Container Terminals
An Initial Domain Analysis & Description Sketch

Dines Bjørner

Fredsvej 11, DK-2840 Holte and DTU, DK-2800 Kgs. Lyngby, Denmark.

e-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~dibj

. **November 22, 2018: 11:05 am**

The ECNU November 2018 Course Project

1

Abstract

We present a recording of stages and steps of a development of a domain analysis & description of an answer to he question: *what, mathematically, is a container terminal?*

Caveats: The present, . **November 22, 2018: 11:05 am**, version of this document is "vastly" incomplete". It is being distributed only to a limited circle of people so that they can see that my ECNU course project proposal is one of substance. Being incomplete, it is rich on incomplete formulas and poor on explanatory text; and can only be understood, i.e., appreciated, that is, requires non-trivial knowledge of:

Dines Bjørner.
Manifest Domains: Analysis & Description.
Formal Aspects of Computing, 29(2):175–225,
Online: July 2016. DOI 10.1007/s00165-016-0385-z
Springer, BCS copyright 2016

Limited Circulation:

This document constitutes my preparation for a possible student project at ECNU in November 2018.
The students are themselves to analyse & describe a domain of container terminals.
Therefore, please, do not circulate this document!

Contents

Abstract

This is a report on an experiment. At any stage of development, and the present draft stage is judged 2/3 “completed” it reflects how I view an answer to the question *what is a container terminal port ?* mathematically speaking.

1 Introduction 3

TO BE WRITTEN

1.1 Survey of Literature on Container-related Matters 4

[1, A Container Line Industry Domain, 2007]

[2, A-Z Dictionary of Export, Trade and Shipping Terms]

[3, Portworker Development Programme: PDP Units]

[4, An interactive simulation model for the logistics planning of container operations in seaports,1996]

[5, Stowage planning for container ships to reduce the number of shifts, 1998]

[6, Container stowage planning: a methodology for generating computerised solutions, 2000]

[7, Container ship stowage problem: complexity and connection to the coloring of circle graphs, 2000] 5

[8, Container stowage pre-planning: using search to generate solutions, a case study, 2001]

[9, A genetic algorithm with a compact solution encoding for the container ship stowage problem, 2002]

[10, Multi-objective ... stowage and load planning for a container ship with container rehandle ..., 2004]

[11, Container terminal operation and operations research - a classification and literature review, 2004]

[12, Online rules for container stacking, 2010]

2 Some Pictures 6

2.1 Terminal Port Container Stowage Area

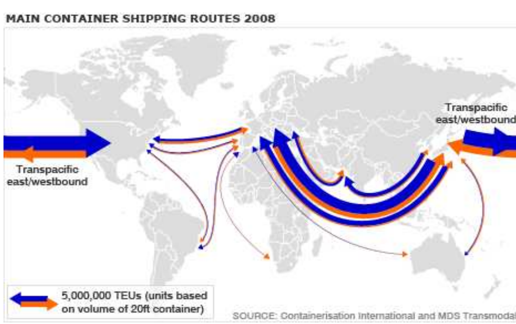
7

Analysis of the above picture:

- The picture shows a *terminal*.
- At bottom we are hinted (through shadows) at *quay cranes* serving (unshown) *vessels*.
- Most of the picture shows a *container stowage area*, here organsed as a series of columns, from one side of the picture to the other side, e.g., left-to-right, sequences (top-to-bottom) of [blue] *bays* with *rows* of *stacks* of *containers*.
- Almost all columns show just one *bay*.
- Three "rightmost" columns show many [non-blue] *bays*.
- Most of the column "tops" and "bottoms" show *stack cranes*.
- The four leftmost columns show *stack cranes* at *bays* "somewhere in the middle" of a column.

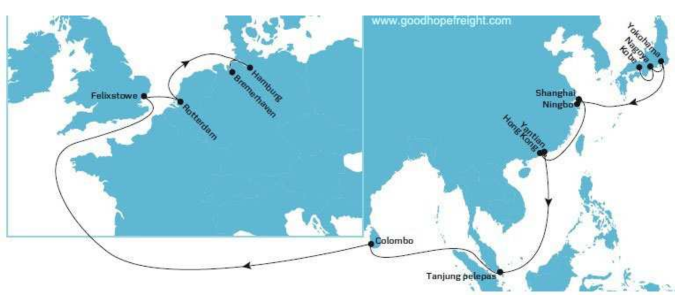
2.2 Container Stowage Area and Quay Cranes 8

2.3 Container Vessel Routes 9



10

11



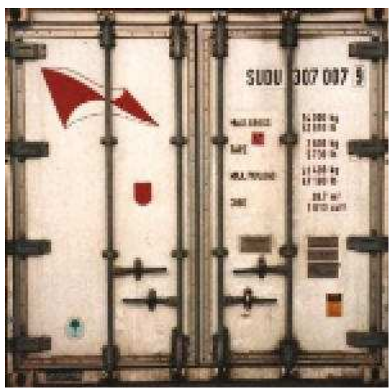
12

2.4 Containers 13

2.4.1 40 and 20 Feet Containers


14

2.4.2 Container Markings 15



2.5 Container Vessels 16


17

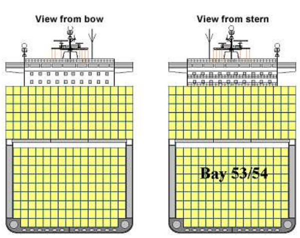
18

Quay cranes and *vessel* showing *row* of *aft (rear) bay*.

2.6 Container Stowage Area: Bays Rows, Stacks and Tier 19

20

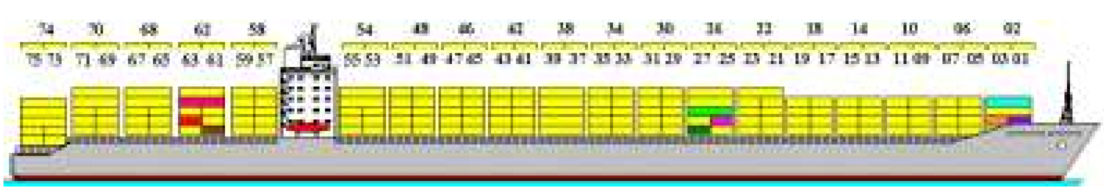
Bay, Row, Tier Numbers. Row Numbers 21



22

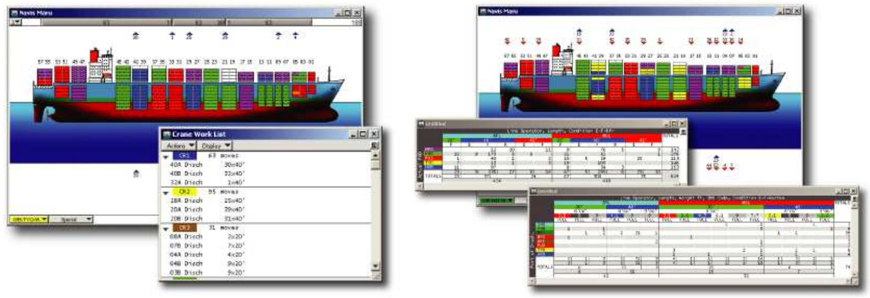
Cross section of a Bay. Tier Numbers.

23



Bay Numbering

2.7 Stowage Software 24

25 

2.8 Quay Cranes 26

2.9 Container Stowage Area and Stack Cranes 27

28

29

2.10 Container Stowage Area 30


31

2.11 Quay Trucks 32

33

2.12 Map of Shanghai and YangShan 34

3 SECT 35

- *Shanghai East Container Terminal*

  - ◈ is the joint venture terminal of
  - ◈ *APM Terminals* and
  - ◈ *Shanghai International Port Group*
  - ◈ in *Wai Gao Qiao* port area of *Shanghai*.

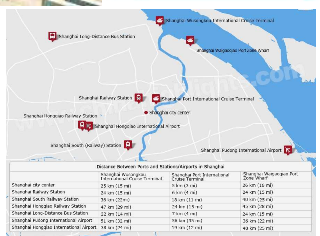
- No.1 Gangjian Road, Pudong New District, Shanghai, China

36

37

38

39



4 Main Behaviours

40

41

- From consumer/origin to consumer/final destination:
  - **container loads** onto **land truck**;
  - **land truck travels** to **terminal stack**;
  - **container unloads** by means of **terminal stack crane** from **land truck** onto **terminal stack**.

- **Container moves** from **stack** to **vessel**:
  - **terminal stack crane moves container** from **terminal stack** to **quay truck**,
  - **quay truck moves container** from **terminal stack** to **quay**,
  - **quay crane moves container** to **top of a vessel stack**;
- **Container moves** on **vessel** from **terminal** to **terminal**:
  - Either container is unloaded at a next terminal port to a stack and from there to a container truck
  - or: container is unloaded at a next terminal port to a stack and from there to a next container vessel.

4.1 A Diagram 42

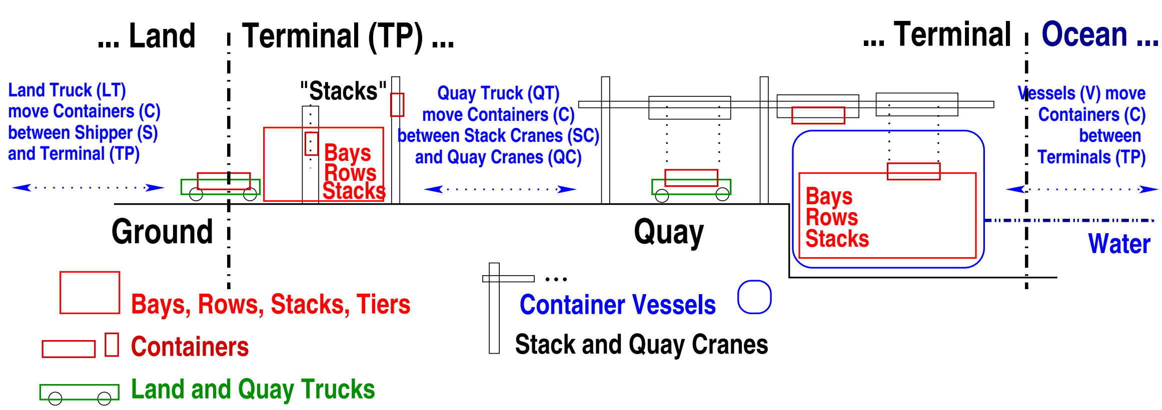


Fig. 1: Container Terminal Ports, I
A "from the side" snapshot of terminal port activities

4.2 Terminology - a Caveat 43

Bay [1]: contains indexed set of *rows* (of stacks of containers).

Container : smallest unit of central (i.e., huge) concern !

Container Stowage Area : An area of a vessel or a terminal where containers are stored, during voyage, respectively awaiting to be either brought out to shippers or onto vessels.

[1]The terms introduced in this section are mine. They are most likely not the correct technical terms of the container shipping and stowage trade. I expect to revise this section, etc.

Crane :

> **Stack Crane** : moves *containers* between *land* or *terminal trucks* and *terminal stacks*.
>
> **Quay Crane** : moves *containers* between [*land* or] *terminal trucks* and *vessels*. 44

Land : ... as you know it ...

Ocean : ... as you know it ...

Shipper : arranges shipment of containers with container lines

Quay : area of terminal next to vessels (hence water).

Row : contains indexed set of *stacks* (of containers).

Stack : contains indexed set of *containers*.

> We shall also, perhaps confusingly, use the term stack referring to the land-based bays of a terminal.

Terminal : area of land and water between land and ocean equipped with container stowage area, and stack and quay cranes, etc. 45

Truck :

> **Land Truck** : privately operated truck transport *containers* between *shippers* and *stack cranes*.
>
> **Quay Truck** : terminal operated special truck transport *containers* between *stack cranes* and *quay cranes*.

Tier : index of *container* in *stack*.

Vessel : contains a *container stowage area*.

4.3 Assumptions 46

Without loss of generality we can assume that there is exactly one stack crane per land-basederminal stack; quay cranes each serve exactly one bay on a vessel; there are enough quay cranes to serve all bays of any berthed vessel; quay trucks may serve any (quay and stack) crane; land trucks may serve more than one terminal; et cetera.

5 Endurants

47

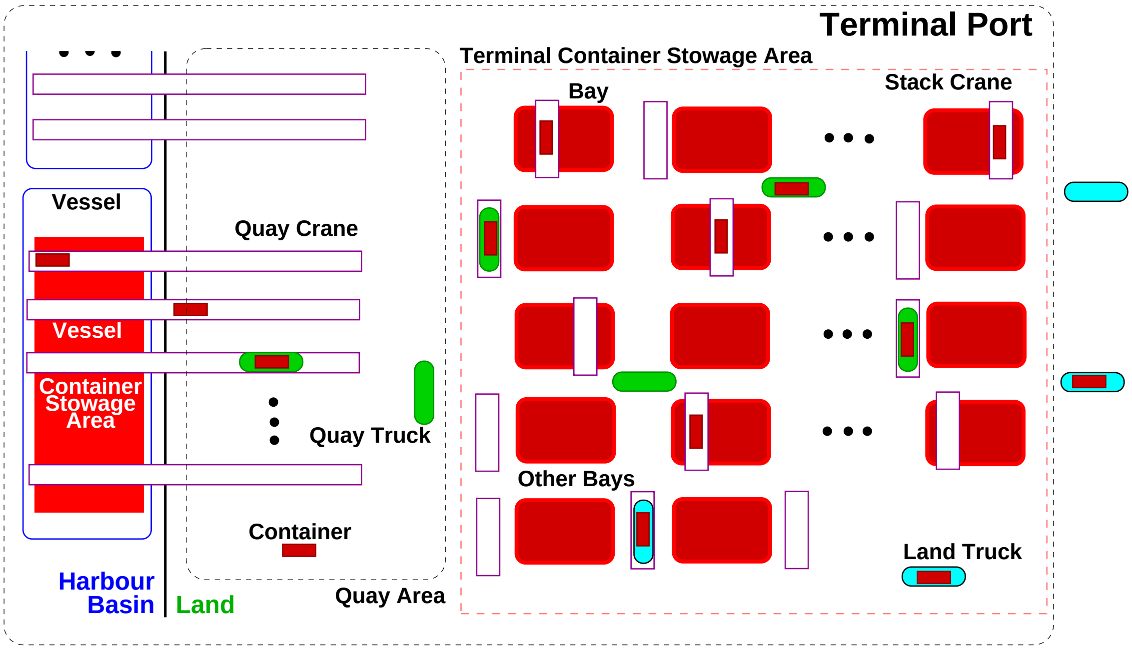


Fig. 2: Container Terminal Ports, II
A "from above" snapshot of terminal port activities

We refer to [13, Sects. 3., 4., and 5.].

Our model focuses initially on parts, that is, manifest, observable phenomena. Our choice of these is expected to be subject to serious revision once we ... MORE TO COME ...

5.1 Parts

48

We refer to [13, Sect. 3.3].

Our model has, perhaps arbitrarily, focused on just some of the manifest, i.e., observable parts of a domain of container terminal ports. We shall invariable refer to container terminal ports as either container terminals, or terminal ports, tp:TP, or just terminals. We expect revisions to the decomposition as shown as we learn more from professional stakeholders, e.g., *APM Terminals/SECT*, Shanghai.

1 In the container line industry, CLI, we can observe

2 a structure, TPS, of all terminal ports, and from each such structure, an indexed set, TPs, of two or more container *terminal* ports, TP;

3 a structure, VS, of all container vessels, and from each such structure, an indexed set, Vs, of one or more container *vessels*, V; and

4 a structure, LTS, of all land trucks, and from each such structure, a non-empty, indexed set, LTs of land trucks, LT;

49

```
type
1   CLI
2   STPs,  TPs = TP-set, TP
3   SVs,    Vs = V-set, V
9   SLTs,   LTs = LT-set, LT
value
2   obs_STPs: CLI → STPs, obs_TPs: STPs → TPs
3   obs_SVs:  CLI → SVs,  obs_Vs: SVs → Vs
9   obs_SLTs:  CLI → SLTs, obs_LTs: SLTs → LTs
axiom
2   ∀ cli:CLI•card obs_TPs(obs_STPs(cli))≥2
3      ∧ card obs_Vs(obs_SVs(cli))≥1
9      ∧ card obs_LTs(obs_SLTs(cli))≥1
```

5.1.1 Terminal Ports 50

In a terminal port, tp:TP, one can observe

5 a [composite] container stowage area, csa:CSA;

6 a structure, sqc:SQC, of quay cranes, and from that, a non-empty, indexed set, qcs:QCs, of one or more quay cranes, qc:QC;

7 structure, sqt:SQT, of quay trucks, and from that a non-empty, indexed set, qts:QTs, of quay trucks, qt:QT;

8 a structure, Scs:SCS, of stack cranes, and from that a non-empty, indexed set, scs:SCs, of one or more stack cranes, sc:SC;

9 a[n atomic] quay[2], q:Q[3]; and

10 a[n atomic] terminal port monitoring and control center, mcc:MCC.

51

[2]We can, without loss of generality, describe a terminal as having exactly one quay (!) – just as we, again without any loss of generality, describe it as having exactly one container stowage area.

[3]*Quay:* a long structure, usually built of stone, where boats can be tied up to take on and off their goods.
Pronunciation: key.
Thesaurus: berth, jetty, key, landing, levy, slip, wharf

type
5 CSA
6 SQC, QCs = QC-**set**, QC
7 SQT, QTs = QT-**set**, QT
8 SCS, SCs = SC-**set**, SC
9 Q
10 MCC
value
5 obs_CSA: TP → CSA
6 obs_SQC: TP → SQC, obs_QCs: SQC → QCs
7 obs_SQT: TP → SQT, obs_QTs: SQT → QTs
8 obs_SCS: TP → SCS, obs_SCs: SCS → SCs
9 obs_Q: TP → Q
10 obs_MCC: TP → MCC
axiom
6 ∀ sqc:SQC•**card** obs_QCs(sqc)≥1
7 ∀ sqt:SQT•**card** obs_QTs(sqt)≥1
8 ∀ scs:SCS•**card** obs_SCs(scs)≥1

5.1.2 Quays 52

Although container terminal port quays can be modelled as composite parts we have chosen to describe them as atomic. We shall subsequently endow the single terminal port quay with such attributes as quay segments, quay positions and berthing[4].

