
9 Let mkA(pq) be an element of ps, then pt_der({mkA(pq)}∪ps′) is ps′.
10 Let mkC(pq,ps′) be an element of ps, then pt_der(ps′∪ps) is ps′.

7. pt_der: P-**set** → P-**set**
8. pt_der({}) ≡ {}
9. pt_der({mkA(pq)}∪ps) ≡ ps
10. pt_der({mkC(pq,ps')}∪ps) ≡ ps'∪ps

11 Therefore, to express that a part is finite we postulate
12 a natural number, *n*, such that a notion of iterated part set derivations lead to an empty set.
13 An iterated part set derivation takes a set of parts and part set derive that set repeatedly, *n* times.
14 If the result is an empty set, then part *p* was finite.

value
11. no_infinite_parts: P → **Bool**
12. no_infinite_parts(p) ≡
12. ∃ n:**Nat** • it_pt_der({p})(n)={}
13. it_pt_der: P**-set** → **Nat** → P**-set**
14. it_pt_der(ps)(n) ≡
14. **let** ps′ = pt_der(ps) **in**
14. **if** n=1 **then** ps′ **else** it_pt_der(ps′)(n−1) **end end**

3.3. Unique Identifications

Each physical part can be uniquely distinguished for example by an abstraction of its properties at a time of origin. In consequence we also endow conceptual parts with unique identifications.

15 In order to refer to specific parts we endow all parts, whether atomic or composite, with **u**nique **id**entifications.
16 We postulate functions which observe these **u**nique **id**entifications, whether as parts in general or as atomic or composite parts in particular.
17 such that any to parts which are distinct have **u**nique **id**entifications.

type
15. UI
value
16. uid_UI: P → UI
axiom
17. ∀ p,p′:P • p≠p′ ⇒ uid_UI(p)≠uid_UI(p′)

A model for uid_UI can be given. Presupposing subsequent material (on attributes and mereology) — "lumped" into part qualities, pq:PQ, we augment definitions of atomic and composite parts:

type
2. AP :: mkA(s_pq:(s_uid:UI,...))
3. CP :: mkC(s_pq:(s_uid:UI,...),s_sps:P**-set**)
value
16. uid_UI(mkA((ui,...))) ≡ ui
16. uid_UI(mkC((ui,...)),...) ≡ ui

Figure 12 illustrates the unique identifications of composite and atomic parts. No two parts have the same unique identifier.

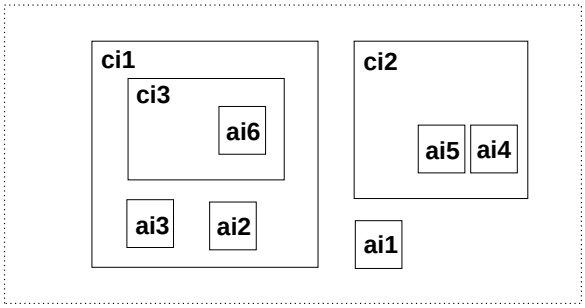

Figure 12: ai_j: atomic part identifiers, ci_k: composite part identifiers

18 We define an auxiliary function, no_prts_uis, which applies to a[ny] part, p, and yields a pair: the number of subparts of the part argument, and the set of unique identifiers of parts within p.

19 no_prts_uis is defined in terms of yet an auxiliary function, sum_no_pts_uis.

```
value
18.  no_prts_uis: P → (Nat × UI-set) → (Nat × UI-set)
18.  no_pts_uis(mkA(ui,...))(n,uis) ≡ (n+1,uis∪{ui})
18.  no_pts_uis(mkC((ui,...),ps))(n,uis) ≡
18.      let (n′,uis′) = sum_no_pts_uis(ps) in
18.      (n+n′,uis∪uis') end
18.  pre: no_infinite_parts(p)
19.  sum_no_pts_uis: P-set → (Nat × UI-set) → (Nat × UI-set)
19.  sum_no_pts_uis(ps)(n,uis) ≡
19.    case ps of
19.      {}→(n,uis),
19.      {mkA(ui,...)}∪ps'→sum_no_pts_uis(ps′)(n+1,uis∪{ui}),
19.      {mkC((ui,...),ps′)}∪ps" →
19.        let (n″,uis″)=sum_no_pts_uis(ps′)(1,{ui}) in
19.        sum_no_pts_uis(ps″)(n+n″,uis∪uis") end
19.    end
19.    pre: ∀ p:P•p ∈ ps ⇒ no_infinite_parts(p)
```

20 That no two parts have the same unique identifier can now be expressed by demanding that the number of parts equals the number of unique identifiers.

```
axiom
20.  ∀ p:P • let (n,uis)=no_prts_uis(0,{}) in n=card uis end
```

3.4. Attributes

3.4.1. Attribute Names and Values

21 Parts have sets of named attribute values, attrs:ATTRS.

22 One can observe attributes from parts.

23 Two distinct parts may share attributes:

   a For some (one or more) attribute name that is among the attribute names of both parts,

   b it is always the case that the corresponding attribute values are identical.

type

21. ANm, AVAL, ATTRS = ANm $\xrightarrow{m}$ AVAL

value

22. attr_ATTRS: P → ATTRS
23. share: P×P → **Bool**
23. share(p,p′) ≡
23. p≠p′ ∧ ∼trans_adj(p,p′) ∧
23a. ∃ anm:ANm • anm ∈ **dom** attr_ATTRS(p) ∩ **dom** attr_ATTRS(p′) ⇒
23b. □ (attr_ATTRS(p))(anm) = (attr_ATTRS(p′))(anm)

The function trans_adj is defined in Sect. 4.4 on Page 18.

3.4.2. Attribute Categories

One can suggest a hierarchy of part attribute categories: static or dynamic values — and within the dynamic value category: inert values or reactive values or active values — and within the dynamic active value category: autonomous values or biddable values or programmable values. By a **static attribute**, a:A, `is_static_attribute`(a), we shall understand an attribute whose values are constants, i.e., cannot change. By a **dynamic attribute**, a:A, `is_dynamic_attribute`(a), we shall understand an attribute whose values are variable, i.e., can change. By an **inert attribute**, a:A, `is_inert_attribute`(a), we shall understand a dynamic attribute whose values only change as the result of external stimuli where these stimuli prescribe properties of these new values. By a **reactive attribute**, a:A, `is_reactive_attribute`(a), we shall understand a dynamic attribute whose values, if they vary, change value in response to the change of other attribute values. By an **active attribute**, a:A, `is_active_attribute`(a), we shall understand a dynamic attribute whose values change (also) of its own volition. By an **autonomous attribute**, a:A, `is_autonomous_attribute`(a), we shall understand a dynamic active attribute whose values change value only "on

their own volition". The values of an autonomous attributes are a "law onto themselves and their surroundings". By a **biddable attribute**, a:A, `is_biddable_attribute(a)`, (of a part) we shall understand a dynamic active attribute whose values are prescribed but may fail to be observed as such. By a **programmable attribute**, a:A, `is_programmable_attribute(a:A)`, we shall understand a dynamic active attribute whose values can be prescribed. By an **external attribute** we mean inert, reactive, active or autonomous attribute. By a **controllable attribute** we mean a biddable or programmable attribute. We define some auxiliary functions:

24 $\mathscr{S}_{\mathscr{A}}$ applies to attrs:ATTRS and yields a grouping $(\mathsf{sa}_1,\mathsf{sa}_2,...,\mathsf{sa}_{n_s})$[4], of **static** attribute values.
25 $\mathscr{C}_{\mathscr{A}}$ applies to attrs:ATTRS and yields a grouping $(\mathsf{ca}_1,\mathsf{ca}_2,...,\mathsf{ca}_{n_c})$[5] of **controllable** attribute values.
26 $\mathscr{E}_{\mathscr{A}}$ applies to attrs:ATTRS and yields a set, $\{\mathsf{eA}_1,\mathsf{eA}_2,...,\mathsf{eA}_{n_e}\}$[6] of **external** attribute names.

type
SA,CA = AVAL*
EA = ANm−st
value
24. $\mathscr{S}_{\mathscr{A}}$: ATTRS → SA
25. $\mathscr{C}_{\mathscr{A}}$: ATTRS → CA
26. $\mathscr{E}_{\mathscr{A}}$: ATTRS → EA

The attribute names of static, controllable and external attributes do not overlap and together make up the attribute names of attrs.

3.5. Mereology

In order to illustrate other than the within and adjacency part relations we introduce the notion of mereology. Figure 13 on the next page illustrates a mereology between parts. A specific mereology-relation is, visually, a •—• line that connects two distinct parts.