5.1.3 Container Stowage Areas: Bays, Rows and Stacks 53

11 From a container stowage area one can observe a non-empty indexed set of bays,

12 From a bay we can observe a non-empty indexed set of rows.

13 From a row we can observe a non-empty indexed set of stacks.

14 From a stack we can observe a possibly empty indexed set of containers.

54

type
11 BAYS, BAYs = BAY-**set**, BAY
12 ROWS, ROWs = ROW-**set**, ROW

[4]Berth: Sufficient space for a vessel to maneuver; a space for a vessel to dock or anchor; (whether occupied by vessels or not). Berthing: To bring (a vessel) to a berth; to provide with a berth.

```
13  STKS, STKs = STK-set, STK
14  CONS, CONs = CON-set, CON
value
11  obs_BAYS: CSA → BAYS, obs_BAYs: BAYS → BAYs
12  obs_ROWS: BAY → ROWS, obs_ROWs: ROWS → ROWs
13  obs_STKS: ROW → STKS, obs_STKs: STKS → STKs
14  obs_CONS: STK → CONS, obs_CONs: CONS → CONs
axiom
11  ∀ bays:BAYs • card bays > 0
12  ∀ rows:ROWs • card rows > 0
13  ∀ stks:STKs • card stks > 0
```

5.1.4 Vessels 55

From (or in) a vessel one can observe

15 [5] a container stowage area

16 and some other parts.

```
type
5   CSA
16    ...
value
5   obs_CSA: V → CSA
16    ...
```

5.1.5 Functions Concerning Container Stowage Areas 56

17 One can calculate

18 the set of all container storage areas:

19 of all terminal ports together with those

20 of all container lines.

```
value
17  cont_stow_areas: CLI → CSA-set
18  cont_stow_areas(cli) ≡
19      {obs_CSA(tp)|tp:TP•tp ∈ obs_TPs(obs_TPS(cli))}
20    ∪ {obs_CSA(cl)|cl:CL•cl ∈ obs_CLs(obs_CLS(cli))}
```


One can calculate the containers of 57

21 a stack,

22 a row,

23 a bay, and

24 a container stowage area.

value

```
21  extr_cons_stack: STK → CONs
21  extr_cons_stack(stk)≡obs_CONs(obs_CONS(stk))
22  extr_cons_row: ROW → CONs
22  extr_cons_row(row) ≡
22     {obs_CONs(obs_CONS(stk))|stk:STK•stk∈obs_STKs(obs_STKS(stk))}
23  extr_cons_bay: BAY → CONs
23  extr_cons_bay(bay) ≡
23     {obs_CONs(obs_CONS(row))|row:ROW•row∈obs_ROWs(obs_ROWS(bay))}
24  extr_cons_csa: CSA → CONs
24  extr_cons_csa(csa) ≡
24     {obs_CONs(obs_CONS(bay))|bay:BAY•bay∈obs_BAYs(obs_BAYS(csa))}
```

5.1.6 Axioms Concerning Container Stowage Areas 58

25 All rows contain different, i.e. distinct containers.

26 All bays contain different, i.e. distinct containers.

27 All container stowage areas contain different, i.e. distinct containers.

value

```
25  ∀ cli:CLI •
25     ∀ csa,csa′:CSA•{csa,csa′}⊆cont_stow_areas(cli) •
25        ∀ row,row′:ROW •
25           {row,row′}⊆obs_ROWs(obs_ROWS(csa))∪obs_ROWs(obs_ROWS(csa′))⇒
25           extr_cons_row(row) ∩ extr_cons_row(row′) = {} ∧
26        ∀ bay,bay′:BAY •
26           {bay,bay′}⊆obs_ROWs(obs_ROWS(csa))∪obs_ROWs(obs_ROWS(csa′))⇒
26           extr_cons_bay(bay) ∩ extr_cons_bay(bay′) = {} ∧
27           extr_cons_csa(csa) ∩ extr_cons_csa(csa′) = {}
```

5.1.7 Stacks 59

An aside: We shall use the term ‘stack’ in two senses: (i) as a component of container storage area bays; and (ii) to refer to the collection of stacks in a bay of a terminal container storage area. 60

28 Stacks are created empty, and hence stacks can be *empty*.

29 One can *push* a *container* onto a *stack* and obtain a *non-empty stack*.

30 One can *pop* a *container* from a *non-epmpty stack* and obtain a pair of a *container* and a possibly empty *stack*.

value
28 empty: () → STK, is_empty: STK → **Bool**
29 push: CON × STK → STK
30 pop: STK $\xrightarrow{\sim}$ (CON × STK)
axiom
28 is_empty(empty()), ∼is_empty(push(c,stk))
29 pop(push(c,stk)) = (c,stk)
30 **pre** pop(stk),pop(push(c.stk)): ∼is_empty(stk)
30 pop(empty()) = **chaos**

5.2 Terminal Port Command Centers 61

5.2.1 Discussion

We consider terminal port monitoring & control command centers to be atomic parts. The purpose of a terminal port command center is to monitor and control the allocation and servicing (berthing) of any visiting vessel to quay positions and by quay cranes, the allocation and servicing of vessels by quay cranes, the allocation and servicing of quay cranes by quay trucks, the allocation and servicing of quay trucks to quay cranes, containers and terminal stacks, the allocation and servicing of land trucks to containers and terminal stacks, This implies that there are means for communication between a terminal command center and vessels, quay cranes, stack cranes, quay trucks, land trucks, terminal stacks and containers. 62

5.2.2 Justification 63

We shall justify the concept of terminal monitoring & control, i.e., command centers. First, using the *domain analysis & description* approach of [13], we know that we are going, through a transcendental deduction, to model certain parts as behaviours. These

parts, we decide, after some analysis that we forego, to be vessels, quay cranes, quay trucks, stack cranes stacks, land trucks, and containers. Behaviours are usually like 64
actors:they can instigate actions. But we decide, in our analysis, that some of these behaviours, quay cranes, quay trucks, stack cranes and stacks, are *"passive" actors:* are
65 behaviourally not endowed with being able to initiate "own" actions. Instead, therefore, of all these behaviours, being able to communicate directly, pairwise, as loosely indicated by the figures of Pages 16 and 18, we model them to communicate *via* their terminal command centers.

This is how we justify the introduction of the concept of terminal command centers.
66 They are an abstraction. In *"ye olde days"* you could observe, not one, but, perhaps, a hierarchy of terminal port offices, staffed by people, [each office, each group of staff] with its set of duties: communicating (by radio-phone) with approaching [and departing] vessels; scheduling quay positions, quay cranes and quay trucks; managing the operation of cranes and trucks; and, on a large scale, calculating stowage: on
67 vessels and in terminals. Today, *"an age of ubiquitous computing"*, most of these offices and their staff are replaced by electronics: sensors, actuators, communication and computing, and with massive stowage data processing: where should containers be stowed on board vessels and in terminals so as to near-optimise all operations.

5.3 Unique Identifications 68

We refer to [13, Sect. 5.1].

31 Vessels have unique identifiers.

32 Quay cranes have unique identifiers.

33 Quay trucks have unique identifiers.

34 Stack cranes have unique identifiers.

35 Bays ("Stacks") of terminal container stowage areas have unique identifiers, cf. Item 39.

36 Land trucks have unique identifiers.

37 Terminal port command centers have unique identifiers.

38 Containers have unique identifiers.

39 Bays of container stowage areas have unique identifiers.

40 Rows of a bay have unique identifiers.

41 Stacks of a row have unique identifiers.

42 The part unique identifier types are mutually disjoint.

69

type
31 VI
32 QCI
33 QTI
34 SCI
35 TBI

36 LTI
37 MCCI
38 CI
39 BI
40 RI
41 SI
axiom
42 VI, QCI, QTI, SCI, TBI, LTI, MCCI, CI, BI, RI and SI mutually disjoint
42 TBI $\subset$ BI

31 uid_V: V $\rightarrow$ VI
32 uid_QC: QC $\rightarrow$ QCI
33 uid_QT: QT $\rightarrow$ QTI
34 uid_SC: SC $\rightarrow$ SCI
34 uid_TBI: BAY $\rightarrow$ TBI
35 uid_LT: LT $\rightarrow$ LTI
37 uid_MCC: MCC $\rightarrow$ MCCI
37 uid_CON: CON $\rightarrow$ CI
34 uid_BAY: BAY $\rightarrow$ BI
35 uid_ROW: ROW $\rightarrow$ RI
36 uid_STK: STK $\rightarrow$ SI

value

5.3.1 Unique Identifiers: Distinctness of Parts 70

43 If two containers are different then their unique identifiers must be different.

axiom
43 $\forall$ con,con$'$:CON $\bullet$ con $\neq$ con$'$ $\Rightarrow$ uid_CON(con) $\neq$ uid_CON(con$'$)

The same distinctness criterion applies to stacks, rows, bays, container storage areas, terminal ports, cranes, vessels, etc.

5.3.2 Unique Identifiers: Two Useful Abbreviations 71

Container positions within a container stowage area can be represented in two ways:

44 by a triple of a bay identifier, a row identifier and a stack identifier, and

45 by these three elements and a tier position (i.e., position within a stack).

44 BRS = BI $\times$ RI $\times$ SI
45 BRSP = BI $\times$ RI $\times$ SI $\times$ **Nat**
axiom
45 $\forall$ (bu,ri,si,n):BRSP $\bullet$ n>0

5.3.3 Unique Identifiers: Some Useful Index Set Selection Functions 72

46 From a container stowage area once can observe all bay identifiers.

47 From a bay once can observe all row identifiers.

48 From a row once can observe all stack identifiers.

49 From a virtual container storage area, i.e., an icsa:iCSA, one can extract all the unique container identifiers.

73

```
value
46  xtr_BIs: CSA → BI-set
46  xtr_BIs(csa) ≡ {uid_BAY(bay)|bay:BAY•bay ∈ xtr_BAYs(csa)}

46  xtr_RIs: BAY → RI-set
47  xtr_RIs(bay) ≡ {uid_ROW(bay)|row:ROW•row ∈ obs_ROWs(bay)}

46  xtr_SIs: ROW → SI-set
48  xtr_SIs(row) ≡ {uid_STK(row)|stk:STK•stk ∈ obs_STKs(row)}

49  xtr_CIs: iCSA → CI-set
49  xtr_CIs(icsa) ≡
49     ... [to come] ...
```

5.3.4 Unique Identifiers: Ordering of Bays, Rows and Stacks 74

The bays of a container stowage area are usually ordered. So are the rows of bays, and stacks of rows. Ordering is here treated as *attributes* of container stowage areas, bays and stacks. We shall treat *attributes* further on.

5.4 States, Global Values and Constraints 75

5.4.1 States

50 We postulate a container line industry *cli*:CLI.

From that we observe, successively, all parts:

51 the set, *cs*:C-**set**, of all containers;

52 the set, *tps*:TPs, of all terminal ports;

53 the set, *vs*:Vs, of all vessels; and

54 the set, *lts*:LTs, of all land trucks.

value
50 cli:CLI
51 cs:C-**set** = obs_Cs(obs_CS(cli))
52 tps:TP-**set** = obs_TPs(obs_TPS(cli))
53 vs:V-**set** = obs_Vs(obs_VS(cli))
54 lts:LTs = obs_LTs(obs_LTS(cli))

We can observe 76

55 $csas$:CSA-**set**, the set of all terminal port container stowage areas of all terminal ports;

56 $bays$:BAY-**set**, the terminal port bays of all terminals;

57 the set, qcs:QC-**set**, of all quay cranes of all terminals;

58 the set, qts:QT-**set**, of all quay trucks of all terminal ports; and

59 the set, scs:SC-**set**, of all terminal (i.e., stack) cranes of all terminal ports.

value
55 $csas$:CSA-**set** = {obs_CSA(tp)|tp:TP•tp ∈ tps}
55 $bays$:BAY-**set** = {obs_BAY(csa)|csa:CSA•csa ∈ $csas$}
57 qcs:QC-**set** = {obs_QCs(obs_QCS(tp))|tp:TP•tp ∈ tps}
58 qts:QT-**set** = {obs_QTs(obs_QTS(tp))|tp:TP•tp ∈ tps}
59 scs:SC-**set** = {obs_SCs(obs_SCS(tp))|tp:TP•tp ∈ tps}

5.4.2 Unique Identifiers 77

Given the generic parts outlined in Sect. 5.4.1 we can similarly define generic sets of unique identifiers.

60 There is the set, c_uis, of all container identifiers;

61 the set, tp_uis, of all terminal port identifiers;

62 the set, mcc_uis, of all terminal port command center identifiers;

63 the set, v_uis, of all vessel identifiers;

64 the set, qc_uis, of quay crane identifiers of all terminal ports;

65 the set, qt_uis, of quay truck identifiers of all terminal ports;

66 the set, *sc_uis*, of stack crane identifiers of all terminal ports;

67 the set, *stk_uis*, of stack identifiers of all terminal ports;

68 the set, *lt_uis*, of all land truck identifiers; and

69 the set, *uis*, of all vessel, crane and truck identifiers.

78

value
60 c_uis:CI-**set** = $\{$uid_C(c)|c:C•c$\in cs\}$
61 tp_uis:TPI-**set** = $\{$uid_TP(tp)|tp:TP•tp$\in tps\}$
62 mcc_uis:TPI-**set** = $\{$uid_MCC(obs_MCC(tp))|tp:TP•tp$\in tps\}$
63 v_uis:VI-**set** = $\{$uid_V(v)|v:V•v$\in vs\}$
64 qc_uis:QCI-**set** = $\{$uid_QC(qc)|qc:QC•qc$\in qcs\}$
65 qt_uis:QTI-**set** = $\{$uid_QT(qt)|qt:QT•qt$\in qts\}$
66 sc_uis:SCI-**set** = $\{$uid_SC(sc)|sc:SC•sc$\in scs\}$
67 stk_uis:BI-**set** = $\{$uid_BAY(stk)|stk:BAY•stk$\in stks\}$
68 lt_uis:LTI-**set** = $\{$uid_LL(lt)|lt:LT•lt$\in lts\}$
69 uis:(VI|QCI|QTI|SCI|BI|LTI)-**set** = $v\_uis \cup qc\_uis \cup qt\_uis \cup sc\_uis \cup stk\_uis \cup lt\_uis$

79

70 the map, *tpmcc_idm*, from terminal port identifiers into the identifiers of respective command centers;

71 the map, *mccqc_idsm*, from command center identifiers into the set of quay crane identifiers of respective ports;

72 the map, *mccqt_idsm*, from command center identifiers into the identifiers of quay trucks of respective ports;

73 the map, *mccsc_idsm*, from command center identifiers into the identifiers of quay trucks of respective ports; and

74 the map, *mccbays_idsm*, from command center identifiers into the set of bay identifiers (i.e., "stacks") of respective ports;

80

value
70 $tpmcc\_idm$:(TI $\xrightarrow[m]{}$ MCCI) = [uid_TP(tp)$\mapsto$uid_MCC(obs_MCC(tp))|tp:TP•tp $\in tps$]
71 $mccqc\_idsm$:(MCCI $\xrightarrow[m]{}$ QCI-**set**)
71 = [$tpmcc\_uim$(uid_TP(tp)) $\mapsto$ { uid_QC(qc)
71 | qc:QC • qc $\in$ obs_QCs(obs_QCS(tp)) } | tp:TP•tp $\in tps$]
72 $mccqt\_idsm$:(MCCI $\xrightarrow[m]{}$ QTI-**set**) =

```
72      = [ tpmcc_uim(uid_TP(tp)) ↦ { uid_QT(qt)
72                | qt:QT • qt ∈ obs_QTs(obs_QTS(tp)) } | tp:TP•tp ∈ tps ]
73  mccsc_idsm:(MCCI →m SCI-set)
73      = [ tpmcc_uim(uid_TP(tp)) ↦ { uid_SC(sc)
73                | sc:SC • sc ∈ obs_SCs(obs_SCS(tp)) } | tp:TP•tp ∈ tps ]
74  mccbays_idsm:(MCCI →m BI-set)
74      = [ tpmcc_uim(uid_TP(tp)) ↦ { uid_B(b)
74                | b:BAY•b ∈ obs_BAYs(obs_BAYS(obs_CSA(tp)))} | tp:TP•tp ∈ tps ]
```

5.4.3 Some Axioms on Uniqueness 81

TO BE WRITTEN

5.5 Mereology 82

We refer to [13, Sect. 5.2].

5.5.1 Physical versus Conceptual Mereology

We briefly discuss a distinction that was not made in [13]: whether to base a mereology on *physical connections* or on *functional* or, as we shall call it, *conceptual relations*. We 83
shall, for this domain model, choose the conceptual view. The physical mereology view can be motivated, i.e. justified, from the figures on pages 16 and 18. The conceptual view is chosen on the basis of the justification of the terminal command centers, cf. Sect. 5.2 on Page 23. We shall model physical mereology as attributes.[5]

5.5.2 Vessels 84

Physical Mereology:

75 Vessels are physically "connectable" to quay cranes of any terminal port.

```
type
75  Phys_V_Mer = QCI-set
value
75  attr_Phys_V_Mer: V → Phys_V_mer
```

85

[5]Editorial note: Names of physical and of conceptual mereologies have to be "streamlined". As now, they are a "mess"!

Conceptual Mereology:

76 Container vessels can potentially visit any container terminal port, hence have as [part of] their mereology, a set of terminal port command center identifiers.

type
76 V_Mer = MCCI-**set**
value
76 mereo_V: V → V_Mer
axiom
76 ∀ v:V • v ∈ *vs* ⇒ mereo_V(v) ⊆ *mcc_uis*

5.5.3 Quay Cranes 86

Physical Mereology: In modelling the physical mereology, though as an attribute, of quay cranes, we need the notion of quay positions.

77 Quay cranes are, at any time, positioned at one or more adjacent quay positions of an identified segment of such.

type
77 Phys_QC_Mereo = QPSId × QP*
value
77 attr_Phys_QC: QC → Phys_QC_Mereo

87

78 The quay positions, qcmereo = (qpsid,qpl):QCMereo, must be proper quay positions of the terminal,

79 that is, the segment identifier, qpsid, must be one of the terminal,

80 and the list, qpl, must be contiguously contained within the so identifier segment.

axiom ∀ tp:TP,
78 **let** q = obs_Q(tp), qcs = obs_QCs(obs_QCS(tp)) **in**
79 ∀ q:Q • q ∈ qcs ⇒
79 **let** (qpsid,qpl) = obs_Mereo(q), qps = attr_QPSs(q) **in**
79 qpsid ∈ **dom** qps
80 ∧ ∃ i,j:**Nat** • {i,j} ∈ **inds** qpl ∧ ⟨(qps(qpsi))[k]|i≤k≤j⟩ = qpl
78 **end end**

88

Conceptual Mereology: The conceptual mereology is simpler.

81 Quay cranes are conceptually related to the command center of the terminal in which they are located.

type
81 QC_Mer = MCCI
value
81 mereo_QC: QC → QC_Mer

5.5.4 Quay Trucks 89

Physical Mereology:

82 Quay trucks are physically "connectable" to quay and stack cranes.

type
82 Phys_QT_Mer = QCI-**set** × QCI-**set**
value
82 attr_Phys_QT_Mer: QT → Phys_QT_Mer

Conceptual Mereology:

83 Quay trucks are conceptually connected to the command center of the terminal port of which they are a part.

type
83 QT_Mer = MCCI
value
83 mereo_QT: QT → QT_Mer

5.5.5 Stack Cranes 90

Physical Mereology:

84 Terminal stack cranes are positioned to serve one or more terminal area bays, one or more quay trucks and one or more land trucks.

85 The terminal stack crane positions are indeed positions of their terminal

86 and no two of them share bays.

type
84 Phys_SCmereo = s_bis:BI-**set** × s_qtis:QTI-**set** × s_ltis:LTI-**set**
axiom
84 ∀ (bis,qtis,ltis):Phys_SCmereo•bis≠{} ∧ qtis≠{} ∧ ltis≠{}
value
84 Phys_SCmereo: SC → Phys_SCmereo
axiom
84 ∀ tp:TP •
84 **let** csa=obs_CSA(tp), bays=obs_BAYs(obs_BAYS(csa)), scs=obs_SCs(obs_SCS(tp)) **in**
85 ∀ sc:SC•sc ∈ scs ⇒ Phys_SCmereo(sc) ⊆ xtr_BIs(csa)
86 ∧ ∀ tp′,tp″:TP•{tc′,tc″}⊆tcs ∧ tc′≠tc″
86 ⇒ s_bis(Phys_SCmereo(tc′)) ∩ s_bis(Phys_SCmereo(tc″))={} **end**

91

Conceptual Mereology: The conceptual stack crane mereology is simple:

87 Each stack is conceptually related to the command center of the terminal at which it is located.

type
87 SC_Mer = MCCI
value
87 mereo_SC: SC → SC_Mer

5.5.6 Container Stowage Areas 92

Bays, Rows and Stacks: The following are some comments related to, but not defining a mereology for container stowage areas.