27 The mereology of a part is a set of unique identifiers of other parts.

type
27. ME = UI-**set**

We may refer to the connectors by the two element sets of the unique identifiers of the parts they connect. For **example** with respect to Fig. 13 on the following page:

[4]– where $\{\mathsf{sa}_1,\mathsf{sa}_2,...,\mathsf{sa}_{n_s}\}\subseteq$**rng** attrs
[5]– where $\{\mathsf{ca}_1,\mathsf{ca}_2,...,\mathsf{ca}_{n_s}\}\subseteq$**rng** attrs
[6]– where $\{\mathsf{eA}_1,\mathsf{eA}_2,...,\mathsf{eA}_{n_e}\}\subseteq$**dom** attrs

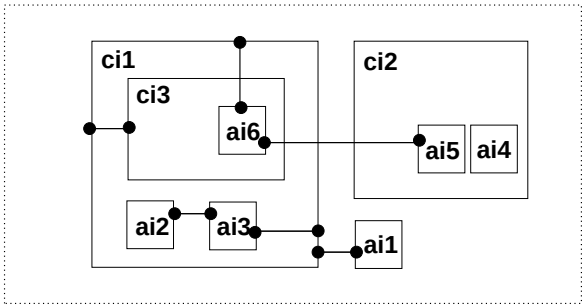

Figure 13: Mereology: Relations between Parts

- $\{ci_1, ci_3\}$,
- $\{ai_6, ci_1\}$,
- $\{ai_6, ai_5\}$ and
- $\{ai_2, ai_3\}$,
- $\{ai_3, ci_1\}$,
- $\{ai_1, ci_1\}$.

3.6. The Model

28 The "whole" is a part.

29 A part value has a part sort name and is either the value of an atomic part or of an abstract composite part.

30 A atomic part value has a part quality value.

31 An abstract composite part value has a part quality value and a set of at least of one or more part values.

32 A part quality value consists of a unique identifier, a mereology, and a set of one or more attribute named attribute values.

```
28  W  =  P
29  P  =  AP | CP
30  AP ::  mkA(s_pq:PQ)
31  CP ::  mkC(s_pq:PQ,s_ps:P-set)
32  PQ =  UI×ME×(ANm →m AVAL)
```

We now assume that parts are not "recursively infinite", and that all parts have unique identifiers

4. Some Part Relations

4.1. 'Immediately Within'

33 One part, p, is said to be *immediately within*, imm_within(p,p′), another part, if p′ is a composite part and p is observable in p′.

```
value
33.  imm_within: P × P → Bool
33.  imm_within(p,p′) ≡
33.      case p′ of
33.          (_,mkA(_,ps)) → p ∈ ps,
33.          (_,mkC(_,ps)) → p ∈ ps,
33.          _ → false
33.      end
```

4.2. 'Transitive Within'

We can generalise the 'immediate within' property.

34 A part, p, is transitively within a part p′, trans_within(p,p′),
   a either if p, is immediately within p′
   b or
   c if there exists a (proper) composite part p″ of p′ such that trans_within(p″,p).

```
value
34.  trans_wihin: P × P → Bool
34.  trans_within(p,p′) ≡
34a.     imm_within(p,p′)
34b.     ∨
34c.     case p′ of
34c.         (_,mkC(_,ps)) → p ∈ ps ∧
34c.             ∃ p″:P• p″ ∈ ps ∧ trans_within(p″,p),
34c.         _ → false
34.      end
```

4.3. 'Adjacency'

35 Two parts, p,p′, are said to be *immediately adj*acent, imm_adj(p,p′)(c), to one another, in a composite part c, such that p and p′ are distinct and observable in c.

```
value
35.  imm_adj: P × P → P → Bool
35.  imm_adj(p,p′)(mkA(_,ps)) ≡ p≠p′ ∧ {p,p′}⊆ps
35.  imm_adj(p,p′)(mkC(_,ps)) ≡ p≠p′ ∧ {p,p′}⊆ps
35.  imm_adj(p,p′)(mkA(_)) ≡ false
```

4.4. Transitive 'Adjacency'

We can generalise the immediate 'adjacent' property.

36 Two parts, p′,p″, of a composite part, p, are trans_adj(p′, p″) in p
  a either if imm_adj(p′,p″)(p),
  b or if there are two p‴ and p⁗ such that
    i p‴ and p⁗ are immediately adjacent parts of p and
    ii p is equal to p‴ or p‴ is properly within p and p′ is equal to p⁗ or p⁗ is properly within p′

We leave the formalisation to the reader.

5. An Axiom System

Classical axiom systems for mereology focus on just one sort of "things", namely $\mathscr{P}$arts. Leśniewski had in mind, when setting up his mereology to have it supplant set theory. So parts could be composite and consisting of other, the sub-parts — some of which would be atomic; just as sets could consist of elements which were sets — some of which would be empty.

5.1. Parts and Attributes

In our axiom system for mereology we shall avail ourselves of two sorts: $\mathscr{P}$arts, and $\mathscr{A}$ttributes.[7]

- **type** $\mathscr{P}, \mathscr{A}$

$\mathscr{A}$ttributes are associated with $\mathscr{P}$arts. We do not say very much about attributes: We think of attributes of parts to form possibly empty sets. So we postulate a primitive predicate, $\in$, relating $\mathscr{P}$arts and $\mathscr{A}$ttributes.

- $\in: \mathscr{A} \times \mathscr{P} \to$ **Bool**.

5.2. The Axioms

The axiom system to be developed in this section is a variant of that in [2]. We introduce the following relations between parts:

[7] Identifiers P and A stand for model-oriented types (parts and atomic parts), whereas identifiers $\mathscr{P}$ and $\mathscr{A}$ stand for property-oriented types (parts and attributes).

part_of:	$\mathbb{P}$:	$\mathscr{P} \times \mathscr{P}$	$\rightarrow$	**Bool**	Page 19
proper_part_of:	$\mathbb{PP}$:	$\mathscr{P} \times \mathscr{P}$	$\rightarrow$	**Bool**	Page 19
overlap:	$\mathbb{O}$:	$\mathscr{P} \times \mathscr{P}$	$\rightarrow$	**Bool**	Page 19
underlap:	$\mathbb{U}$:	$\mathscr{P} \times \mathscr{P}$	$\rightarrow$	**Bool**	Page 19
over_crossing:	$\mathbb{OX}$:	$\mathscr{P} \times \mathscr{P}$	$\rightarrow$	**Bool**	Page 19
under_crossing:	$\mathbb{UX}$:	$\mathscr{P} \times \mathscr{P}$	$\rightarrow$	**Bool**	Page 20
proper_overlap:	$\mathbb{PO}$:	$\mathscr{P} \times \mathscr{P}$	$\rightarrow$	**Bool**	Page 20
proper_underlap:	$\mathbb{PU}$:	$\mathscr{P} \times \mathscr{P}$	$\rightarrow$	**Bool**	Page 20

Let $\mathbb{P}$ denote **part-hood**; p_x is part of p_y, is then expressed as $\mathbb{P}(p_x, p_y)$.[8] (1) Part p_x is part of itself (reflexivity). (2) If a part p_x is part p_y and, vice versa, part p_y is part of p_x, then $p_x = p_y$ (anti-symmetry). (3) If a part p_x is part of p_y and part p_y is part of p_z, then p_x is part of p_z (transitivity).

$$\forall p_x : \mathscr{P} \bullet \mathbb{P}(p_x, p_x) \tag{1}$$

$$\forall p_x, p_y : \mathscr{P} \bullet (\mathbb{P}(p_x, p_y) \wedge \mathbb{P}(p_y, p_x)) \Rightarrow p_x = p_y \tag{2}$$

$$\forall p_x, p_y, p_z : \mathscr{P} \bullet (\mathbb{P}(p_x, p_y) \wedge \mathbb{P}(p_y, p_z)) \Rightarrow \mathbb{P}(p_z, p_z) \tag{3}$$

Let $\mathbb{PP}$ denote **proper part-hood**. p_x is a proper part of p_y is then expressed as $\mathbb{PP}(p_x, p_y)$. $\mathbb{PP}$ can be defined in terms of $\mathbb{P}$. $\mathbb{PP}(p_x, p_y)$ holds if p_x is part of p_y, but p_y is not part of p_x.

$$\mathbb{PP}(p_x, p_y) \triangleq \mathbb{P}(p_x, p_y) \wedge \neg \mathbb{P}(p_y, p_x) \tag{4}$$

Overlap, $\mathbb{O}$, expresses a relation between parts. Two parts are said to overlap if they have "something" in common. In classical mereology that 'something' is parts. To us parts are spatial entities and these cannot "overlap". Instead they can 'share' attributes.

$$\mathbb{O}(p_x, p_y) \triangleq \exists a : \mathscr{A} \bullet a \in p_x \wedge a \in p_y \tag{5}$$

Underlap, $\mathbb{U}$, expresses a relation between parts. Two parts are said to underlap if there exists a part p_z of which p_x is a part and of which p_y is a part.

$$\mathbb{U}(p_x, p_y) \triangleq \exists p_z : \mathscr{P} \bullet \mathbb{P}(p_x, p_z) \wedge \mathbb{P}(p_y, p_z) \tag{6}$$

Think of the underlap p_z as an "umbrella" which both p_x and p_y are "under".
Over-cross, $\mathbb{OX}$, p_x and p_y are said to over-cross if p_x and p_y overlap and p_x is not part of p_y.

$$\mathbb{OX}(p_x, p_y) \triangleq \mathbb{O}(p_x, p_y) \wedge \neg \mathbb{P}(p_x, p_y) \tag{7}$$

[8] Our notation now is not RSL but a conventional first-order predicate logic notation.