88 A bay of a container stowage area

  a. has either a predecessor
  b. or a successor,
  c. or both (and then distinct).
  d. No row cannot have neither a predecessor nor a successor.

89 A row of a bay has a predecessor and a successor, the first stack has no predecessor and the last stack has no successor.

90 A stack of a row has a predecessor and a successor, the first stack has no predecessor, and the last stack has no successor.

value
88 BAY_Mer: BAY → ({|'nil'|}|BI) × (BI|{|'nil'|})
89 ROW_Mer: ROW → ({|'nil'|}|RI) × (RI|{|'nil'|})
90 STK_Mer: STK → ({|'nil'|}|SI) × (SI|{|'nil'|})
axiom
88 ∀ csa:CSA • **let** bs = obs_BAYs(obs_BAYS(csa)) **in**
88 ∀ b:BAY • b ∈ bs ⇒
88 **let** (nb,nb′) = mereo_BAY(b) **in**
88 **case** (nb,nb′) **of**
88a. ('nil',bi) → bi ∈ xtr_BIs(csa),
88b. (bi,'nil') → bi ∈ xtr_BIs(csa),
88d. ('nil','nil') → **chaos**,
88c. (bi,bi′) → {bi,bi′} ⊆ xtr_BIs(csa) ∧ bi≠bi′
88 **end end end**
89 as for rows
90 as for stacks

5.5.7 Bay Mereology 94

Physical Vessel Bay Mereology:

91 A vessel bay is topologically related to the vessel on board of which it is placed and to the set of all quay cranes of all terminal ports.

type
91 Phys_VES_BAY_Mer = VI × QCI-**set**

Conceptual Vessel Bay Mereology:

92 A vessel bay is conceptually related to the set of all command centers of all terminal ports.

type
92 V_BAY_Mer = MCCI-**set**

95

Physical Terminal Port Bay (cum Stack) Mereology:

93 A terminal bay (cum stack) is topologically related to the stack cranes of a given terminal port and all land trucks.

type
93 Phys_STK_Mer = SCI-**set** × LTI-**set**

Conceptual Terminal Port Bay (cum Stack) Mereology:

94 A terminal port bay is conceptually related to the command center of its port.

type
94 T_BAY_Mer = MCCI

5.5.8 Land Trucks 96

Physical Mereology:

95 Land trucks are physically "connectable" to stack cranes – of any port.

type
95 Phys_LT_Mer = SCI-**set**
value
95 attr_Phys_LT_Mer: LT → Phys_LT_Mer

Conceptual Mereology:

96 Land trucks are conceptually connected to the command centers of any terminal port.

type
96 LT_Mer = MCCI-**set**
value
96 mereo_LT: LT → LT_Mer

5.5.9 Command Center 97

Command centers are basically conceptual quantities. Hence we can expect the physical mereology to be the conceptual mereology.

97 Command centers are physically and conceptually connected toall vessels, all cranes of the terminal port of the command center, all quay trucks of the terminal port of the command center, all stacks (i.e., bays) of the terminal port of the command center, and all land trucks, and all containers.

type
97 MCC_Mer = VI-**set**×QCI-**set**×QTI-**set**×SCI-**set**×BI-**set**×LTI-**set**×CI-**set**
value
97 mereo_MCC: MCC → MCC_Mer
axiom
97 ∀ tp:TP • tp ∈ tps •
97 **let** qcs:QC-**set** • qcs = obs_QCs(obs_QCS(tp)),
97 qts:QT-**set** • qts = obs_QTs(obs_QTS(tp)),
97 scs:SC-**set** • scs = obs_SCs(obs_SCS(tp)),
97 bs:iBAY-**set** • bs = obs_Bs(obs_BS(obs_CSA(tp))) **in**
97 **let** vis:VI_set • vis = {uid_VI(v)|v:V•v ∈ vs},
97 qcis:QCI_set • qcis = {uid_QCI(qc)|qc:QC•qc ∈ qcs},
97 qtis:QTI_set • qcis = {uid_QTI(qc)|qt:QT•qt ∈ qts},
97 scis:SCI-**set** • scis = {uid_SCI(sc)|sc:SC•sc ∈ scs},
97 bis:iBAY-**set** • bis = {uid_BI(b)|b:iBAY•b ∈ bs},
97 ltis:LTI-**set** • ltis = {uid_LTI(lt)|lt:LT•lt ∈ lts},
97 cis:SCI-**set** • cis = {uid_CI(c)|c:C•c ∈ cs} **in**
97 mereo_MCC(obs_MCC(tp)) = (vis,qcis,scis,sis,bis,ltis,cis) **end end**

5.5.10 Conceptual Mereology of Containers 99

The physical mereology of any container is modelled as a container attribute.

98 The conceptual mereology is modelled by containers being connected to all terminal command centers.

type
98 C_Mer = MCCI-**set**
value
98 mereo_C: C → C_Mer
axiom
98 ∀ c:C • mereo_C(c) = mcc_uis

5.6 Attributes 100

We refer to [13, Sect. 5.3].

5.6.1 States

By a state we shall mean one or more parts such that these parts have *dynamic* attributes, in our case typically *programmable* attributes.

5.6.2 Actions

Actions apply to states and yield possibly updated states and, usually, some result
101 values.

We shall in this section, Sect. 5.6, on attributes, outline a number of *simple* (usually called *primitive*) actions of states. These actions are invoked by some behaviours either at their own volition, or in response to events occurring in other behaviours. The action outcomes are simple enough, but calculations resulting in these outcomes are not. Together the totality of the actions performed by the terminal's monitoring & control of vessels, cranes, trucks and the container stowage area, reflect the complexity of stowage handling.

5.6.3 Attributes: Quays 102

99 Quays are segmented into one or more quay segments, qs:QS, each with a sequence of one or more crane positions, cp:CP.

100 Quay segements and

101 crane positions are further unspecified.

type
99 QPOS = QS $\times$ CP* **axiom** $\forall$ (_,cpl):QPOS•cpl$\neq\langle\rangle$
100 QS
101 CP

5.6.4 Attributes: Vessels 103

102 A vessel is

a. either at sea, at some *programmable* geographical location (longitude and latitude),

b. or in some *programmable* terminal port – designated by the identifier of its command center and its quay position.

103 We consider the “remainder” of the vessel state as a programmable attribute – which we do not further define. The remainder includes all information about all containers, their bay/row/stack/tier positions, their bill-of-ladings, etc.

104 There may be other vessel attributes.

104

```
type
102     V_Pos == AtSea | InPort
102a.   Longitude, Latitude
102a.   AtSea :: Longitude × Latitude
102b.   InPort :: MCCI × QPOS
103     VΣ
104     ...
value
102     attr_V_Pos: V → V_Pos
104     attr_VΣ: V → VΣ
104     attr_...: V → ...
axiom
102b.   ∀ mkInPort(ti):InPort • ti ∈ tp_uis
```

5.6.5 Attributes: Quay Cranes 105

105 At any one time a quay crane may *programmably* hold a container or may not. We model the container held by a crane by the container identifier.

106 At any one time a quay crane is *programmably* positioned in a quay position within a quay segment.

107 Quay cranes may have other attributes.

```
type
105   QCHold == mkNil('nil') | mkCon(ci:CI)
106   QCPos = QSId × QP
107   ...
value
105   attr_QCHold: QC → QCHold
106   attr_QCPos: QC → QCPos
107   ...
```

5.6.6 Attributes: Quay Trucks 106

108 At any one time a land truck may *programmably* hold a container or may not. We model the container held by a quay truck by the container identifier.

109 Quay trucks may have other attributes.

Note that we do not here model the position of quay trucks.

type
108 QTHold == mkNil('nil') | mkCon(ci:CI)
109 ...
value
108 attr_QTHold: QT → QTHold
109 ...

5.6.7 Attributes: Terminal Stack Cranes 107

110 At any one time a stack crane may *programmably* hold a container or may not. We model the container held by a crane by the container identifier.

111 Stack cranes are *programmably* positioned at a terminal bay.

112 Stack cranes may have other attributes.

type
110 SCHold == mkNil('nil') | mkCon(ci:CI)
111 SCPos = BI
111 ...
value
110 attr_SCHold: SC → SCHold
111 attr_SCPos: SC → SCPos
112 ...

5.6.8 Attributes: Container Stowage Areas 108

113 Bays of container storage areas *statically* have total order.

114 Rows of bays *statically* have total order.

115 Stacks of rows *statically* have total order.

We abstract orderings in two ways.

type
113 $\mathsf{BOm} = \mathsf{BI} \underset{m}{\rightarrow} \textbf{Nat}, \mathsf{BOl} = \mathsf{BI}^*$
114 $\mathsf{ROm} = \mathsf{RI} \underset{m}{\rightarrow} \textbf{Nat}, \mathsf{ROl} = \mathsf{RI}^*$
115 $\mathsf{SOm} = \mathsf{SI} \underset{m}{\rightarrow} \textbf{Nat}, \mathsf{SOl} = \mathsf{SI}^*$
axiom
113 $\forall$ bom:BOm•**rng** bom=$\{$1:**card dom** bom$\}$, $\forall$ bol:BOl•**inds** bol=$\{$1:**len** bol$\}$
114 $\forall$ rom:ROm•**rng** rom=$\{$1:**card dom** rom$\}$, $\forall$ rol:ROl•**inds** rol=$\{$1:**len** rol$\}$
115 $\forall$ som:SOm•**rng** som=$\{$1:**card dom** som$\}$, $\forall$ sol:SOl•**inds** sol=$\{$1:**len** sol$\}$
value
113 attr_BOm: CSA $\rightarrow$ BOm, attr_BOl: CSA $\rightarrow$ BOl
114 attr_ROm: BAY $\rightarrow$ ROm, attr_ROl: BAY $\rightarrow$ ROl
115 attr_SOm: ROW $\rightarrow$ SOm, attr_SOl: ROW $\rightarrow$ SOl

CSAs, BAYs, ROWs and STKs have (presently further) ***static*** *d*escriptions[6] and terminal and vessel container stowage areas have definite numbers 109

116 of bays,

117 and any one such bay a definite number of rows,

118 and any one such row a definite number of stacks,

119 and any one such stack a maximum loading of containers.

110

type
116 CASd
117 BAYd
118 ROWd
119 STKd
value
116 attr_CSAD: CSA $\rightarrow$ BI $\underset{m}{\rightarrow}$ CSAd
117 attr_BAYD: BAY $\rightarrow$ RI $\underset{m}{\rightarrow}$ BAYd
118 attr_ROWD: ROW $\rightarrow$ SI $\underset{m}{\rightarrow}$ ROWd
119 attr_STKD: STK $\rightarrow$ (**Nat** $\times$ STKd)

[6]Such descriptions include descriptions of for what kind of containers a container stowage area, a bay, a row and a stack is suitable: flammable, explosives, etc.

5.6.9 Attributes: Land Trucks 111

120 At any one time a land truck may *programmably* hold a container or may not. We model the container held by a land truck by the container identifier.

121 Land trucks also possess a further undefined *programmable* land truck state.

122 Land trucks may have other attributes.

Note that we do not here model the position of land trucks.

type
120 LTHold == mkNil('nil') | mkCon(ci:CI)
121 LTΣ
122 ...
value
120 attr_LTHold: LT $\rightarrow$ LTHold
121 attr_LTΣ: LT $\rightarrow$ LTΣ
122 ...

5.6.10 Attributes: Command Center 112

123 The `syntactic description`[7] of the spatial positions of quays, cranes and the container storage area of a terminal, TopLogDescr, is a *static* attribute.

124 The `syntactic description`[8] of the terminal state, i.e., the actual positions and deployment of vessels at quays, quay and stack cranes, quay and land trucks, and the actual container "contents" of these, TermΣDescr, is a *programmable* attribute.

type
123 TopLogDescr
124 MCCΣDescr
value
123 attr_TopLogDescr: MCC $\rightarrow$ TopLogDescr
124 attr_TermΣDescr: MCC $\rightarrow$ TermΣDescr

[7] A `syntactic description` describes something, i.e., has some `semantics`, from which it is, of course, different.

[8] The `syntactic description` of the terminal state is, of course, not that state, but only its description. The terminal state is the combined states of all cranes, trucks and the container storage area.

5.6.11 Attributes: Containers 113

125 A Bill-of-Lading[9] is a *static* container attribute. [10]

type
125 BoL
value
125 attr_BoL: C → BoL

114

126 At any one time a container is positioned either

a. in a stack on a vessel: at sea or in a terminal, or
b. on a quay crane in a terminal port, being either unloaded from or loaded onto a vessel, or
c. on a quay truck to or from a quay crane, i.e., from or to a stack crane, in a terminal port, or
d. on a stack crane in a terminal port, being either unloaded from a quay truck onto a terminal stack or loaded from a terminal stack onto a quay truck, or
e. on a stack in a terminal port, or
f. on a land truck, or
g. idle.

A container position is a *programmable* attribute.

127 There are other container attributes. For convenience we introduce an aggregate attribute: CAttrs for all attributes.

115

[9] https://en.wikipedia.org/wiki/Bill_of_lading: A bill of lading (sometimes abbreviated as B/L or BoL) is a document issued by a carrier (or their agent) to acknowledge receipt of cargo for shipment. In British English, the term relates to ship transport only, and in American English, to any type of transportation of goods. A bill of Lading must be transferable, and serves three main functions: it is a conclusive receipt, i.e. an acknowledgment that the goods have been loaded; and it contains or evidences the terms of the contract of carriage; and it serves as a document of title to the goods, subject to the nemo dat rule. Bills of lading are one of three crucial documents used in international trade to ensure that exporters receive payment and importers receive the merchandise. The other two documents are a policy of insurance and an invoice. Whereas a bill of lading is negotiable, both a policy and an invoice are assignable. In international trade outside of the USA, Bills of lading are distinct from waybills in that they are not negotiable and do not confer title. The **nemo dat rule:** that states that the purchase of a possession from someone who has no ownership right to it also denies the purchaser any ownership title.

[10] For waybills see https://en.wikipedia.org/wiki/Waybill: A waybill (UIC) is a document issued by a carrier giving details and instructions relating to the shipment of a consignment of goods. Typically it will show the names of the consignor and consignee, the point of origin of the consignment, its destination, and route. Most freight forwarders and trucking companies use an in-house waybill called a house bill. These typically contain "conditions of contract of carriage" terms on the back of the form. These terms cover limits to liability and other terms and conditions

```
type
126     CPos  == onV | onQC | onQT | onSC | onStk | onLT | Idle
126a.   onV    :: VI × BRSP × VPos
126a.   VPos  == AtSea | InTer
126a.   AtSea :: Geo
126a.   InTer :: QPSid × QP+
126b.   onQC  :: MCCI × QCI
126c.   onQT  :: MCCI × QTI
126d.   onSC  :: MCCI × SCI
126e.   onStk  :: MCCI × BRSP
126f.   onLT  :: MCCI × LTI
126g.   Idle   :: {|"idle"|}
127      CAttrs
value
126     attr_CPos: C → CPos
127     attr_CAttrs: C → CAttrs
```

6 Perdurants 116

We refer to [13, Sect. 7].

6.1 A Modelling Decision

In the *transcendental interpretation* of parts into behaviours we make the following modelling decisions: All atomic and all composite parts become separate behaviours. But there is a twist. Vessels and terminal stacks are now treated as "atomic" be-
117 haviours. Containers that up till now were parts of container stowage areas on vessels and in terminal stacks are not behaviours embedded in the behaviours of vessels and
118 terminal stacks, but are "factored" out as separate, atomic behaviours.

This modelling decision entails that container stowage areas, CSAs, of vessels and terminal stacks are modelled by replacing the [physical] containers of these CSAs with *virtual container stowage areas*, vir_CSAs. Where there "before" were containers there are now, instead, descriptions of these: their unique identifiers, their mereology, and their attributes.

6.2 Virtual Container Storage Areas 119

In our transition from endurants to perdurants we shall thus need a notion of container stowage areas which, for want of a better word, we shall call *virtual CSAs*. Instead of stacks embodying containers, they embody

128 container information: their unique identifier, mereology and attributes.

We must secure that no container is referenced more than once across the revised-model;

129 that is, that all ci:CIs are distinct.

120

type
5′ vir_CSA
11′ vir_BAY_s = vir_BAY-**set**, vir_BAY
12′ vir_ROW_s = vir_ROW-**set**, vir_ROW = vir_STK-**set**
13′ vir_STK = vir_STK-**set**, vir_STK
14′ vir_STK = CInfo*
128 CInfo = CI × CMereo × CAttrs
value
5′ attr_vir_CSA: TP → vir_CSA
11′ attr_vir_BAY_s: vir_CSA → vir_BAY_s, vir_BAY_s = vir_BAY-**set**, vir_BAY
11′ uid_vir_BAY: vir_BAY → BI
12′ attr_vir_ROW_s: vir_BAY → vir_ROW_s
axiom
129 [all CIs of all vir_CSAs are distinct]

6.3 Changes to The Parts Model 121

We revise the parts model of earlier:

type
2 STPs, TPs = TP-**set**, TP
3 SVs, Vs = V-**set**, V
value
2 obs_STPs: CLI → STPs, obs_TPs: STPs → TPs
3 obs_SVs: CLI → SVs, obs_Vs: SVs → Vs

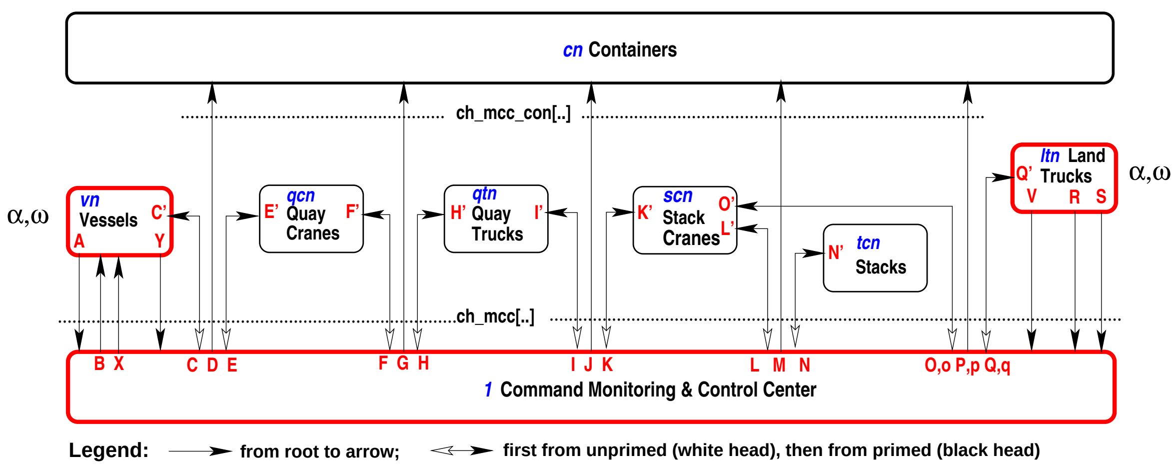
We treat the former CSAs of terminal ports as a composite, concrete part, vir_BAY_m consisting of a set of atomic virtual bays, vir_BAY. 122

type
11′ vir_BAY_s = vir_BAY-**set**, vir_BAY
value
5 obs_BAY_s: TP → vir_BAY_s
5 uid_BAY: vir_BAY → BI

And we treat the former CSAs of vessels as a programmable attribute of vessels:

attr_vir_CSA: V → vir_CSA

6.4 Basic Model Parts 123



124 **Fig. 3: The Container Terminal Behaviours**[11]

There are c_n container behaviours, where c_n is the number of all containers of the system we are modelling. For each terminal port there is 1 controller behaviour, v_n vessel behaviours, where v_n is the number of vessels visiting that terminal port, qc_n quay crane behaviours, where qc_n is the number of quay cranes of that terminal port, qt_n quay truck behaviours, where qt_n is the number of quay trucks of that terminal port, lt_n land truck behaviours, where lt_n is the number of land trucks (of that terminal port), and tb_n terminal stack behaviours, where ts_n is the number of terminal bays of
125 that terminal port.