Under-cross, $\mathbb{UX}$, p_x and p_y are said to under cross if p_x and p_y underlap and p_y is not part of p_x.

$$\mathbb{UX}(p_x, p_y) \triangleq \mathbb{U}(p_x, p_z) \wedge \neg\mathbb{P}(p_y, p_x) \tag{8}$$

Proper Overlap, $\mathbb{PO}$, expresses a relation between parts. p_x and p_y are said to properly overlap if p_x and p_y over-cross and if p_y and p_x over-cross.

$$\mathbb{PO}(p_x, p_y) \triangleq \mathbb{OX}(p_x, p_y) \wedge \mathbb{OX}(p_y, p_x) \tag{9}$$

Proper Underlap, $\mathbb{PU}$, p_x and p_y are said to properly underlap if p_x and p_y under-cross and p_x and p_y under-cross.

$$\mathbb{PU}(p_x, p_y) \triangleq \mathbb{UX}(p_x, p_y) \wedge \mathbb{UX}(p_y, p_x) \tag{10}$$

5.3. Satisfaction

We shall sketch a proof that the *model* of the previous section, Sect. 3, *satisfies* is a model for the *axioms* of this section. To that end we first define the notions of *interpretation*, *satisfiability*, *validity* and *model*. **Interpretation:** By an interpretation of a predicate we mean an assignment of a truth value to the predicate where the assignment may entail an assignment of values, in general, to the terms of the predicate. **Satisfiability:** By the satisfiability of a predicate we mean that the predicate is true for some interpretation. **Valid:** By the validity of a predicate we mean that the predicate is true for all interpretations. **Model:** By a model of a predicate we mean an interpretation for which the predicate holds.

5.3.1. A Proof Sketch

We assign

37 P as the meaning of $\mathscr{P}$
38 ATR as the meaning of $\mathscr{A}$,
39 imm_within as the meaning of $\mathbb{P}$,
40 trans_within as the meaning of $\mathbb{PP}$,
41 $\in$: ATTR×ATTRS-**set**→**Bool** as the meaning of $\in$: $\mathscr{A} \times \mathscr{P} \rightarrow$**Bool** and
42 sharing as the meaning of $\mathbb{O}$.

With the above assignments is is now easy to prove that the other axiom-operators $\mathbb{U}$, $\mathbb{PO}$, $\mathbb{PU}$, $\mathbb{OX}$ and $\mathbb{UX}$ can be modeled by means of imm_within, within, ATTR×ATTRS-**set**→**Bool** and sharing.

6. A Semantic CSP Model of Mereology

The model of Sect. 3 can be said to be an abstract model-oriented definition of the syntax of mereology. Similarly the axiom system of Sect. 5 can be said to be an abstract property-oriented definition of the syntax of mereology. We show that to every mereology there corresponds a program of cooperating sequential processes CSP. We assume that the reader has practical knowledge of Hoare's CSP [3].

6.1. Parts $\simeq$ Processes

The model of mereology presented in Sect. 3 focused on (i) parts, (ii) unique identifiers and (iii) mereology. To parts we associate CSP processes. Part processes are indexed by the unique part identifiers. The mereology reveals the structure of CSP channels between CSP processes.

6.2. Channels

We define a general notion of a vector of channels. One vector element for each "pair" of distinct unique identifiers. Vector indices are set of two distinct unique identifiers.

43 Let w be the "whole" (i.e., a part).
44 Let uis be the set of all unique identifiers of the "whole".
45 Let M be the type of messages sent over channels.
46 Channels provide means for processes to synchronise and communicate.

value
43. w:P
44. uis = **let** (_,uis′)=no_prts_uis(w) **in** uis′ **end**
type
45. M
channel
46. {ch[{ui,ui′}]:M|ui,ui′:UI•ui≠ui′ ∧ {ui,ui′}⊆uis}

47 We also define channels for access to external attribute values.

Without loss of generality we do so for all possible parts and all possible attributes.

channel
47. {xch[ui,an]:AVAL|ui:UI• ui ∈ uis,an:ANm}

6.3. Compilation

We now show how to compile "real-life, actual" parts into **RSL-Text**. That is, turning "semantics" into syntax !

```
value
  comp_P: P → RSL-Text
  comp_P(mkA(ui,me,attrs)) ≡ "ℳa(ui,me,attrs)"
  comp_P(mkC((ui,me,attrs),{p1,p2,...,pn})) ≡
    "ℳc(ui,me,attrs) ||
    " comp_process(p1)"||"comp_process(p2)"||"..."||"comp_process(pn)
```