The vessel, the land truck and the terminal monitoring & control [command] center behaviours are *pro-active:* At their own initiative (volition), they may decide to communicate with other behaviours. The crane, quay truck, stack and container behaviours are *passive:* They respond to interactions with other behaviours.

6.5 Actions, Events, Channels and Behaviours 126

We refer to [13, Sect. 7.1].

In building up to the behavioral analysis & description of the terminal container domain we first analyse the actions and events of that domain. These actions and
127 events are the building blocks of behaviours.

[11] The labeling **A, B, C, D, ..., X, Y** may seem arbitrary, but isn't!

Actions, to remind the reader, are explicitly performed by an actor, i.e., a behaviour, calculates some values and, usually, effect a state change.

Events *"occur to"* actors (behaviours), that is, are not initiated by these, but usually effect state changes.

6.6 Actions 128

We refer to [13, Sects. 7.1.5, 7.3.1].

The unloading of containers from and the loading of container onto container stowage areas are modelled by corresponding actions on virtual container stowage areas. Vessels, land trucks and terminal monitoring & control centers, i.e., command centers, are here modelled as the only entities that can *initiate* actions.

6.6.1 Command Center Actions 129

Motivating the Command Center Concept: We refer to the **[A,B,...,U]** labeled arrows of the figure on Page 44.

Imagine a terminal port. It has several vessels berthed along quays. It also has quay space, i.e., positions, for more vessels to berth. Berthed vessels are being serviced by several, perhaps many quay cranes. The totality of quay cranes are being serviced by [many more] quay trucks. The many quay trucks service several terminal bays, i.e., 130
stacks. Land trucks are arriving, attending *stacks* and leaving. Quite a "busy scene". So is the case for all container terminal ports. 131

The concept of a *monitoring & control, i.e., a command* **center**, is an abstract one; the figure on Page 44 does not show a part with a **... center** label. The *actions* of vessels and trucks, and the *events* of cranes, terminal stacks and trucks are either hap-hazard, no-one interferes, they somehow "just happen", or they are somehow co-ordinated.

Whether "free-wheeling" or "more-or-less coordinated" we can think of a *command center* as somehow *monitoring and controlling actions and events*. 132

Terminal *monitoring & control centers*, also interchangeably referred to as *command centers*, are thus where the logistics of container handling takes place.

You may think of this command center as receiving notices from vessels and land trucks as to their arrival and with information about their containers; thus building up awareness, i.e., a state, of the containers of all incoming and arrived vessels and land trucks, the layout of the terminal and the state of its container stowage area, the current whereabouts of vessels, cranes and trucks. Quite a formidable "state". 133

We shall therefore model the "comings" and "goings" of vessels, trucks, cranes and stacks as if they were monitored and controlled by a command center, In our modelling we are not assuming any form of efficiency; there is, as yet no notion of optimality, nor of freedom from mistakes and errors. Our modelling – along these

lines – is "hidden" in action pre- and post-conditions and thus allows for any degree
134 of internal non-determinism.

Calculate Next Transaction: The *core* action of the command center is `calc_nxt_transaction`. We shall define `calc_nxt_transaction` only by its signature and a pair of **pre/post** conditions. In this way we do not have to consider *efficiency, security, safety, etc.*, issues. These, i.e., the efficiency, security, safety, etc., issues can "always" be included in an *requirements engineering* implementation of `calc_nxt_transaction`. Basically the `calc_nxt_transaction` has to consider which of a
135 non-trivially large number of possible actions have to be invoked. They are listed in Items 131 to 137 below. The `calc_nxt_transaction` occurs in time, and occur repeatedly, endlessly, i.e., "ad-infinitum", At any time that `calc_nxt_transaction` is invoked the monitoring and control command center (mcc) is in some state. That state changes as the result of both monitoring actions and control actions. The `calc_nxt_transaction` therefore non-deterministically-internally chooses one among several
136 possible alternatives. If there is no alternative, then a **skip** action is performed.

The command center, mcc, models the following actions and events: **[A]** the `update` of the mcc state, mccσ, in response to the vessel action that inform the mcc of the vessel arrival.

130 The result of a `calc_nxt_transaction` is an transaction designator, MCCTrans and a state change. There are several alternative designators. We mention some:

131 **[B]:** the `calculation` of `vessel` `positions` for [their] arrivals;

137 132 **[CDE]:** the `calculation` of `vessel` to `quay` `crane` container transfers;

133 **[FGH]:** the `calculation` of `quay` `crane` to `quay` `truck` container transfers;

134 **[IJK]:** the `calculation` of `quay` `truck` to `stack` `crane` container transfers;

135 **[LMN]:** the `calculation` of `stack` `crane` to `stack` container transfers;

138 136 **[OPQ]:** the `calculation` of `land truck` to `stack` `crane` container transfers;

137 **[X]:** the `calculation` that stowage, for a given vessel, has completed; and

138 the calculation that there is no next transaction that can be commenced.

139 The signature of the `calc_nxt_transaction` involves the unique identifier, mereology, static and programmable attributes, i.e., the state of the command center, and indicates that a command center transaction results and a next state "entered".

140 For this, the perhaps most significant action of the entire container terminal port operation, we "skirt" the definition and leave to a pair pf **pre/post** conditions that of characterising the result and next state. 139

type
130 MCCTrans == QayPos | VSQC_Xfer | QCQT_Xfer | QTSC_Xfer
130 | SCSTK_Xfer | SCLT_Xfer | LT_Dept | VS_Dept | Skip
131 **[B]:** QuayPos :: VI × QPos
132 **[CDE]:** VSQC_Xfer :: VI × BRS × CI × QCI
133 **[FGH]:** QCQT_Xfer :: QCI × CI × QTI
134 **[IJK]:** QTSC_Xfer :: QTI × CI × SCI
135 **[LMN]:** SCSTK_Xfer :: SCI × CI × BRS
136 **[OPQ]:** SCLT_Xfer :: SCI × CI × LTI
137 **[X]:** VS_Dept :: VI
138 Skip :: nil
value
139 calc_nxt_transaction: MCCI×mereoMCC×statMCC→MCCΣ→MCCTrans×MCCΣ
139 calc_nxt_transaction(mcci,mccmereo,mmstat)(mccσ) **as** (mcctrans,mccσ')
140 **pre**: $\mathcal{P}_{calc\_nxt\_trans}$((mcci,mccmereo,mccstat)(mccσ))
140 **post**: $\mathcal{Q}_{calc\_nxt\_trans}$((mcci,mccmereo,mccstat)(mccσ))(mcctrans,mccσ')

140

The above mentioned actions are invoked by the command center in its endeavour to see containers moved from vessels to customers. A similar set of actions affording movement of containers customers to vessels, i.e., in the reverse direction: from land trucks to stack cranes, from stacks to quay trucks, from quay trucks to quay cranes, and from quay cranes to vessels, round off the full picture of all command center actions. 141

Command Center Action [A]: `update_mcc_from_vessel`:

141 Command centers

142 upon receiving arrival information, v_info, from arriving vessels, v_i, can update their state "accordingly".

143 We leave undefined the pre- and post-conditions.

value
141 update_mcc_from_vessel: VSMCC_MSG × MCC_Σ → MCC_Σ
142 update_mcc_from_vessel((vs_i,vir_csa,vs_info),mcc_σ) **as** mcc_σ'
143 **pre**: $P_{upd\_mcc\_f\_v}$((vs_i,vir_csa,vs_info),mcc_σ)
143 **post**: $Q_{upd\_mcc\_f\_v}$((vs_i,vir_csa,vs_info),mcc_σ)(mcc_σ')

142

Command Center Action [B]: `calc_ves_pos`:

144 Command centers

145 can calculate, q_pos, the quay segment and quay positions for an arriving vessel, v_i.

146 We leave undefined the pre- and post-conditions.

value
144 calc_ves_pos: MCCI$\times$MCC_mereo$\times$TopLog$\times$MCC$\Sigma\times$VI $\rightarrow$ (QSId$\times$QP*)$\times$MCCΣ
145 calc_ves_pos(mcc_i,mcc_mereo,toplog,mcc_σ,v_i) **as** (q_pos,mcc_σ')
146 **pre**: $P_{calc\_ves\_pos}$(mcc_i,mcc_mereo,toplog,mcc_σ,v_i)
146 **post**: $Q_{calc\_ves\_pos}$(mcc_i,mcc_mereo,toplog,mcc_σ,v_i)(q_pos,mcc_σ')

143

Command Center Action [C-D-E]: `calc_ves_qc`

147 The command center non-deterministically internally calculates

148 a pair of a triplet: the bay-row-stack coordinates, brs, from which a top container, supposedly ci, is to be removed by quay crane qci, and a next command center state reflecting that calculation (and that the identified quay crane is being so alerted).

149 We leave undefined the relevant pre- and post-conditions

144

value
147 calc_ves_qc: MCCΣ $\rightarrow$ (BRS$\times$CI$\times$QCI) $\times$ MCCΣ
148 calc_ves_qc(mccσ) **as** ((brs,ci,qci),mccσ')
149 **pre**: $\mathcal{P}_{calc\_ves\_qc}$(mcc$\sigma$)
149 **post**: $\mathcal{Q}_{calc\_ves\_qc}$(mcc$\sigma$)((brs,ci,qci),mcc$\sigma$')

145

Command Center Action [F-G-H]: `calc_qc_qt`

150 The command center non-deterministically internally

151 calculates a pair of a triplet: the identities of the quay crane from which and the quay truck to which the quay crane is to transfer a container, and an update command center state reflecting that calculation (and that the identified quay crane, container and truck are being so alerted).

152 We leave undefined the relevant pre- and post-conditions

value
150 calc_qc_qt: MCCΣ $\rightarrow$ (QCI$\times$CI$\times$QTI) $\times$ MCCΣ
151 calc_qc_qt(mccσ) **as** ((qci,ci,qti),mccσ')
152 **pre**: $\mathcal{P}_{calc\_qc\_qt}$(mccσ)
152 **post**: $\mathcal{Q}_{calc\_qc\_qt}$(mccσ)((qci,ci,qti),mccσ')

147

Command Center Action [I-J-K]: `calc_qt_sc`

153 The command center non-deterministically internally

154 calculates a pair of a triplet: the identities of a quay truck, a container, and a stack crane, and an update command center state reflecting that calculation (and that the identified quay truck, container and stack crane are being so alerted).

155 We leave undefined the relevant pre- and post-conditions

148

value
153 calc_qt_sc: MCCΣ $\rightarrow$ (QTI$\times$CI>SCI) $\times$ MCCΣ
154 calc_qt_sc(mccσ) **as** ((qti,ci,sci),mccσ')
155 **pre**: $\mathcal{P}_{calc\_qt\_sc}$(mccσ)
155 **post**: $\mathcal{Q}_{calc\_qt\_sc}$(mccσ)((qti,ci,sci),mccσ')

149

Command Center Action [L-M-N]: `calc_sc_stack`

156 The command center non-deterministically internally calculates a pair:

157 a triplet of the identities of a stack crane, a container and a terminal bay/row/stack triplet and a new state that reflects this action.

158 We leave undefined the relevant pre- and post-conditions

value
156 calc_sc_stack: MCCΣ $\rightarrow$ (SCI$\times$CI$\times$BRS)$\times$MCCΣ
157 calc_sc_stack(mccσ) **as** ((sci,ci,brs),mccσ')
158 **pre**: $\mathcal{P}_{calc\_sc\_stack}$(mcc$\sigma$)
158 **post**: $\mathcal{Q}_{calc\_sc\_stack}$(mcc$\sigma$)((sci,ci,brs),mcc$\sigma'$)

150

Command Center Action [N-M-L]: `calc_stack_sc`

159 The command center non-deterministically internally calculates a pair:

160 a triplet of a terminal bay/row/stack triplet and the identities of a container and a stack crane, and a new state that reflects this action.

161 We leave undefined the relevant pre- and post-conditions

value
159 calc_stack_sc: MCC$\Sigma \rightarrow$ (BRS$\times$CI$\times$SCI)$\times$MCCΣ
160 calc_stack_sc(mccσ) **as** ((brs,ci,sci),mccσ')
161 **pre**: $\mathcal{P}_{calc\_stack\_sc}$(mcc$\sigma$)
161 **post**: $\mathcal{Q}_{calc\_stack\_sc}$(mcc$\sigma$)((brs,ci,sci),mcc$\sigma'$)

151

Command Center Action [O-P-Q]: `calc_sc_lt`

162 The command center non-deterministically internally calculates a pair:

163 a triplet of the identities of a stack crane, a container and a land truck, and a new state that reflects this action.

164 We leave undefined the relevant pre- and post-conditions.

value
162 calc_sc_lt: MCC$\Sigma \rightarrow$ (BRS$\times$CI$\times$SCI)$\times$MCCΣ
163 calc_sc_lt(mccσ) **as** ((sci,ci,lti),mccσ')
164 **pre**: $\mathcal{P}_{calc\_sc\_lt}$(mccσ)
164 **post**: $\mathcal{Q}_{calc\_sc\_lt}$(mccσ)((sci,ci,lti),mccσ')

152

Command Center Action [Q-P-O]: `calc_lt_sc`

165 The command center non-deterministically internally calculates a pair:

166 a triplet of the identities of a land truck, a container and a stack crane, and a new state that reflects this action.

167 We leave undefined the relevant pre- and post-conditions.

value
165 calc_lt_sc: MCC$\Sigma \rightarrow$ (BRS$\times$CI$\times$SCI)$\times$MCCΣ
166 calc_lt_sc(mccσ) **as** ((lti,ci,sci),mccσ')
167 **pre**: $\mathcal{P}_{calc\_lt\_sc}$(mccσ)
167 **post**: $\mathcal{Q}_{calc\_lt\_sc}$(mccσ)((lti,ci,sci),mccσ')

153

Command Center: Further Observations Please observe the following: any terminal command center repeatedly and non-deterministically alternates between any and all of these actions. Observe further that: The intention of the pre- and post-conditions [Items 143, 146, 149, 152, 155, 158, 161, 167, and 164], express requirements to the command center states, $\mathsf{mcc}\sigma$:$\mathsf{mcc}\Sigma$, w.r.t. the information it must handle. Quite a complex state.

6.6.2 Container Storage Area Actions 154

We define two operations on virtual CSAs:

168 one of stacking (loading) a container, referred to by its unique identifier in a virtual CSA,

169 and one of unstacking (unloading) a container;

170 both operations involving bay/row/stack references.

155

type
170 BRS = BI × RI × SI
value
168 load_CI: vir_CSA × BRS × CI → vir_CSA
168 load_CI(vir_csa,(bi,ri,si),ci) **as** vir_csa′
168 **pre**: $\mathcal{P}_{load}$(vir_csa,(bi,ri,si),ci)
168 **post**: $\mathcal{Q}_{load}$(vir_csa,(bi,ri,si),ci)(vir_csa′)
169 unload_CI: vir_CSA × BRS $\tilde{\to}$ CI × vir_CSA
169 unload_CI(vir_csa,(bi,ri,si)) **as** (ci,vir_csa′)
169 **pre**: $\mathcal{P}_{unload}$(vir_csa,(bi,ri,si))
169 **post**: $\mathcal{Q}_{unload}$(vir_csa,(bi,ri,si))(ci,vir_csa′)

156

The Load Pre-/Post-Condtions

171 The virtual vir_CSA, i.e., vir_csa, must be well-formed;

172 the ci must not be embodied in that vir_csa; and

173 the bay/row/stack reference, (bi,ri,si) must be one of the [virtual] container stowage area.

value

168 $\mathcal{P}_{load}$(vir_csa,(bi,ri,si),ci) ≡
171 well_formed(vir_csa) cf. 25– 27 on Page 22
172 ∧ ci ∉ xtr_CIs(vir_csa) cf. 49 on Page 26
174 ∧ valid_BRS(bi,ri,si)(vir_csa)

174 valid_BRS: BRS → iCSA → **Bool**
174 valid_BRS(bi,ri,si)(vir_csa) ≡
174 bi∈ **dom**vir_csa∧ri∈ **dom**vir_csa(bi)∧si∈ **dom**(vir_csa(bi))(ri)

157

174 The resulting vir_CSA, i.e., vir_csa′, must have the same bay, row and stack identifications, and

175 except for the designated bay, row and stack, must be unchanged.

176 The designated "before", i.e., the stack before loading, must equal the tail of the "after", i.e., the loaded stack, and

177 the top of the "after" stack must equal the "input" argument container identifier.,

158

value

169 $\mathcal{Q}_{load}$(vir_csa,(bi,ri,si),ci)(vir_csa′) ≡
174 **dom** vir_csa = **dom** vir_csa′
174 ∧ ∀ bi′:BI•bi′∈ **dom** vir_csa(bi′)
174 ⇒ **dom** vir_csa(bi′)=**dom** vir_csa′(bi′)
174 ∧ ∀ ri′:RI•bi′∈ **dom** (vir_csa(bi′))()
174 ⇒ **dom** (vir_csa(bi′))(ri′)=(**dom** vir_csa′(bi′))(ri′)
174 ∧ ∀ si′:BI•bi′∈ **dom** vir_csa(bi′)
174 ⇒ **dom** ((vir_csa(bi′))(ri′))(si′)=**dom**((vir_csa′(bi′))(ri′))(si′)
175 ∧ ∀ bi′:BI•bi′ ∈ **dom** vir_csa \ {bi}
175 ⇒ vir_csa \ {bi}=vir_csa′ \ {bi}
175 ∧ ∀ ri′:RI•ri′ ∈ **dom** vir_csa(bi) \ {ri}
175 ⇒ (vir_csa(bi))(ri′)=(vir_csa′(bi))(ri′)
175 ∧ ∀ si′:SI•si′ ∈ **dom** (vir_csa)(ri′) \ {si}
175 ⇒ ((vir_csa)(bi′))(si′)=((vir_csa′)(bi′))(si′)
176 ∧ **tl**((vir_csa′)(bi′))(si′)=((vir_csa′)(bi′))(si′)
177 ∧ **hd**((vir_csa′)(bi′))(si′)=ci

159

The Unload Pre-/Post-Conditions

178 The virtual vir_csa, i.e., vir_csa,

179 must be wellformed; and

180 the bay/row/stack reference, (bi,ri,si) must be one of the [virtual] container stowage area.

value
178 $\mathcal{P}_{unload}$(vir_csa,(bi,ri,si)) ≡
179 well_formed(vir_csa)
180 ∧ valid_BRS(bi,ri,si)(vir_csa)

160

181

182

183

184

185

161

value
169 $\mathcal{Q}_{unload}$(vir_csa,(bi,ri,si))(ci,vir_csa′) ≡
181 **dom** vir_csa = **dom** vir_csa′
182 ∧ ∀ bi′:BI • bi′ ∈ **dom** vir_csa \ {bi}
182 ⇒ vir_csa \ {bi} = vir_csa′ \ {bi}
183 ∧ ∀ ri′:RI • ri′ ∈ **dom** vir_csa(bi) \ {ri}
183 ⇒ (vir_csa(bi))(ri′) = (vir_csa′(bi))(ri′)
184 ∧ ∀ si′:SI • si′ ∈ **dom** (vir_csa)(ri′) \ {si}
184 ⇒ ((vir_csa)(bi′))(si′) = ((vir_csa′)(bi′))(si′)
185 ∧ ((vir_csa′)(bi′))(si′)=**tl**((vir_csa′)(bi′))(si′)
185 ∧ **hd**((vir_csa)(bi′))(si′)=ci

6.6.3 Vessel Actions 162

Vessels (and land trucks) are in a sense, the primary movers in understanding the terminal container domain. Containers are, of course, at the very heart of this domain. But without container vessels (and land trucks) arriving at ports *nothing would happen!* So the actions of vessels are those of actively announcing their arrivals at and departures from ports, and participating, more passively, in the unloading and loading of containers. 163

Action [A]: `calc_next_port`:

186 Vessels can calculate, `calc_next_port`, the unique identifier, `mcc_i`, of that ports' monitoring & control center.

187 We do not further define the pre- and post-conditions of the `calc_next_port` action.

value
186 calc_next_port: VI$\times$VS_Mereo$\times$VS_Stat $\rightarrow$ vir_CSA$\times$VSΣ $\rightarrow$ MCCI$\times$VSΣ
186 calc_next_port(vs_i,vs_mereo,vs_stat)(vir_csa,vσ) ia (mcc_i,vsσ')
187 **pre**: $\mathcal{P}_{calc-next-port}$(vs$\sigma$,vs_mereo,vs_stat)
187 **post**: $\mathcal{Q}_{calc-next-port}$(vs$\sigma$,vs_mereo,vs_stat)(mcc_i,vsσ')

164

Vessel Action [B]: `calc_ves_msg`:

188 Vessels can calculate, `calc_ves_info`, the vessel information, vs_info:VS_Info, to be handed to the next ports' command center.