The so-called core process expressions $\mathscr{M}_a$ and $\mathscr{M}_c$ relate to *a*tomic and *c*omposite parts. They are defined, schematically, below as just $\mathscr{M}$. The compilation expressions have two elements: (i) those embraced by double quotes: "...", and (ii) those that invoke further compilations, The first texts, (i), shall be understood as **RSL-Text**s. The compilation invocations, (ii), as expending into **RSL-Text**s. We emphasize the distinction between 'usages' and 'definitions'. The expressions between double quotes: "..." designate usages. We now show how some of these usages require "definitions". These 'definitions' are not the result of 'parts-to-processes' compilations. They are shown here to indicate, to the domain engineers, what must be further described, beyond the 'mere' compilations.

```
value
  ℳ: ui:UI×me:ME×attrs:ATTRS → ca:𝒞𝒜(attrs) → RSL-Text
  ℳ(ui,me,attrs)(ca) ≡
      let (me′,ca′) = ℱ(ui,me,attrs)(ca) in ℳ(ui,me′,attrs)(ca′) end
  ℱ: ui:UI×me:ME×attrs:ATTRS→ca:CA→
      in in_chs(ui,attrs) in,out in_out_chs(ui,me)→ME×CA ′
```

Recall (Page 15) that $\mathscr{C}_{\mathscr{A}}$(attrs) is a grouping, $(ca_1,ca_2,...,ca_{n_c})$, of controlled attribute values.

48 The in_chs function applies to a set of uniquely named attributes and yields some **RSL**-Text, in the form of **in**put channel declarations, one for each external attribute.

```
48.  in_chs: ui:UI × attrs:ATTRS → RSL−Text
48.  in_chs(ui,attrs) ≡ "in { xch[ui,xai] | xai:ANm • xai∈ℰ𝒜(attrs) }"
```

49 The in_out_chs function applies to a pair, a unique identifier and a mereology, and yields some **RSL**-Text, in the form of **in**put/**out**put channel declarations, one for each unique identifier in the mereology.

49. in_out_chs: ui:UI × me:ME → **RSL−Text**
49. in_out_chs(ui,me) ≡ "**in**,**out** { xch[ui,ui′]|ui:UI • ui'∈me }"

$\mathscr{F}$ is an action: it returns a possibly updated mereology and possibly updated controlled attribute values. We present a rough sketch of $\mathscr{F}$. The $\mathscr{F}$ action non-deterministically internal choice chooses between

- either [1,2,3,4]
  - [1] accepting input from
  - [4] a suitable ("offering") part process,
  - [2] optionally offering a reply;
  - [3] leading to an updated state;
- or [3,4]
  - [5] finding a suitable "order" (val)
  - [8] to a suitable ("inquiring") behaviour,
  - [6] offering that value,
  - [7] leading to an updated state;
- or [9] doing own work leading to an new state.

value
```
  F(ui,me,attrs)(ca) ≡
[1]     ⌷ {let val=ch[{ui,ui′}]? in
[2]         (ch[{ui,ui′}]!in_reply(val,(ui,me,attrs))(ca)) ;
[3]          in_update(val,(ui,me,attrs))(ca) end
[4]       | ui′:UI • ui′ ∈ me}
[5]  ⌈⌉ ⌷ {let val=await_reply(ui′,me,attrs)(ca) in
[6]         ch[{ui,ui′}]!val ;
[7]         out_update(val,(ui,me,attrs))(ca) end
[8]       | ui′:UI • ui′ ∈ me}
[9]  ⌈⌉   (me,own_work(ui,attrs)(ca))
```

```
in_reply: VAL×(ui:UI×me:ME×attrs:ATTRS)→ca:CA→
                  in in_chs(attrs) in,out in_out_chs(ui,me)→VAL
in_update: VAL × (ui:UI×me:ME×attrs:ATTRS)→ca:CA→
                  in,out in_out_chs(ui,me)→ME×CA
await_reply: (ui:UI,me:ME)→ca:CA→in,out in_out_chs(ui,me:ME)→VAL
out_update: (VAL×(ui:UI×me:ME<>attrs:ATTRS))→ca:CA→
                  in,out in_out_chs(ui,me)→ME×CA
own_work: (ui:UI×attrs:ATTRS)→CA→in,out in_out_chs(ui,me) CA
```

The above definitions of channels and core functions $\mathscr{M}$ and $\mathscr{F}$ are not examples of what will be compiled but of what the domain engineer must, after careful

analysis, “create”.

6.4. Discussion

6.4.1. General

A little more meaning has been added to the notions of parts and their mereology. The within and adjacent to relations between parts (composite and atomic) reflect a phenomenological world of geometry, and the mereological relation between parts reflect both physical and conceptual world understandings: physical world in that, for example, radio waves cross geometric “boundaries”, and conceptual world in that ontological classifications typically reflect lattice orderings where *overlaps* likewise cross geometric “boundaries”.

6.4.2. Specific

The notion of parts is far more general than that of [4]. We have been able to treat Stansław Leśniewski’s notion of mereology sôlely based on parts, that is, their semantic values, without introducing the notion of the syntax of parts. Our compilation functions are (thus) far more general than defined in [4].