189 This information is combined with the vessel identifier and its virtual CSA,

190 We leave undefined the pre- and post-conditions over vessel states and vessel information.

165

type
188 VS_Info
189 VS_MCC_MSG :: VI$\times$vir_CSA$\times$VS_Info
value
188 calc_ves_msg: VI$\times$VMereo$\times$VStat $\rightarrow$ VS_Pos$\times$vir_CSA$\times$VSΣ $\rightarrow$ VS_MCC_MSG$\times$VSΣ
188 calc_ves_msg(vs_i,vs_mereo,vs_stat)(vpos,vir_csa,vsσ) **as** (vs_mcc_msg,vsσ')
190 **pre**: $\mathcal{P}_{calc\_ves\_mcc\_msg}$(vs_i,vs_mereo,vs_stat)(vpos,vir_csa,vsσ)
190 **post**: $\mathcal{Q}_{calc\_ves\_mcc\_msg}$(vs_i,vs_mereo,vs_stat)(vpos,vir_csa,vσ)(vs_mcc_msg,vsσ')

6.6.4 Land Truck Actions 166

Land trucks can initiate the following actions vis-a-vis a targeted terminal port command center: announce, to a terminal command center, its arrival with a container; announce, to a terminal command center, its readiness to haul a container. Land trucks furthermore interacts with stack cranes – as so directed by terminal command
167 centers.

Land Truck Action [R]: `calc_truck_delivery`:

191 Land trucks, upon approaching, from an outside, terminal ports, calculate

192 the identifier of the next port's command center and a next land truck state.

We do not define the

193 pre- and

194 post conditions of this calculation. 168

value

191 calc_truck_delivery: CI $\times$ TRUCKΣ $\rightarrow$ MCCI $\times$ LTΣ
192 calc_truck_delivery(ci,ltσ) **as** (mcci,ltσ')
193 **pre**: $\mathcal{P}_{calc\_truck\_deliv}$(ci,lt$\sigma$)
194 **post**: $\mathcal{Q}_{calc\_truck\_deliv}$(ci,lt$\sigma$)(mcci,lt$\sigma'$)

169

Land Truck Action [S]: `calc_truck_avail`:

195 Land trucks, when free, i.e., available for a next haul, calculate

196 the identifier of a suitable port's command center and a next land truck state.

We do not define the

197 pre- and

198 post conditions of this calculation.

value

195 calc_truck_avail: LTI $\times$ LTΣ $\rightarrow$ MCCI $\times$ LTΣ
196 calc_truck_avail(lti,ltσ) **as** (mcci,ltσ')
197 **pre**: $\mathcal{P}_{calc\_truck\_avail}$(lti,lt$\sigma$)
198 **post**: $\mathcal{Q}_{calc\_truck\_avail}$(lti,lt$\sigma$)(mcci,lt$\sigma'$)

6.7 Events 170

We refer to [13, Sect. 7.1.6 and 7.3.2]. Events occur to all entities. For reasons purely of presentation we separate events into active part initiation events and active part completion events. Active part initiation events are those events that signal the initi- 171
ation of actions. (Let **[Θ]** designate an action, then **[Θ']** designates the completion of that action.) Active part completion events are those events that signal the completion of actions. We do not show the lower case [**d, f, g, h, i, j, k, l, m, n, o**] in Fig. 3.

6.7.1 Active Part Initiation Events 172

Vessels:

199 [α_{vessel}] approaching terminal port;

200 [**A**] informing the command center, mcc, of a terminal port, of arrival;

201 [**B**] receiving from an mcc directions as to quay berth positions;

202 [**C**] receiving from an mcc, for each container to be unloaded or loaded, directions as to these unloads and ladings – and these actual unloads/ladings;

203 [**X**] receiving from an mcc directions of completion of stowage (no more unloads/loads);

204 [**Y**] informing the mcc of its departure from terminal port; or

205 [ω_{vessel}] leaving a terminal port.

173

Land Trucks:

206 [α_{land_truck}] approaching a terminal port;

207 [**W**] informing its mcc of its arrival;

208 [**V**] being directed, by an mcc, as to the stack (crane) of destination;

209 [**S**] the unloading, to a stack crane, of a container;

210 [**T**] the loading of a container from a stack crane;

211 [**R**] informing its mcc of its departure; or

212 [ω_{land_truck}] leaving a terminal port.

174

Containers: the transfers from

213 [**D**] vessel to quay crane;

214 [**d**] quay crane to vessel;

215 [**G**] quay crane to quay truck;

216 [**g**] quay truck to quay crane;

217 [**J**] quay truck to stack crane;

218 [**j**] stack crane to quay truck;

219 [**M**] stack crane to stack;

220 [**m**] stack to stack crane;

221 [**P**] stack crane to land truck; or from

222 [**p**] land truck to stack crane.

175

Quay Cranes: being informed, by the command center, mcc, of a container to be

223 [**E**] picked-up from a vessel;

224 [**e**] set-down on a vessel;

225 [**F**] set-down on a quay truck; or

226 [**f**] picked-up from a quay truck.

Quay Trucks: being informed, by the command center, mcc, of a container to be

227 [**H**] loaded from a quay crane;

228 [**h**] picked-up by a quay crane;

229 [**I**] picked-up by a stack crane; or

230 [**i**] loaded from a stack crane.

176

[Terminal] Stack Cranes: being informed, by the command center, mcc, of a container to be

231 [**K**] picked-up from a quay truck;

232 [**k**] loaded on to a quay truck;

233 [**L**] picked-up from a stack;

234 [**l**] loaded on to a stack;

235 [**O**] picked-up from a land truck; or

236 [**o**] loaded on to a land truck.

177

[Terminal Bay] Stacks: being informed, by the command center, mcc, of a container to be

237 [**N**] set-down, of a container, from a stack crane; or

238 [**n**] picked-up, of a container, by a stack crane.

These events, in most cases, prompt interaction with the terminal command center.

6.7.2 Active Part Completion Events: 178

We do not show, in Fig. 3, the **c', e', h', o', q', t'** events.

239 **[C']**

240 **[E']**

241 **[H']**

242 **[O']**

243 **[Q']**

244 **[T']**

6.8 Channels 179

We refer to [13, Sect. 7.2], and we refer to Sect. 5.2 and to Fig. 2 on Page 44.

6.8.1 Channel Declarations

There are channels between terminal port monitoring & control command center (mcci) and that command centers and that terminal port's

245 all the **containers** (ci), that might visit the terminal port; ch_mcc_con[mcci,ci][12];

246 **vessels** (vi) that might visit that port, ch_mcc[mcci,vi][13];

247 **quay cranes** (qci) of that port, ch_mcc[mcci,qci][14];

248 **quay trucks** (qti) of that port , ch_mcc[mcci,qti][15];

249 **stack cranes** (sci) of that port, ch_mcc[mcci,sci][16];

250 **stacks [bays]** (stki) of that port, ch_mcc[mcci,stki][17]; and

251 **land trucks** (lti) of, in principle, any port, ch_mcc[mcci,lti][18].

252 We shall define the concrete types of messages communicated by these channels subsequently (Sect. 6.8.2).

180

channel
245 {ch_mcc_con[mcci,ci]|mcci:MCCI,ci:CI•mcci∈mcc_uis∧ci∈c_uis}:MCC_Con_Cmd
246-251 {ch_mcc[mcci,ui]|mcci:MCCI,ui:(VI|QCI|QTI|SCI|STKI|LTI) • mcci∈mcc_uis∧ui∈uis}:MCC_Msg
type
252 MCC_Con_Msg, MCC_Msg

6.8.2 Channel Messages 181

We present a careful analysis description, for the channels declared above, of the rather rich variety of messages communicated over channels. All messages "goes to" (a few) or "comes from" (the rest) the command center. Messages from quay cranes, quay trucks, stack cranes, and land trucks – directed at the command center – are all
182 in response to the *events* of their being loaded or unloaded.

[12]cf. Item 98 on Page 35
[13]cf. Item 76 on Page 30
[14]cf. Item 81 on Page 31
[15]cf. Item 83 on Page 31
[16]cf. Item 87 on Page 32
[17]cf. Item 94 on Page 34
[18]cf. Item 96 on Page 34

A,B,X,Y,C′: Vessel Messages

253 There are a number **command center – vessel** and vice-versa messages:

a. **A**: Vessels announce their (forthcoming) arrival to the next destination terminal by sending such information, VSArrv, to its monitoring & control (also referred to as command) center, that enables it to handle those vessels' berthing, unloading and loading (of container stowage).[19]

b. **B**: The terminal command center informs such arriving vessels of their quay segment positions, VSQPos.

c. **X**: The terminal command center informs vessels of completion of stowage handling, VSComp.

d. **Y**: Vessels inform the terminal of their departure, VesDept.

183

type
253 MCC_Cmd == VSArrv|VSQPos|VSComp|VSDept|...
253a. **A**: VSArrv :: VI × vir_CSA
253b. **B**: VSQPos :: VI × (QSId × QP^+)
253c. **X**: VSComp :: MCCI × VI
253d. **Y**: VSDept :: MCCI × VI

184

C,D,E,E′: Vessel/Container/Quay Crane Messages

254 The terminal command center, at a time it so decides, "triggers" the simultaneous transitions, **C,D,E**, of

a. **C**: unloading (loading) from (to) a vessel stack position of a container (surrogate), VSQC_Xfer, QCVS_Xfer),

b. **D**: notifying the physical, i.e., the actual container that it is being unloaded (loaded), C_VStoQC (C_QCtoVS), and

c. **E**: loading (unloading) the container (surrogate) onto (from) a quay crane, VStoQC (QCtoVS).

255 **C′,E′**: The vessel and the quay crane, in response to their being unloaded, respectively loaded with a container "moves" that load, from its top vessel bay/row/stack position to the quay crane and notifies the terminal command center of the completion of that move, VSQC_Compl.

185

[19] What exactly that information is, i.e., any more concrete type model of Ves_Info cannot be given at this early stage in our development of *what a terminal is*.

type
253 MCC_Cmd == ... | VSQC_Xfer | QCVS_Xfer | C_VtoQC | C_QCtoV | VQC_Compl
254a. VSQC_Xfer, QCVS_Xfer :: VI × (BRS × CI) × QCI
254b. C_VStoQC, C_QCtoVS :: VI × CI × QCI
254c. VStoQC, QCtoVS :: VI × CI × QCI
255 VSQC_Compl == VS_UnLoad | VS_Load
255 VS_UnLoad, VS_Load :: VI × CI × QCI

186

F,G,H,H′: Quay Crane/Container/Quay Truck Messages

256 The terminal command center, at a time it so decides "triggers" the simultaneous transitions, **F,G,H**: QCtoQT, of

a. **F**: the removal of the container from the quay crane,
b. **G**: the notification of the physical container that it is now being transferred to a quay truck, and
c. **H**: the loading of that container to a quay truck.
d. **H′**: The quay truck, in response to it being loaded notifies the terminal command center of the completion of that move.

type
253 MCC_Cmd == ... | QCtoQT | ...
256 QCtoQT == UnloadCQC | NowConQT | LoadCQT | QCtoQTCompl
256a. UnloadCQC :: CI × QCI
256b. NowConQT :: CI × QTI
256c. LoadCQT :: CI × QTI
256d. QCtoQTCompl :: ...

187

I,J,K,K′: Quay Truck/Container/Stack Crane Messages

257 The terminal command center, at a time it so decides "triggers" the simultaneous transitions, **I,J,K**: QTtoSC, of

a. **I**: the removal of a container from a quay truck,
b. **J**: the notification of the physical container that it is now being transferred to a stack crane, and
c. **K**: the loading of that container to a stack crane.

258 **K′**: The stack crane, in response to it being loaded notifies the terminal command center of the completion of that move.

type
257 MCC_Cmd = ... | QTtoSC | ...
257 QTtoSC == UnLoadCQT | NowConSC | | QCQTCompl
257a. UnLoadCQT :: CI × QRI
257b. NowConSC :: CI × SCI
257c. LoadCSC :: CI × SCI
258 QCSCCompl :: ...

188

L,M,N,N′: Stack Crane/Container/Stack Messages

259 The terminal command center, at a time it so decides "triggers" the simultaneous transitions, **L,M,N**: SCtoStack, of

a. **L**: the unloading of the container from a stack crane;
b. **M**: the notification of the physical container that it is now being transferred to a stack, and
c. **N**: the loading of that container to a stack.

260 **N′**: The stack, in response to it being loaded, notifies the terminal command center of the completion of that move.

type
259 MCC_Cmd = ... | SCtoStack | ...
259 SCtoStack == UnLoadCSC | NowConSTK | LoadConSTK | SCStkCompl
259a. UnLoadCSC :: CI × SCI
259b. NowConSTK :: CI × BRS
259c. LoadConSTK :: CI × BRS
260 SCStkCompl :: ...

189

O,P,Q,Q′: Land Truck/Container/Stack Crane Messages

261 The terminal command center, at a time it so decides "triggers" the simultaneous transitions, **O,P,Q**: LTtoSC, of

a. **Q**: the unloading of the container from a land truck to a stack crane;
b. **P**: the notification of the physical container that it is now being transferred to a stack crane, and

c. **O**: the loading of that container to a stack crane.

d. **O′**: The stack crane, in response to it being loaded, notifies the terminal command center of the completion of that move.[20]

190

type
261 MCC_Cmd = ... | LTtoSC | ...
261 LTtoSC == UnLoadCLT | NowConSC | LoadConSC | LTtoSCCompl
261a. UnLoadCLT :: CI × LTI
261b. NowConSC :: CI × SCI
261c. LoadConSC :: CI × SCI
261d. LTtoSCCompl :: ...

191

R,S,T,U,Q,V: Land Truck Messages

262 These are the messages that are communicated either from land trucks to command centers or vice versa:

a. **R**: Land trucks, when approaching a terminal port, informs that port of its offer to deliver an identified container to stowage.

b. **S**: Land trucks, when approaching a terminal port, informs that port of its offer to accept (load) an identified container from stowage.

c. **T**: Land trucks, at a terminal, are informed by the terminal of the stack
192 crane at which to deliver (unload) an identified container.

d. **U**: Land trucks, at a terminal, are informed by the terminal of the stack crane from which to accept an identified container.

e. **Q**: Land trucks, at a termial, are informed by the terminal of the stack crane at which to unload (deliver) an identified container.

f. **q**: Land trucks, at a terminal, are informed by the terminal of the stack crane at which to load (accept) an identified container.

g. **V**: Land trucks, at a terminal, inform the terminal of their departure.

193

type
262 MCC_Cmd = ... | LTCmd | ...
262 LTCmd == LTDlvr | LTFtch | LTtoSC | LTfrSC | LTDept
262a. LTDlvr :: LTI × CI

[20]The **O′** event is "the same" as the **K′** event.

262b. LTFtch :: LTI × CI
262c. LTtoSC :: LTI × CI
262d. LTfrSC :: LTI × CI
262g. LTDept :: LTI

6.9 Behaviours 194

We refer to [13, Sects. 7.1.7, 7.3.3-4-5, and 7.4].

To every part of the domain we associate a behaviour. Parts are in space: there are the manifest parts, and there are the notion of their corresponding behaviours. Behaviours are in space and time. We model behaviours as processes defined in $\texttt{RSL}^+$. We cannot see these processes. We can, however, define their effects. 195

Parts may move in space: vessels, cranes, trucks and containers certainly do move in space; processes have no notion of spatial location. So we must "fake" the movements of movable parts. We do so as follows: We associate with containers the programmable attribute of location, as outlined in Items 126– 126g. on Page 41. We omit, for this model, the more explicit modelling of vessels, cranes and trucks but refer to their physical mereologies. 196

In the model of endurants, cf. Page 20, we modelled vessel and terminal container stowage areas as physically embodying containers, and we could move containers: push and pop them onto, respectively from bay stacks. This model must now, with containers being processes, be changed. The stacks, STACK, of container stowage areas, CAS, now embody unique container identifiers! We rename these stacks into cistack:CiSTACK

6.9.1 Terminal Command Center 197

The terminal command center is at the core of activities of a terminal port. We refer to the figure on Page 44. "Reading" that figure left-to-right illustrates the movements of containers from **[C-D-E]** vessels to quay cranes, **[F-G-H]** quay cranes to quay trucks, **[I-J-K]** quay trucks to stack cranes, **[L-M-N]** stack cranes to stacks, and from **[O-P-Q]** land truck to stack cranes. A similar "reading" of that figure from right-to-left 198
would illustrate the movements of containers from **[q-p-o]** stack cranes to land trucks; **[n-m-l]** stacks to stack cranes; **[k-j-i]** stack cranes to quay trucks; **[h-g-f]** quay trucks to quay cranes; and from **[e-d-c]** quay cranes to vessels. We have not show the **[c-d-e-f-g-h-i-j-k-l-m-n-o-p-q]** labels, but their points should be obvious (!). 199

The Command Center Behaviour: We distinguish between the command center behaviour offering to *monitor* primarily vessels and land trucks, secondarily cranes, quay cranes and stacks, and offering to *control* vessels, cranes, trucks and containers. 200

263 The signature of the command center behaviour is a triple of the command center identifier, the conceptual command center mereology and the static command center attributes (i.e., the topological description of the terminal); the programmable command center attributes (i.e., the command center state); and the input/output channels for the command center.

The command center behaviour non-deterministically (externallY) chooses between

264 either monitoring inputs from

265 or controlling (i.e., outputs to)

201 vessels, cranes, trucks, stacks and containers.

```
value
263  command_center:
263    mcci:MCCI×(vis,qcis,qtis,scis,bis,ltis,cis):MCC_Mer×MCC_Stat
263    → MCCΣ →
263      in,out { ch_mcc[ mcci,ui ]n
263                | mcci:MCCI,ui:(VI|QCI|QTI|SCI|BI|LTI)
263                  • ui∈vis∪qcis∪qtis∪scis∪bis∪ltis }
263      out { ch_mcc_con[ mcci,ci ] | ci:CI•ci ∈ cis }  Unit
263  command_center(mcci,(vis,qcis,scis,bis,ltis,cis),mcc_stat)(mccσ) ≡
264     monitoring(mcci,(vis,qcis,scis,bis,ltis,cis),mcc_stat)(mccσ)
263   []
265     control(mcci,(vis,qcis,scis,bis,ltis,cis),mcc_stat)(mccσ)
```

202

The Command Center Monitor Behaviours: The command center monitors the behaviours of vessels, cranes and trucks: **[A,Y′,C′,E′,F′,H′,I′,K′,L′,N′,O′,Q′]**. The input message thus received is typed:

```
type
    VCT_Info = ...
```

That information is used by the command center to update its state:

```
value
    update_MCCΣ: VCT_Infor → MCCΣ → MCCΣ
```

203 The definition of monitoring is simple.

266 The signature of the monitoring behaviour is the same as the command center behaviour.

267 The monitor non-deterministically externally ([]) offers to accept any input, vct_info, message from any vessel, any land truck and from local terminal port quay trucks and cranes.

268 That input, vct_info, enters the update of the command center state, from mccσ to mccσ'.

269 Whereupon the monitoring behaviour resumes being the command center behaviour with an updated state.

204

```
value
266  monitoring: mcci:MCCI × mis:MCC_Mereo × MCC_Stat
266     → MCCΣ
266     → in,out {chan_mcc[mcci,i] | i ∈ mis}  Unit
266  monitoring(mcci,mis,mcc_stat)(mccσ) ≡
267     let vct_info = [] { chan_mcc[mcci,i] ? | i ∈ mis } in
268     let mccσ′ = update_MCCΣ((vct_info,ui))(mccσ) in
269     command_center(mcci,mis,mcc_stat)(mccσ′) end end
```

205

The Command Center Control Behaviours:

270 The command center control behaviour has the same signature as the command center behaviour (formula Items 263).

271 In each iteration of the command center behaviour in which it chooses the control alternative it calculates[21] a next [output] transaction. This calculation is at the very core of the overall terminal port. We shall have more to say about this in Sect. 7.1 on Page 79.

Items, 272a.–272j. represent 10 alternative transactions. 206

272 They are "selected" by the **case** clause (Item 272).

So for each of these 10 alternatives there the command center offers a communication. For the **[CDE, FGH, IJK, LMN, OPQ, opq]** cases there is the same triple of concurrently synchronised events. For the **[B,T,X]** clauses there are only a single synchronisation effort. The command center events communicates:

[21] For `calc_nxt_transaction` see Items 130 – 140 on Page 47

a. **[B]** the quay positions to arriving vessels,
   the transfer of containers
b. **[CDE]** from vessel stacks to quay cranes,
207 c. **[FGH]** quay cranes to quay trucks,
d. **[IJK]** quay trucks to stack cranes,
e. **[LMN]** stack cranes to stacks,
f. **[OPQ]** stack cranes to land trucks, and
g. **[opq]** land trucks to stack cranes.
   We also illustrate
h. **[T]** the bays to which a land truck is to deliver, or fetch a container, and
i. **[X]** the "signing off" of a vessel by the command center.
j. For the case that the next transaction cannot be determined [at any given point in time] there is nothing to act upon.

208

273 After any of these alternatives the command center control behaviour resumes being the command center behaviour with the state updated from the next transaction calculation.

```
value
270   control: mcci:MCCI×(vis,qcis,qtis,scis,bis,ltis,cis):MCC_Mer×MCC_Stat → MCCΣ →
263       in,out {ch_mcc[ mcci,ui ]|mcci:MCCI,ui:(VI|QCI|QTI|SCI|BI|LTI)•ui∈vis∪qcis∪qtis∪scis∪bis∪ltis}
263       out { ch_mcc_con[ mcci,ci ] | ci:CI•ci ∈ cis }  Unit
270   control(mcci,(vis,qcis,scis,bis,ltis,cis),mcc_stat)(mccσ) ≡
271     let (mcc_trans,mccσ′) = calc_nxt_transaction(mcci,mcc_mereo,mcc_stat)(mccσ) in
272     case mcc_trans of
272a. [B]     mkVSQPos(vi,qp) → ch_mcc[ mcci,vi ] ! mkVSQPos(vi,qp),
272b. [CDE]   mkVSQC_Xfer(vi,(brs,ci),qci) →
272b. [C]              ch_mcc[ mcci,vi ] ! mkVes_UnLoad(ci,brs)
272b. [D]              ‖ ch_mcc_con[ mcci,ci ] ! mkNewPos(mcci,qci)
272b. [E]              ‖ ch_mcc[ mcci,qci ] ! mkQC_Load(ci),
272c. [FGH]   mkQCQT_Xfer(qci,ci,qti) →
272c. [F]              ch_mcc[ mcci,qci ] ! mkQC_UnLoad(ci)
272c. [G]              ‖ ch_mcc_con[ mcci,ci ] ! mkNewPos(mcci,qti)
272c. [H]              ‖ ch_mcc[ mcci,qti ] ! mkQT_Load(ci),
272d. [IJK]   mkQTSC_Xfer(qti,ci,sci) →
272d. [I]              ch_mcc[ mcci,qci ] ! mkQT_UnLoad(ci)
272d. [J]              ‖ ch_mcc_con[ mcci,ci ] ! mkNewPos(mcci,sci)
272d. [K]              ‖ ch_mcc[ mcci,qti ] ! mkSC_Load(ci),
272e. [LMN]   mkSCSTK_Xfer(brs,ci,sci,sti) →
272e. [L]              ch_mcc[ mcci,sci ] ! mkSC_UnLoad(ci)
272e. [M]              ‖ ch_mcc_con[ mcci,ci ] ! mkNewPos(mcci,brs)
```

```
272e. [N]              ∥ ch_mcc[ mcci,stki ] ! mkSTK_Load(ci,brs),
272f. [OPQ]  mkSCLT_Xfer(sci,ci,lti) →
272f. [O]              ch_mcc[ mcci,sci ] !  mkSC_UnLoad(ci)
272f. [P]              ∥ ch_mcc_con[ mcci,ci ] ! mkNewPos(mcci,lti)
272f. [Q]              ∥ ch_mcc[ mcci,lti ] ! mkLT_Load(ci),
272g. [opq]   mkLTSC_Xfer(sci,ci,lti) →
272g. [o]              ch_mcc[ mcci,sci ] ! mkSC_Load(ci)
272g. [p]              ∥ ch_mcc_con[ mcci,ci ] ! mkNewPos(mcci,cti)
272g. [q]              ∥ ch_mcc[ mcci,lti ] ! mkLT_UnLoad(ci),
272h. [T]     mkLT_Dept(lti) → ch_mcc[ mcci,lti ] ! LTDept(mcci,lti),
272i.  [Y]    mkVSComp(mcci,vi) → ch_mcc[ mcci,vi ] ! VSComp(mcci,vi),
272i.  [X]    mkVSDept(mcci,vi) → ch_mcc[ mcci,vi ] ! VSDept(mcci,vi),
272j.         _ → skip
272      end ; command_center(mcci,(vis,qcis,scis,bis,ltis,cis),mcc_stat)(mccσ′) end
```

6.9.2 Vessels

209

274 The signature of the **vessel** behaviour is a triple of the vessel identifier, the conceptual vessel mereology, the static vessel attributes, and the programmable vessel attributes. [We presently leave static attributes unspecified: ...]

Nondeterministically externally, [], the vessel decides between

275 **[A]** either approaching a port,

276 **[]** or [subsequently] arriving at that port,

or [subsequently] participating in the

277 **[]** unloading and

278 **[]** loading of containers of containers,

279 **[]** or [finally] departing from that port.

210

value

```
274 vessel: vi:VI×mccis:V_Mereo×V_Sta_Attrs → (V_Pos×vir_CSA×VΣ)
274      → in,out {ch_mcc[ mcci,vi ]|mcci:MCCI•mcci∈mccis}  Unit
274 vessel(vi,mccis,...)(vpos,vir_csa,vσ) ≡
275       port_approach(vi,mccis,...)(vpos,vir_csa,vσ)
276    [] port_arrival(vi,mccis,...)(vpos,vir_csa,vσ)
277    [] unload_container(vi,mccis,...)(vpos,vir_csa,vσ)
278    [] load_container(vi,mccis,...)(vpos,vir_csa,vσ)
279    [] port_departure(vi,mccis,...)(vpos,vir_csa,vσ)
```

211

Port Approach

280 The signature of port_approach behaviour is identical to that of vessel behaviour.

281 On approaching any port the vessel calculates the identity of that port's command center.

282 Then, with an updated state, it calculates the information to be handed over to the designated terminal –

283 **[A]** which is then communicated from the vessel to the command center;

284 whereupon the vessel resumes being a vessel albeit with a doubly updated state.

212

```
value
280 port_approach: vi:VI×vs_mer:VS_Mereo×VS_Stat→(VS_Pos×vir_CSA×VΣ)
280     → in,out {ch_mcc[mcci,vi]|mcci:MCCI•mcci∈mccis} Unit
280 port_approach(vi,vs_mer,vs_stat)(vpos,vir_csa,vσ) ≡
281     let (mcci,vσ′) = calc_next_port(vi,vs_mer,vs_stat)(vpos,vir_csa,vσ) in
282     let (mkVInfo(vi,vir_csa,vs_info),vσ″) = calc_ves_msg(vpos,vir_csa,vσ′) in
283     ch_mcc[mcci,vi] ! mkVS_Info(vi,vir_csa,vs_info) ;
284     vessel(vi,vs_mer,vs_stat)(vpos,vir_csa,vσ″) end end
```

213

Port Arrival

285 The signature of port_arrival behaviour is identical to that of vessel behaviour.

286 **[B]** Non-deterministically externally the vessel offers to accept a terminal port quay position from any terminal port's command center.

287 The vessel state is updated accordingly.

288 Whereupon the vessel resumes being a vessel albeit with a state updated with awareness of its quay position.

289 The vessel is ready to receive such quay position from any terminal port.

214

```
value
285 port_arrival: vi:VI×mccis:V_Mereo×V_Sta_Attrs → (V_Pos×vir_CSA×VΣ)
285     → in,out {ch_mcc[mcci,vi]|mcci:MCCI•mcci∈mccis} Unit
285 port_arrival(vi,mccis,...)(vpos,vir_csa,vσ) ≡
286     { let mkVSQPos(vi,(qs,cpl)) = ch_mcc[mcci,vi] ? in
287       let vσ′ = upd_ves_state(mcci,(qs,cpl))(vσ) in
288       vessel(vi,mccis,...)(mkInPort(mcci,mkVSQPos(qs,cpl)),vir_csa,vσ′) end end
289     | mcci:MCCI•mcci∈mccis }
```

215

Unloading of Containers

290 The signature of port_arrival behaviour is identical to that of vessel behaviour.

291 **[C]** The vessel offers to accept, ch_mcc_v[mcci,vi] ?, a directive from the command center of the terminal port at which it is berthed, to unload, mkUnload((bi,ri,si),ci). a container, identified by ci, at some container stowage area location ((bi,ri,si)).

292 The vessel unloads the container – identified by ci′.

293 If the unloaded container identifier is different from the expected **chaos** erupts!

294 The vessel state, vσ', is updated accordingly.

295 **[C']** "Some time has elapsed since the unload directive, modelling" the completion, from the point of view of the vessel, of the unload operation –

296 whereupon the command center is informed of this completion (**[']**).

297 The vessel resumes being the vessel in a state reflecting the unload.

216

```
value
290 unload_container: vi:VI × mccis:V_Mereo × V_Sta_Attrs → (V_Pos × iCSA × VΣ) →
290     in,out {ch_mcc[ mcci,vi ]|mcci:MCCI•mcci∈mccis}  Unit
290 unload_container(vi,mccis,...)(vpos,vir_csa,vσ) ≡
291     let mkVes_UnLoad(ci,(bi,ri,si)) = ch_mcc[ mcci,vi ] ? in
292     let (ci′,vir_csa′) = unload_CI((bi,ri,si),vir_csa) in
293     if ci′ ≠ ci then chaos end;
294     let vσ″ = unload_update_VΣ((bi,ri,si),ci)(vir_csa′) in
295     wait sometime ;
304     ch_mcc[ mcci,vi ] ! mkCompl(mkV_UnLoad((bi,ri,si),ci)) ;
305     vessel(vi,mccis,...)(vpos,vir_csa′,vσ″) end end end
```

217

Loading of Containers

298 The signature of load_container behaviour is identical to that of vessel behaviour.

299 **[c]** The vessel offers to accept, ch_mcc_v[mcci,vi] ?, a directive from the command center of the terminal port at which it is berthed, to load, mkLoad((bi,ri,si),ci). a container, identified by ci, at some container stowage area location ((bi,ri,si)).

300 The vessel (in co-operation with a quay crane, see later) then unloads the container – identified by ci.

301 The vessel state, vσ', is updated accordingly.

302 **[c']** "Some time has elapsed since the unload directive, modelling" the completion, from the point of view of the vessel, of the unload operation – whereupon the command center is informed of this completion (**[']**).

303 and the vessels resumes being the vessel in a state reflecting the load.

218

```
value
298 load_container: vi:VI×mccis:V_Mereo×V_Sta_Attrs → (V_Pos×vir_CSA×VΣ)
298     → in,out {ch_mcc[ mcci,vi ]|mcci:MCCI•mcci∈mccis}  Unit
298 load_container(vi,mccis,...)(vpos,vir_csa,vσ) ≡
299     let mkV_Load((bi,ri,si),ci) = ch_mcc[ mcci,vi ] ? in
300     let vir_csa′ = load_CI(vir_csa,(bi,ri,si),ci) in
301     let vσ′ = load_update_VΣ((bi,ri,si),ci) in
302     ch_mcc[ mcci,vi ] ! mkCompl(mkV_Load((bi,ri,si),ci)) ;
303     vessel(vi,mccis,...)(vpos,vir_csa′,vσ′) end end end
```

219

Port Departure

304 The signature of port_departure behaviour is identical to that of vessel behaviour.

305 **[Y]** At some time some command center informs a vessel that *stowage*, i.e., the unloading and loading of containers has ended.

306 Vessels update their states accordingly.

307 **[Y']** Vessels respond by informing the command center of their departure.

308 Whereupon vessels resume being vessels.

```
value
304 port_departure: vi:VI×mccis:V_Mereo×V_Sta_Attrs → (V_Pos×vir_CSA×VΣ)
304     → in,out {ch_mcc[ mcci,vi ]|mcci:MCCI•mcci∈mccis}  Unit
304 port_departure(vi,mccis,v_sta)(vpos,vir_csa,vσ) ≡
305     let mkStow_Compl(mcci,vi) [] { ch_mcc[ mcci,vi ] ? | mcci:MCCI•mcci∈mccis } in
306     let vσ′ = update_vessel_state(mkVes_Dept(mcci,vi))(vσ) in
307     ch_mcc[ mcci,vi ] ! mkVes_Dept(mcci,vi) ;
308     vessel(vi,mccis,v_sta)(vpos,vir_csa,vσ′) end end
```

• • •

The next three behaviours: quay_crane, quay_truck and stack_crane, are very similar. One substitutes, line-by-line, command center/quay crane, quay crane/quay truck, quay truck/stack crane et cetera !

6.9.3 Quay Cranes 221

309 The signature of the quay_crane behaviour is a triple of the quay crane identifier, the conceptual quay crane mereology, the static quay crane attributes, the programmable quay crane attributes – and the 'command center'/'quay crane' channel.

310 The quay crane offers, non-deterministially externally, to

311 either, **[E]**, accept a directive of a *'container transfer from vessel to quay crane'*.

  a. The quay crane then resumes being a quay crane now holding (a surrogate of) the transferred container.

222

312 or, **[F]** accept a directive of a transfer *'container from quay crane to quay truck'*.

  a. The quay crane then resumes being a quay crane now holding (a surrogate of) the transferred container.

```
value
309    quay_crane: qci:QCI × mcci:QC_Mer × QC_Sta → (QCHold×QCPos)
309        → ch_mcc[ mcci,qci ]  Unit
309    quay_crane(qci,mcci,qc_sta)(qchold,qcpos) ≡
311        let mkVSQC(ci) = ch_mcc[ mcci,qci ] ? in
311a.          quay_crane(qci,mcci,qc_sta)(mkCon(ci),qcpos) end
310      []
311        let mkQCVS(ci) = ch_mcc[ mcci,qci ] ? in
312a.          quay_crane(qci,mcci,qc_sta)(mkCon(ci),qcpos) end
```


6.9.4 Quay Trucks 223

313 The signature of the quay_truck behaviour is a triple of the quay truck identifier, the conceptual quay truck mereology, the static quay truck attributes, the programmable quay truck attributes – and the 'command center'/'quay truck' channel.

314 The quay truck offers, non-deterministially externally, to

315 either, **[H]**, accept a directive of a *'container transfer from quay crane to quay truck'*.

  a. The quay truck then resumes being a quay truck now holding (a surrogate of) the transferred container.

224

316 or, **[I]**, accept a directive of a *'container transfer from quay truck to quay crane'*.

  a. The quay truck then resumes being a quay truck now holding (a surrogate of) the transferred container.

```
value
313   quay_truck: qti:QTI × mcci:QC_Mer × QT_Sta → (QTHold×QTPos)
313       → ch_mcc[mcci,qci]  Unit
313   quay_truck(qti,mcci,qt_sta)(qthold,qtpos) ≡
315       let mkQCQT(ci) = ch_mcc[mcci,qti] ? in
315a.       quay_crane(qti,mcci,qc_sta)(mkCon(ci),qcpos) end
314     []
316       let mkQTQC(ci) = ch_mcc[mcci,qti] ? in
316a.       quay_crane(qti,mcci,qc_sta)(mkCon(ci),qcpos) end
```

6.9.5 Stack Crane 225

317 The signature of the stack_crane behaviour is a triple of the stack crane stack crane identifier, the conceptual mereology, the static stack crane attributes, the programmable stack crane attributes – and the 'command center'/'stack crane' channel.

318 The stack crane offers, non-deterministially externally, to

319 either, **[K]**, accept a directive of a *'container transfer from quay truck to stack crane'*.

a. The stack crane then resumes being a stack crane now holding (a surrogate of) the transferred container.

320 or, **[L]**, accept a directive of a *'container transfer from stack crane to quay truck'*.

a. The stack crane then resumes being a stack crane now holding (a surrogate of) the transferred container.

value
```
317   stack_crane: sci:SCI × mcci:SC_Mer × SC_Sta → (SCHold×SCPos)
317       → ch_mcc[ mcci,sci ]  Unit
317   stack_crane(sci,mcci,sc_sta)(schold,scpos) ≡
319       let mkQTSC(ci) = ch_mcc[ mcci,sci ] ? in
319a.        stack_crane(sci,mcci,sc_sta)(mkCon(ci),scpos) end
318     []
320       let mkSCQT(ci) = ch_mcc[ mcci,sci ] ? in
320a.        stack_crane(sci,mcci,sc_sta)(mkCon(ci),scpos) end
```

6.9.6 Stacks 227

The stack behaviour is very much like the unload_container container behaviour of the vessel, cf. Items 290 – 294 on Page 69.