7. Concluding Remarks

7.1. Relation to Other Work

The present contribution has been conceived in the following context.

My first awareness of the concept of ‘mereology’ was from listening to many presentations by **Douglas T. Ross** (1929–2007) at IFIP working group WG 2.3 meetings over the years 1980–1999. In [7] Douglas T. Ross and John E. Ward reports on the 1958–1967 MIT project for *computer-aided design (CAD) for numerically controlled production.*[9] Pages 13–17 of [7] reflects on issues bordering to and behind the concerns of mereology. Ross’ thinking is clearly seen in the following text: *“. . . our consideration of fundamentals begins not with design or problem-solving or programming or even mathematics, but with philosophy (in the old-fashioned meaning of the word) – we begin by establishing a “world-view”. We have repeatedly emphasized that there is no way to bound or delimit the potential areas of application of our system, and that we must be prepared to cope with any conceivable problem. Whether the system will assist in any way in the solution of a given problem is quite another matter, . . . , but in order to have a firm and uniform foundation, we must have a uniform philosophical basis upon which to approach any given problem. This “world-view” must provide a working framework and methodology in terms of which any aspect of our awareness of*

[9] Doug is said to have coined the term and the abbreviation CAD [8].

the world may be viewed. It must be capable of expressing the utmost in reality, giving expression to unending layers of ever-finer and more concrete detail, but at the same time abstract chimerical[10] *visions bordering on unreality must fall within the same scheme. "Above all, the world-view itself must be concrete and workable, for it will form the basis for all involvement of the computer in the problem-solving process, as well as establishing a viewpoint for approaching the unknown human component of the problem-solving team."* Yes, indeed, the philosophical disciplines of ontology, epistemology and mereology, amongst others, ought be standard curricula items in the computer science and software engineering studies, or better: domain engineers cum software system designers ought be imbued by the wisdom of those disciplines as was Doug. *"... in the summer of 1960 we coined the word plex to serve as a generic term for these philosophical ruminations. "Plex" derives from the word plexus, "An interwoven combination of parts in a structure", (Webster). ... The purpose of a* **'modeling plex'** *is to represent completely and in its entirety a "thing", whether it is concrete or abstract, physical or conceptual. A 'modeling plex' is a trinity with three primary aspects, all of which must be present. If any one is missing a complete representation or modeling is impossible. The three aspects of plex are* **data, structure,** *and* **algorithm.** *... "* which *"... is concerned with the behavioral characteristics of the plex model – the interpretive rules for making meaningful the data and structural aspects of the plex, for assembling specific instances of the plex, and for interrelating the plex with other plexes and operators on plexes. Specification of the algorithmic aspect removes the ambiguity of meaning and interpretation of the data structure and provides a complete representation of the thing being modeled."* In the terminology of the current paper a plex is a part (whether composite or atomic), the data are the properties (of that part), the structure is the mereology (of that part) and the algorithm is the process (for that part). Thus Ross was, perhaps, a first instigator (around 1960) of object-orientedness. A first, "top of the iceberg" account of the mereology-ideas that Doug had then can be found in the much later (1976) three page note [9]. Doug not only 'invented' CAD but was also the father of AED (Algol Extended for Design), the Automatically Programmed Tool (APT) language, SADT (Structured Analysis and Design Technique) and helped develop SADT into the IDEF0 method for the Air Force's Integrated Computer-Aided Manufacturing (ICAM) program's IDEF suite of analysis and design methods. Douglas T. Ross went on for many years thereafter, to deepen and expand his ideas of relations between mereology and the programming language concept of type at the IFIP WG2.3 working group meet-

[10]Chimerical: existing only as the product of unchecked imagination: fantastically visionary or improbable

ings. He did so in the, to some, enigmatic, but always fascinating style you find on Page 63 of [9].

In [10] **Henry S. Leonard** and **Henry Nelson Goodman:** *A Calculus of Individuals and Its Uses* present the American Pragmatist version of Leśniewski's mereology. It is based on a single primitive: *discreet.* The idea of the calculus of individuals is, as in Leśniewski's mereology, to avoid having to deal with the empty sets while relying on explicit reference to classes (or parts).

[2] **R. Casati** and **A. Varzi:** *Parts and Places: the structures of spatial representation* has been the major source for this paper's understanding of mereology. Although our motivation was not the spatial or topological mereology, [11], and although the present paper does not utilize any of these concepts' axiomatision in [2, 11] it is best to say that it has benefited much from these publications.