321 The signature of the stack behaviour is a triple of the stack, i.e. terminal port bay identifier, the conceptual bay mereology, the static bay attributes, the programmable bay attributes and the 'command center'/'stack' channel. 228

322 The stack offers, **[N]**, to accept directive of a *'container transfer from stack crane to stack'*.

a. The stack behaviour loads the container, identified by ci′, to the bay/row/stack top, identified by (bi,ri,si).
b. If the unloaded container identifier is different from the expected **chaos** erupts!
c. The stack state, bay′, is updated accordingly.
d. **[N']** "Some time has elapsed since the load directive, modelling" the completion, from the point of view of the vessel, of the unload operation –
e. whereupon the command center is informed of this completion (**[']**).
f. The stack then resumes being a stack now holding (a surrogate of) the transferred container.

229

```
value
321    stack: tbi:TBI×mcci:STK_Mer×Stk_Sta_Attrs → (iCSA × Stk_Dir) →
321      in,out {ch_mcc[mcci,vi]|mcci:MCCI•mcci∈mccis} Unit
321    stack(tbi,mcci,stk_sta)(bay,dir) ≡
322      let mkUnload((bi,ri,si),ci) = ch_mcc[mcci,tbi] ? in
322a.    let (ci′,bay′) = unload_CI((bi,ri,si),bay) in
322b.    if ci′ ≠ ci then chaos end ;
322c.    let bay″ = unload_update_BAY((bi,ri,si),ci)(bay′) in
322d.    wait sometime ;
322e.    ch_mcc[mcci,tbi] ! mkCompl(mkUnload((bi,ri,si),ci)) ;
322f.    stack(tbi,mcci,stk_sta)(bay″,dir) end end end
```

6.9.7 Land Trucks 230

323 The signature of the land_truck behaviour is a triple of the land truck identifier, the conceptual land truck mereology and the static land truck attributes, and the programmable land truck attributes.

324 **R**

a. The land truck calculates the identifier of the next port's command center

b. and communicates with this center as to its intent to deliver a container identified by ci,

c. whereupon the land truck resumes being that.

231

325 **T**

a. The command center informs the land truck of the bay ('stack'), brs, at which to deliver the container,

b. whereupon the land truck resumes being that.

326 **Q**

a. The command center informs the land truck of the delivery of a container from a stack crane,

b. ...,

c. whereupon the land truck resumes being that.

232

327 **V**

a. The land truck informs the command center of its intent to depart from the terminal port,

b. whereupon the land truck resumes by eaving the terminal port.

233

```
value
323  land_truck:
323
323  land_truck(lti,lt_mer,lt_sta)(lt_pos,lt_hold) ≡
324        next_port(lti,lt_mer,lt_sta)(lt_pos,lt_hold)
325     [] stack_location(lti,lt_mer,lt_sta)(lt_pos,lt_hold)
326     [] stack_crane_to_land_truck(lti,lt_mer,lt_sta)(lt_pos,lt_hold)
327     [] land_truck_departure(lti,lt_mer,lt_sta)(lt_pos,lt_hold)
```

```
value
324   next_port(lti,lt_mer,lt_sta)(...,mkHold(ci,cσ)) ≡
324a.      let mcci = calc_truck_delivery(ci,cσ) in
324b.      ch_mcc[ mcci,lti ] ! mkDlvr(ci,cσ) ;
324c.      land_truck(lti,lt_mer,lt_sta)(...,...) end ???
```

```
value
325   stack_location(lti,lt_mer,lt_sta)(...,mkHold(ci,cσ)) ≡
325a.      let  mkLT_Pos(mcci,brs) = { ch_mcc[ mcci,lti ] ? | mcci:MCCI • mcci ∈ mcc_uis }
325b.      land_truck(lti,lt_mer,lt_sta)(...,lt_hold) end ???
```

```
value
326   stack_crane_to_land_truck(lti,lt_mer,lt_sta)(lt_pos,lt_hold) ≡
326a.
326b.
```

```
value
327   land_truck_departure(lti,lt_mer,lt_sta)(...,...) ???
327a.      ch_mcc[ mcci,lti ] ! mkDept(lti) ;
327b.      land_truck(lti,lt_mer,lt_sta)(...,...)  ???
```


6.9.8 Containers 234

In RSL, as with all formal specification languages one cannot “move” values. So we model containers of vessels and of terminal port stacks as separate behaviours and replace their “values”, C in vessel and terminal port stacks by their unique identifications, CI.

328 The signature of the container behaviour is simple: the container identifier, its
235 mereology, its static values, its position and state[22], and its input channels.

329 **[D,G,J,M,P]** The container is here simplified to just, at any moment, accepting a new position from any terminal ports command center;

330 whereupon the container resumes being that with that new position.

value
```
328  container: ci:CI×mcci_uis:C_Mer×C_Stat → (CPos×CΣ)
328     → in { ch_mcc_con[ mcci,ci ]
328           | mcci:MCCI • mcci∈mcci_uis }  Unit
328  container(ci,mcci_uis,...)(pos,sσ) ≡
329     let mkNewPos(p) = { ch_mcc_con[ mcci,ci ] ?
329                        | mcci:MCCI•mcci∈mcci_uis } in
330     container(ci,mcci_uis,...)(mkNewPos(p),sσ) end
```

6.10 Initial System 236

6.10.1 The Distributed System

We remind ourselves that the container line industry includes a set of vessels, a set of land trucks, a set of containers and a set of terminal ports. We rely on the states
237 expounded in Sect. 5.4.1’s Items 50 on Page 26 – 54 on Page 26.

331 The signature of τ_initial_system is that of a function from an endurant endurant container line industry to its perdurant behaviour, i.e., **Unit**.

This behaviour is expressed as

332 the distributed composition of all vessel behaviours in parallel with

333 the distributed composition of all land truck behaviours in parallel with

334 the distributed composition of all container behaviours in parallel with

335 the distributed composition of all terminal port behaviours.

value
332 τ_initial_system: CLI $\rightarrow$ **Unit**
332 τ_initial_system(cli) $\equiv$
332 $\|\{$ τ_vessel(v) | v:V • v $\in vs$ $\}$
333 $\|$ $\|\{$ τ_land_truck(lt) | lt:LT • lt $\in lts$ $\}$
334 $\|$ $\|\{$ τ_container(c) | c:CON • c $\in cs$ $\}$
335 $\|$ $\|\{$ τ_terminal_port(tp) | tp:TP • tp $\in tps$ $\}$

6.10.2 Initial Vessels 239

336 The signature of the i_vessel `transalation function` is simple: a τ`ranslator` from endurant vessel `parts` v to perdurant vessel `behaviours`, i.e., **Unit**.

337 The transcendental deduction then consists of obtaining the proper arguments for the vessel behaviour –

338 and invoking that behaviour.

value
336 τ_vessel: V $\rightarrow$ **Unit**
336 τ_vessel(v) $\equiv$
337 **let** v_ui = uid_V(v), v_mer = mereo_V(v),
337 v_sta = attr_V_Sta(v), v_pos = attr_V_Pos(v),
337 v_csa = attr_iCSA(v), vσ = attr_VΣ(v) **in**
338 vessel(v_ui,v_mer,v_sta)(v_pos,v_csa,vσ) **end**

6.10.3 Initial Land Trucks 240

Similarly:

τ_land_truck: LT $\rightarrow$ **Unit**
τ_land_truck(lt) $\equiv$
let lt_ui = uid_LT(lt), lt_mer = mereo_LT(lt),
lt_sta = attr_LT_Sta(lt), lt_pos = attr_LT_Pos(lt),
lt_hold = attr_LT_Hold(v), ltσ = attr_LTΣ(lt) **in**
vessel(lt_ui,lt_mer,lt_sta)(lt_pos,lt_hold,ltσ) **end**

[22] As for state: I need to update the container attribute section, Sect. 5.6.11 on Page 41 to reflect a state (for example: the component contents of a container)

6.10.4 Initial Containers 241

Similarly:

```
τ_container: CON → Unit
τ_container(con) ≡
      let c_ui = uid_CON(con), c_mer = mereo_CON(con),
          c_sta = attr_C_Sta(con), c_pos = attr_C_Pos(con),
          cσ = attr_CONΣ(lt) in
      container(c_ui,c_mer,c_sta)(c_pos,cσ) end
```

6.10.5 Initial Terminal Ports 242

Terminal ports consists of a set of quay cranes, a set of quay trucks a set of stack
243 cranes, and a set of stacks. They translate accordingly:

```
τ_terminal_port: TP → Unit
τ_terminal_port(tp) ≡
      let qcs = obs_QCs(obs_QCS(tp)),
            qts = obs_QTs(obs_QTS(tp)),
            scs = obs_SCs(obs_SCS(tp)),
            stks = obs_STKs(obs_STKS(tp)) in
   ‖ { τ_quay_crane(qc) | qc:QC • qc ∈ qcs } ‖
   ‖ { τ_quay_truck(qt) | qt:QT • qt ∈ qts } ‖
   ‖ { τ_stack_crane(sc) | sc:SC • sc ∈ scs } ‖
   ‖ { τ_stack(stk) | stk:STK • stk ∈ stks } end
```

6.10.6 Initial Quay Cranes 244

```
τ_quay_crane: QC → Unit
τ_(qc) ≡
      let qc_ui = uid_QC(qc), qc_mer = mereo_QC(qc),
          qc_sta = attr_QC_Sta(qc), qc_pos = attr_QC_Pos(qc),
          qcσ = attr_QCΣ(qc) in
      quay_crane(qc_ui,qc_mer,qc_sta)(qc_pos,qcσ) end
```

6.10.7 Initial Quay Trucks 245

```
τ_quay_truck: QT → Unit
τ_quay_truck(qt) ≡
    let qt_ui = uid_QT(qt), qt_mer = mereo_QT(qt),
        qt_sta = attr_QT_Sta(qt), qt_pos = attr_QT_Pos(qt),
        qtσ = attr_QTΣ(qt) in
    quay_truck(qt_ui,qt_mer,qt_sta)(qt_pos,qtσ) end
```

6.10.8 Initial Stack Cranes 246

```
τ_stack_crane: SC → Unit
τ_stack_crane(sc) ≡
    let sc_ui = uid_SC(sc), sc_mer = mereo_SC(sc),
        sc_sta = attr_SC_Sta(sc), sc_pos = attr_SC_Pos(sc),
        scσ = attr_SCΣ(sc) in
    container(sc_ui,sc_mer,sc_sta)(sc_pos,scσ) end
```

6.10.9 Initial Stacks 247

```
τ_stack: STK → Unit
τ_stack(stk) ≡
    let stk_ui = uid_STK(stk), stk_mer = mereo_STK(stk),
        stk_sta = attr_STK_Sta(stk),
        stkσ = attr_STKΣ(stk) in
    stack(stk_ui,stk_mer,stk_sta)(stkσ) end
```

7 Conclusion 248

TO BE WRITTEN

7.1 An Interpreation of the Behavioural Description 249

TO BE WRITTEN

7.2 What Has Been Done 250

TO BE WRITTEN

7.3 What To Do Next 251

TO BE WRITTEN

7.4 Acknowledgements 252

This report was begun when I was first invited to lecture, for three weeks in November 2018, at ECNU[23], Shanghai, China. For this and for my actual stay at ECNU, I gratefully acknowledge Profs. He JiFeng, Zhu HuiBiao, Wang XiaoLing and Min
253 Zhang. I chose at the time of the invitation to lead the course students through a major, non-trivial example. Since Shanghai is also one of the major container shipping ports of the world, and since the Danish company Maersk, through its subsidiary, APMTerminals, operates a major container terminal port, I decided on the subject fo this experimental report. I gratefully acknowledge the support the ECNU course received from APMTerminals, through its staff, Messrs Henry Bai and Niels Roed.

8 Bibliography 254

8.1 References

[1] Dines Bjørner. A Container Line Industry Domain. Techn. report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, June 2007. [24].

[2] Bureau Export. A-Z Dictionary of Export, Trade and Shipping Terms. www.export-bureau.com/trade_shipping_terms/dictionary.html, 2007.

[3] International Labour Organisation. Portworker Development Programme: PDP Units. Enumerate PDP units. , April 2002.

[4] K.V. Ramani. An interactive simulation model for the logistics planning of container operations in seaports. *SIMULATION*, 66(5):291–300, 1996.

[5] Mordecai Avriel, Michal Penn, Naomi Shpirer, and Smadar Witteboon. Stowage planning for container ships to reduce the number of shifts. *Annals of Operations Research*, 76(9):55–71, January 1998.

[6] I.D. Wilson and P.A. Roach. Container stowage planning: a methodology for generating computerised solutions. *Journal of the Operational Research Society*, 51(11):1248–1255, 1 November 2000. Palgrave Macmillan. University of Glamorgan, UK.

[23] ECNU: East China Normal University

[24] http://www2.imm.dtu.dk/˜db/container-paper.pdf

[7] Mordecai Avriel, Michal Penn, and Naomi Shpirer. Container ship stowage problem: complexity and connection to the coloring of circle graphs. *Discrete Applied Mathematics*, 103(1–3):271–279, 15 July 2000. Faculty of Industrial Engineering and Management, Technion, Israel Institute of Technology, Haifa 3200, Israel.

[8] I.D. Wilson, P.A. Roach, and J. A. Ware. Container stowage pre-planning: using search to generate solutions, a case study. *Knowledge-Based Systems*, 14(3–4):137–145, June 2001.

[9] Opher Dubrovsky, Gregory Levitin, and Michal Penn. A genetic algorithm with a compact solution encoding for the container ship stowage problem. *Journal of Heuristics*, 8(6):585–599, November 2002.

[10] Akio Imai, Kazuya Sasaki, Etsuko Nishimura, and Stratos Papadimitriou. Multi-objective simultaneous stowage and load planning for a container ship with container rehandle in yard stacks. *European Journal of Operational Research*, 171:373–389, 2006.

[11] Dirk Steenken, Stefan Voß, and Robert Stahlbock. Container terminal operation and operations research - a classification and literature review. *OR Spectrum*, 26(1):3–49, January 2004.

[12] Bram Borgman, Eelco van Asperen, and Rommert Dekker. Online rules for container stacking. *OR Spectrum*, 32:687–716, 19 March 2010.

[13] Dines Bjørner. A Domain Analysis & Description Method – Principles, Techniques and Modelling Languages. Paper submitted for publication, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, May 16 2018. [25].

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

[25] http://www.imm.dtu.dk/~dibj/2018/tosem/Bjorner-TOSEM.pdf

9 Summary of Internal Types 255

9.1 Unique Identifiers

32 pp.24 QCI
33 pp.24 QTI
34 pp.24 SCI
35 pp.24 TBI
36 pp.24 LTI
37 pp.24 MCCI
38 pp.24 CI
39 pp.24 BI
40 pp.24 RI
41 pp.24 SI

9.2 Mereologies

value
76 pp.30 V_Mer = MCCI-**set**
81 pp.31 QC_Mer = MCCI
83 pp.31 QT_Mer = MCCI
87 pp.32 SC_Mer = MCCI
94 pp.34 T_BAY_Mer = MCCI
96 pp.34 LT_Mer = MCCI-**set**
97 pp.35 MCC_Mer =
97 pp.35 VI-**set**×QCI-**set**×QTI-**set**×
97 pp.35 SCI-**set**×TBI-**set**×LTI-**set**×CI-**set**
98 pp.35 C_Mer = MCCI-**set**

9.3 Attributes

type Vessels:
102 pp.36 V_Pos == AtSea | InPort
102a. pp.36 Longitude, Latitude
102a. pp.36 AtSea :: Longitude × Latitude
102b. pp.37 InPort :: MCCI × QPOS
103 pp.37 VΣ

type Quay Cranes:
105 pp.37 QCHold == mkNil(′nil′)
105 pp.37 | mkCon(ci:CI)
106 pp.37 QCPos = QSId × QP

type Quay Trucks:
108 pp.38 QTHold == mkNil(′nil′)
108 pp.38 | mkCon(ci:CI)

type Stack Cranes:
110 pp.38 SCHold == mkNil(′nil′)
110 pp.38 | mkCon(ci:CI)
111 pp.38 SCPos = BI

type Terminal [i.e., Bay] Stacks
113 pp.38 BOm = BI $\overrightarrow{m}$ **Nat**, BOl = BI*
114 pp.38 ROm = RI $\overrightarrow{m}$ **Nat**, ROl = RI*
115 pp.38 SOm = SI $\overrightarrow{m}$ **Nat**, SOl = SI*

type Land Trucks:
120 pp.40 LTHold == mkNil(′nil′)
120 pp.40 | mkCon(ci:CI)
121 pp.40 LTΣ

type Command Centers:
123 pp.40 TopLogDescr
124 pp.40 MCCΣDescr

type Containers:
125 pp.41 BoL
126 pp.41 CPos == onV | onQC | onQT
126 pp.41 | onSC | onStk | onLT | Idle

concrete **type**s of onV, onQC, onQT, onSC, onStk, onLT and Idle
126a. pp.41 onV :: VI × BRSP × VPos
126a. pp.41 VPos == AtSea | InTer
126a. pp.41 AtSea :: Geo
126a. pp.41 InTer :: QPSid × QP$^+$
126b. pp.41 onQC :: MCCI × QCI
126c. pp.41 onQT :: MCCI × QTI
126d. pp.41 onSC :: MCCI × SCI
126e. pp.41 onStk :: MCCI × BRSP
126f. pp.41 onLT :: MCCI × LTI
126g. pp.41 Idle :: {|"idle"|}

10 RSL: The RAISE Specification Language – A Primer

256

10.1 Type Expressions

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

10.1.1 Atomic Types

Atomic types have (atomic) values. That is, values which we consider to have no proper constituent (sub-)values, i.e., cannot, to us, be meaningfully "taken apart".

RSL has a number of *built-in* atomic types. There are the Booleans, integers, natural numbers, reals, characters, and texts. 257

Basic Types::

type
[1] **Bool** **true**, **false**
[2] **Int** ... , −2, −2, 0, 1, 2, ...
[3] **Nat** 0, 1, 2, ...
[4] **Real** ..., −5.43, −1.0, 0.0, 1.23···, 2,7182···, 3,1415···, 4.56, ...
[5] **Char** "a", "b", ..., "0", ...
[6] **Text** "abracadabra"

10.1.2 Composite Types

Composite types have composite values. That is, values which we consider to have proper constituent (sub-)values, i.e., can be meaningfully "taken apart". There are two ways of expressing composite types: either explicitly, using concrete type expressions, or implicitly, using sorts (i.e., abstract types) and observer functions.

Concrete Composite Types From these one can form type expressions: finite sets, infinite sets, Cartesian products, lists, maps, etc.

Let A, B and C be any type names or type expressions, then the following are type expressions:

Composite Type Expressions::

[7] A-**set**
[8] A-**infset**
[9] A × B × ... × C
[10] A^*
[11] A^ω

[12] A $\overrightarrow{m}$ B
[13] A → B
[14] A $\tilde{\rightarrow}$ B
[15] (A)
[16] A | B | ... | C
[17] mk_id(sel_a:A,...,sel_b:B)
[18] sel_a:A ... sel_b:B

The following the meaning of the atomic and the composite type expressions:

1 The Boolean type of truth values **false** and **true**.

2 The integer type on integers ..., –2, –1, 0, 1, 2, ...

3 The natural number type of positive integer values 0, 1, 2, ...

4 The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period ("."), followed by a natural number (the fraction).

5 The character type of character values ″a″, ″bb″, ...

6 The text type of character string values ″aa″, ″aaa″, ..., ″abc″, ...

7 The set type of finite cardinality set values.

8 The set type of infinite and finite cardinality set values.

9 The Cartesian type of Cartesian values.

10 The list type of finite length list values.

11 The list type of infinite and finite length list values.

12 The map type of finite definition set map values.

13 The function type of total function values.

14 The function type of partial function values.

15 In (A) A is constrained to be:

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

16 The postulated disjoint union of types A, B, . . . , and C.

17 The record type of mk_id-named record values mk_id(av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

18 The record type of unnamed record values (av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

Sorts and Observer Functions

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

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

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

10.2 Type Definitions

10.2.1 Concrete Types

Types can be concrete in which case the structure of the type is specified by type expressions:

Type Definition::

type
  A = Type_expr

Some schematic type definitions are:

Variety of Type Definitions::

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

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

Record Types::

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

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

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

10.2.2 Subtypes

In RSL, each type represents a set of values. Such a set can be delimited by means of predicates. The set of values b which have type B and which satisfy the predicate $\mathcal{P}$, constitute the subtype A:

Subtypes::

```
type
  A = {| b:B • 𝒫(b) |}
```

10.2.3 Sorts — Abstract Types

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

Sorts::

type
A, B, ..., C

10.3 The RSL Predicate Calculus

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

Propositional Expressions::

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

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

10.3.1 Simple Predicate Expressions

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

Simple Predicate Expressions::

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

are simple predicate expressions.

10.3.2 Quantified Expressions

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

Quantified Expressions::

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

are quantified expressions — also being predicate expressions.

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

10.4 RSL Values and Operations

10.4.1 Arithmetic

Arithmetic::

type
  Nat, **Int**, **Real**
value
  $+,-,*$: **Nat**$\times$**Nat**$\rightarrow$**Nat** | **Int**$\times$**Int**$\rightarrow$**Int** | **Real**$\times$**Real**$\rightarrow$**Real**
  $/$: **Nat**$\times$**Nat**$\xrightarrow{\sim}$**Nat** | **Int**$\times$**Int**$\xrightarrow{\sim}$**Int** | **Real**$\times$**Real**$\xrightarrow{\sim}$**Real**
  $<,\leq,=,\neq,\geq,>$ (**Nat**|**Int**|**Real**) $\rightarrow$ (**Nat**|**Int**|**Real**)

10.4.2 Set Expressions

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

Set Enumerations::

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

Set Comprehension The expression, last line below, to the right of the $\equiv$, expresses set comprehension. The expression "builds" the set of values satisfying the given predicate. It is abstract in the sense that it does not do so by following a concrete algorithm.

Set Comprehension::

type
  A, B
  P = A → **Bool**
  Q = A $\xrightarrow{\sim}$ B
value
  comprehend: A-**infset** × P × Q → B-**infset**
  comprehend(s,P,Q) ≡ { Q(a) | a:A • a ∈ s ∧ P(a)}

10.4.3 Cartesian Expressions

Cartesian Enumerations Let e range over values of Cartesian types involving A, B, ..., C, then the below expressions are simple Cartesian enumerations:

Cartesian Enumerations::

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

10.4.4 List Expressions

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

{⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ...} ∈ A^*
{⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ..., ⟨e1,e2,...,en,... ⟩, ...} ∈ A^ω

⟨ a_i .. a_j ⟩

The last line above assumes a_i and a_j to be integer-valued expressions. It then expresses the set of integers from the value of e_i to and including the value of e_j. If the latter is smaller than the former, then the list is empty.

List Comprehension The last line below expresses list comprehension.

List Comprehension::

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

10.4.5 Map Expressions

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

Map Enumerations::

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

Map Comprehension The last line below expresses map comprehension:

Map Comprehension::

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

10.4.6 Set Operations

Set Operator Signatures

Set Operations::

value
19 $\in$: A $\times$ A-**infset** $\rightarrow$ **Bool**
20 $\notin$: A $\times$ A-**infset** $\rightarrow$ **Bool**
21 $\cup$: A-**infset** $\times$ A-**infset** $\rightarrow$ A-**infset**
22 $\cup$: (A-**infset**)-**infset** $\rightarrow$ A-**infset**
23 $\cap$: A-**infset** $\times$ A-**infset** $\rightarrow$ A-**infset**
24 $\cap$: (A-**infset**)-**infset** $\rightarrow$ A-**infset**
25 $\backslash$: A-**infset** $\times$ A-**infset** $\rightarrow$ A-**infset**
26 $\subset$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
27 $\subseteq$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
28 $=$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
29 $\neq$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
30 **card**: A-**infset** $\stackrel{\sim}{\rightarrow}$ **Nat**

Set Examples

Set Examples::

examples
a $\in$ {a,b,c}
a $\notin$ {}, a $\notin$ {b,c}
{a,b,c} $\cup$ {a,b,d,e} = {a,b,c,d,e}
$\cup${{a},{a,bb},{a,d}} = {a,b,d}
{a,b,c} $\cap$ {c,d,e} = {c}
$\cap${{a},{a,bb},{a,d}} = {a}
{a,b,c} $\backslash$ {c,d} = {a,bb}
{a,bb} $\subset$ {a,b,c}
{a,b,c} $\subseteq$ {a,b,c}
{a,b,c} = {a,b,c}
{a,b,c} $\neq$ {a,bb}
card {} = 0, **card** {a,b,c} = 3

Informal Explication

19 $\in$: The membership operator expresses that an element is a member of a set.

20 $\notin$: The nonmembership operator expresses that an element is not a member of a set.

21 $\cup$: The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.

22 $\cup$: The distributed prefix union operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.

23 $\cap$: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.

24 $\cap$: The prefix distributed intersection operator. When applied to a set of sets,
259 the operator gives the set whose members are in some of the operand sets.

25 $\backslash$: The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.

26 $\subseteq$: The proper subset operator expresses that all members of the left operand set are also in the right operand set.

27 $\subset$: The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.

28 $=$: The equal operator expresses that the two operand sets are identical.

29 $\neq$: The nonequal operator expresses that the two operand sets are *not* identical.

30 **card**: The cardinality operator gives the number of elements in a finite set.

Set Operator Definitions The operations can be defined as follows ($\equiv$ is the definition symbol):

Set Operation Definitions::

value

$s' \cup s'' \equiv \{ a \mid a{:}A \bullet a \in s' \vee a \in s'' \}$

$s' \cap s'' \equiv \{ a \mid a{:}A \bullet a \in s' \wedge a \in s'' \}$

$s' \setminus s'' \equiv \{ a \mid a{:}A \bullet a \in s' \wedge a \notin s'' \}$

$s' \subseteq s'' \equiv \forall a{:}A \bullet a \in s' \Rightarrow a \in s''$

$s' \subset s'' \equiv s' \subseteq s'' \wedge \exists a{:}A \bullet a \in s'' \wedge a \notin s'$

$s' = s'' \equiv \forall a{:}A \bullet a \in s' \equiv a \in s'' \equiv s \subseteq s' \wedge s' \subseteq s$

$s' \neq s'' \equiv s' \cap s'' \neq \{\}$

card $s \equiv$

  if $s = \{\}$ **then** 0 **else**

  let $a{:}A \bullet a \in s$ **in** $1 +$ **card** $(s \setminus \{a\})$ **end end**

pre s /∗ is a finite set ∗/
card s ≡ **chaos** /∗ tests for infinity of s ∗/

10.4.7 Cartesian Operations

Cartesian Operations::

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

value
va:A, vb:B, vc:C, vd:D

(va,vb,vc):G0,
(va,vb,vc):G1
((va,vb),vc):G2
(va3,(vb3,vc3)):G3

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

10.4.8 List Operations

List Operator Signatures

List Operations::

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

List Operation Examples

List Examples::

examples
hd⟨a1,a2,...,am⟩=a1
tl⟨a1,a2,...,am⟩=⟨a2,...,am⟩
len⟨a1,a2,...,am⟩=m

inds⟨a1,a2,...,am⟩={1,2,...,m}
elems⟨a1,a2,...,am⟩={a1,a2,...,am}
⟨a1,a2,...,am⟩(i)=ai
⟨a,b,c⟩^⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
⟨a,b,c⟩=⟨a,b,c⟩
⟨a,b,c⟩ ≠ ⟨a,b,d⟩

Informal Explication

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a
260 number of elements larger than or equal to i, gives the ith element of the list.
- $\widehat{\ }$: Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- =: The equal operator expresses that the two operand lists are identical.
- ≠: The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

List Operator Definitions

value
  is_finite_list: A^ω → **Bool**

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

$$
\begin{array}{l}
\textbf{inds}\ \mathsf{q} \equiv \\
\quad \textbf{case}\ \mathsf{is\_finite\_list(q)}\ \textbf{of} \\
\quad\quad \textbf{true} \rightarrow \{\ \mathsf{i} \mid \mathsf{i}{:}\textbf{Nat} \bullet 1 \leq \mathsf{i} \leq \textbf{len}\ \mathsf{q}\ \}, \\
\quad\quad \textbf{false} \rightarrow \{\ \mathsf{i} \mid \mathsf{i}{:}\textbf{Nat} \bullet \mathsf{i}{\neq}0\ \}\ \textbf{end} \\
\\
\textbf{elems}\ \mathsf{q} \equiv \{\ \mathsf{q(i)} \mid \mathsf{i}{:}\textbf{Nat} \bullet \mathsf{i} \in \textbf{inds}\ \mathsf{q}\ \}
\end{array}
$$

261

$$
\begin{array}{l}
\mathsf{q(i)} \equiv \\
\quad \textbf{if}\ \mathsf{i}{=}1 \\
\quad\quad \textbf{then} \\
\quad\quad\quad \textbf{if}\ \mathsf{q}{\neq}\langle\rangle \\
\quad\quad\quad\quad \textbf{then let}\ \mathsf{a{:}A,q'{:}Q} \bullet \mathsf{q}{=}\langle \mathsf{a}\rangle\widehat{\ }\mathsf{q'}\ \textbf{in}\ \mathsf{a}\ \textbf{end} \\
\quad\quad\quad\quad \textbf{else chaos end} \\
\quad\quad \textbf{else}\ \mathsf{q(i{-}1)}\ \textbf{end} \\
\\
\mathsf{fq}\ \widehat{\ }\ \mathsf{iq} \equiv \\
\quad\quad \langle\ \textbf{if}\ 1 \leq \mathsf{i} \leq \textbf{len}\ \mathsf{fq}\ \textbf{then}\ \mathsf{fq(i)}\ \textbf{else}\ \mathsf{iq(i} - \textbf{len}\ \mathsf{fq)}\ \textbf{end} \\
\quad\quad\ \mid \mathsf{i}{:}\textbf{Nat} \bullet \textbf{if len}\ \mathsf{iq}{\neq}\textbf{chaos then}\ \mathsf{i} \leq \textbf{len}\ \mathsf{fq}{+}\textbf{len end}\ \rangle \\
\quad \textbf{pre}\ \mathsf{is\_finite\_list(fq)} \\
\\
\mathsf{iq'} = \mathsf{iq''} \equiv \\
\quad \textbf{inds}\ \mathsf{iq'} = \textbf{inds}\ \mathsf{iq''} \wedge \forall\ \mathsf{i}{:}\textbf{Nat} \bullet \mathsf{i} \in \textbf{inds}\ \mathsf{iq'} \Rightarrow \mathsf{iq'(i)} = \mathsf{iq''(i)} \\
\\
\mathsf{iq'} \neq \mathsf{iq''} \equiv \sim(\mathsf{iq'} = \mathsf{iq''})
\end{array}
$$

10.4.9 Map Operations

Map Operator Signatures and Map Operation Examples

$$
\begin{array}{l}
\textbf{value} \\
\quad \mathsf{m(a)}\colon\ \mathsf{M} \rightarrow \mathsf{A} \overset{\sim}{\rightarrow} \mathsf{B},\ \mathsf{m(a)} = \mathsf{b} \\
\\
\quad \textbf{dom}\colon\ \mathsf{M} \rightarrow \mathsf{A}\text{-}\textbf{infset}\ [\,\text{domain of map}\,] \\
\quad\quad\ \textbf{dom}\ [\,\mathsf{a1{\mapsto}b1,a2{\mapsto}b2,...,an{\mapsto}bn}\,] = \{\mathsf{a1,a2,...,an}\} \\
\\
\quad \textbf{rng}\colon\ \mathsf{M} \rightarrow \mathsf{B}\text{-}\textbf{infset}\ [\,\text{range of map}\,] \\
\quad\quad\ \textbf{rng}\ [\,\mathsf{a1{\mapsto}b1,a2{\mapsto}b2,...,an{\mapsto}bn}\,] = \{\mathsf{b1,b2,...,bn}\} \\
\\
\quad \dagger\colon\ \mathsf{M} \times \mathsf{M} \rightarrow \mathsf{M}\ [\,\text{override extension}\,] \\
\quad\quad\ [\,\mathsf{a{\mapsto}b,a'{\mapsto}bb',a''{\mapsto}bb''}\,]\ \dagger\ [\,\mathsf{a'{\mapsto}bb'',a''{\mapsto}bb'}\,] = [\,\mathsf{a{\mapsto}b,a'{\mapsto}bb'',a''{\mapsto}bb'}\,]
\end{array}
$$

262

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

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

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

=,$\neq$: M $\times$ M $\rightarrow$ **Bool**

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

Map Operation Explication

- $m(a)$: Application gives the element that a maps to in the map m.
- **dom**: Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.
- †: Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some "pairings" of the right operand map.
- $\cup$: Merge. When applied to two operand maps, it gives a merge of these maps.
- $\backslash$: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements that are not in the right operand set. 263
- /: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.
- =: The equal operator expresses that the two operand maps are identical.
- $\neq$: The nonequal operator expresses that the two operand maps are *not* identical.
- $\circ$: Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1, to the range elements of the right operand map, m_2, such that if a is in the definition set of m_1 and maps into b, and if b is in the definition set of m_2 and maps into c, then a, in the composition, maps into c.

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

Map Operation Redefinitions::

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

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

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

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

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

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

10.5 λ-Calculus + Functions

10.5.1 The λ-Calculus Syntax

λ-Calculus Syntax::

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

10.5.2 Free and Bound Variables 264

Free and Bound Variables:: Let x, y be variable names and e, f be λ-expressions.

- $\langle$V$\rangle$: Variable x is free in x.
- $\langle$F$\rangle$: x is free in $\lambda y \bullet e$ if $x \neq y$ and x is free in e.
- $\langle$A$\rangle$: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

10.5.3 Substitution 265

In RSL, the following rules for substitution apply:

Substitution::

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

  for all variables a≠ x;
- **subst**([N/x](P Q)) ≡ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x]($\lambda x \bullet P$)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ $\lambda y \bullet$ **subst**([N/x]P),

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

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

10.5.4 α-Renaming and β-Reduction 266

α and β Conversions::

- α-renaming: λx•M

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

- β-reduction: (λx•M)(N)

  All free occurrences of x in M are replaced by the expression N provided that no free variables of N thereby become bound in the result. (λx•M)(N) ≡ **subst**([N/x]M)

10.5.5 Function Signatures 267

For sorts we may want to postulate some functions:

Sorts and Function Signatures::

```
type
   A, B, C
value
   obs_B: A → B,
   obs_C: A → C,
   gen_A: BB×C → A
```

10.5.6 Function Definitions 268

Functions can be defined explicitly:

Explicit Function Definitions::

```
value
   f: Arguments → Result
   f(args) ≡ DValueExpr

   g: Arguments →̃ Result
   g(args) ≡ ValueAndStateChangeClause
   pre P(args)
```

Or functions can be defined implicitly: 269

Implicit Function Definitions::

```
value
   f: Arguments → Result
   f(args) as result
   post P1(args,result)

   g: Arguments →̃ Result
   g(args) as result
   pre P2(args)
   post P3(args,result)
```

The symbol $\stackrel{\sim}{\to}$ indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function.

10.6 Other Applicative Expressions 270

10.6.1 Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

Let Expressions::

let a = $\mathcal{E}_d$ **in** $\mathcal{E}_b$(a) **end**

is an "expanded" form of:

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

10.6.2 Recursive let Expressions

Recursive **let** expressions are written as:

Recursive let Expressions::

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

is "the same" as:

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

where:

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

10.6.3 Predicative let Expressions

Predicative **let** expressions:

Predicative let Expressions::

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

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

10.6.4 Pattern and "Wild Card" let Expressions

Patterns and *wild cards* can be used:

Patterns::

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

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

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

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

10.6.5 Conditionals

Various kinds of conditional expressions are offered by RSL:

Conditionals::

```
if b_expr then c_expr else a_expr
end

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

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

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


10.6.6 Operator/Operand Expressions

Operator/Operand Expressions::

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

10.7 Imperative Constructs

10.7.1 Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative constructs. Imperative constructs are thus inevitable in RSL.

Statements and State Change::

```
  Unit
value
  stmt: Unit → Unit
  stmt()
```

- Statements accept no arguments.
- Statement execution changes the state (of declared variables).
- **Unit** → **Unit** designates a function from states to states.
- Statements, stmt, denote state-to-state changing functions.
- Writing () as “only” arguments to a function “means” that () is an argument of type **Unit**.

10.7.2 Variables and Assignment

Variables and Assignment::

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

10.7.3 Statement Sequences and skip

Sequencing is expressed using the ';' operator. **skip** is the empty statement having no value or side-effect.

Statement Sequences and skip::

2. **skip**
3. stm_1;stm_2;...;stm_n

10.7.4 Imperative Conditionals

Imperative Conditionals::

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

10.7.5 Iterative Conditionals

Iterative Conditionals::

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

10.7.6 Iterative Sequencing

Iterative Sequencing::

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

10.8 Process Constructs

10.8.1 Process Channels

Let A and B stand for two types of (channel) messages and i:KIdx for channel array indexes, then:

Process Channels::

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

declare a channel, c, and a set (an array) of channels, k[i], capable of communicating values of the designated types (A and B).

10.8.2 Process Composition

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels. Let P() and Q stand for process expressions, then:

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

express the parallel (‖) of two processes, or the nondeterministic choice between two processes: either external ([]) or internal (⌈⌉). The interlock (╫) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

10.8.3 Input/Output Events

Let c, k[i] and e designate channels of type A and B, then:

Input/Output Events::

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

expresses the willingness of a process to engage in an event that “reads” an input, respectively “writes” an output.

10.8.4 Process Definitions

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, via which channels they wish to engage in input and output events.

Process Definitions::

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

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

The process function definitions (i.e., their bodies) express possible events.

10.9 Simple RSL Specifications

Often, we do not want to encapsulate small specifications in schemes, classes, and objects, as is often done in RSL. An RSL specification is simply a sequence of one or more types, values (including functions), variables, channels and axioms:

Simple RSL Specifications::

type
  ...
variable
  ...
channel
  ...
value
  ...
axiom
  ...

In practice a full specification repeats the above listings many times, once for each "module" (i.e., aspect, facet, view) of specification. Each of these modules may be "wrapped" into scheme, class or object definitions.[26]

[26]For schemes, classes and objects we refer to [14, Chap. 10]

10.10 RSL Index