Domain descriptions, besides mereological notions, also depend, in their successful form. on FCA: Formal Concept Analysis. Here a main inspiration has been drawn , since the mid 1990s from **B. Ganter** and **R. Wille's** *Formal Concept Analysis — Mathematical Foundations* [12]. *The approach takes as input a matrix specifying a set of objects and the properties thereof, called attributes, and finds both all the "natural" clusters of attributes and all the "natural" clusters of objects in the input data, where a "natural" object cluster is the set of all objects that share a common subset of attributes, and a "natural" property cluster is the set of all attributes shared by one of the natural object clusters. Natural property clusters correspond one-for-one with natural object clusters, and a concept is a pair containing both a natural property cluster and its corresponding natural object cluster. The family of these concepts obeys the mathematical axioms defining a lattice, a Galois connection).* Thus the choice of adjacent and embedded ('within') parts and their connections is determined after serious formal concept analysis.

7.2. What Has Been Achieved ?

We have given a model-oriented specification of mereology. We have indicated that the model satisfies a widely known axiom system for mereology. We have suggested that (perhaps most) work on mereology amounts to syntactic studies. So we have suggested one of a large number of possible, schematic semantics of mereology. And we have shown that to every mereology there corresponds a set of communicating sequential process (CSP).

7.3. Future Work

I hereby offer collaboration with, say, a young PhD student, to furnish a formal proof instead of the sketch outline in Sect. 5.3.1 on Page 20. We need to characterise, in a proper way, the class of CSP programs for which there corresponds

a mereology. Are you game ? One could also wish for an extensive editing and publication of Doug Ross' surviving notes.

8. Bibliography

8.1. Bibliographical Notes

This paper shall be seen in the context of the following other papers:

- [4, Manifest Domains: Analysis & Description] lays the foundation for analysing & describing a large class of domains. It introduces two calculi of analysis and of description prompts.
- [13, Domain Facets: Analysis & Description] continues that of [4] by enlarging the scope of phenomena being analysed and described.
- [14, Formal Models of Processes and Prompts] presents an operational semantics for the analysis and description processes and prompts covered in [4].
- [5, From Domain Descriptions to Requirements Prescriptions] shows how to systematically 'derive' core elements of requirements from domain descriptions.
- [15, Domains: Their Simulation, Monitoring and Control] discusses software product lines of demos, simulators, monitors and monitors & controllers as they relate to descriptions and prescriptions for the product line domain.

Together these papers, the present and those referenced above, form a scientific and engineering basis for domain engineering.

References

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

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

[3] C. A. R. Hoare, Communicating Sequential Processes, C.A.R. Hoare Series in Computer Science, Prentice-Hall International, 1985, published electronically: http://www.usingcsp.com/cspbook.pdf (2004).

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

[5] D. Bjørner, From Domain Descriptions to Requirements Prescriptions – A Different Approach to Requirements Engineering, Submitted for consideration to Formal Aspects of Computing.

[6] D. Bjørner, Pipelines – a Domain Description: `http://www.imm.dtu.dk/-~dibj/pipe-p.pdf`, Experimental Research Report 2013-2, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark (Spring 2013).

[7] D. T. Ross, J. E. Ward, Investigations in computer-aided design for numerically controlled production, Final Technical Report ESL-FR-351, , 1 December 1959 – 3 May 1967. Electronic Systems Laboratory Electrical Engineering Department, MIT, Cambridge, Massachusetts 02139 (May 1968).

[8] D. T. Ross, Computer-aided design, Commun. ACM 4 (5) (1961) 41–63.

[9] D. T. Ross, Toward foundations for the understanding of type, in: Proceedings of the 1976 conference on Data: Abstraction, definition and structure, ACM, New York, NY, USA, 1976, pp. 63–65, http://doi.acm.org/10.1145/800237.807120.

[10] H. S. Leonard, N. Goodman, The Calculus of Individuals and its Uses, Journal of Symbolic Logic 5 (1940) 45–44.

[11] B. Smith, Mereotopology: A Theory of Parts and Boundaries, Data and Knowledge Engineering 20 (1996) 287–303.

[12] B. Ganter, R. Wille, Formal Concept Analysis — Mathematical Foundations, Springer-Verlag, January 1999.

[13] D. Bjørner, Domain Facets: Analysis & Description, Submitted for consideration to Formal Aspects of ComputingHttp://www.imm.dtu.dk/~dibj/-2016/facets/faoc-facets.pdf.

[14] D. Bjørner, Domain Analysis and Description – Formal Models of Processes and Prompts, Submitted for consideration to Formal Aspects of ComputingHttp://www.imm.dtu.dk/~dibj/2016/process/process-p.pdf.

[15] D. Bjørner, Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions, Experimental Research, Fredsvej 11, DK–2840 Holte, Denmark, http://www.imm.dtu.dk/~dibj/2016/demos/-faoc-demo.pdf (2016).