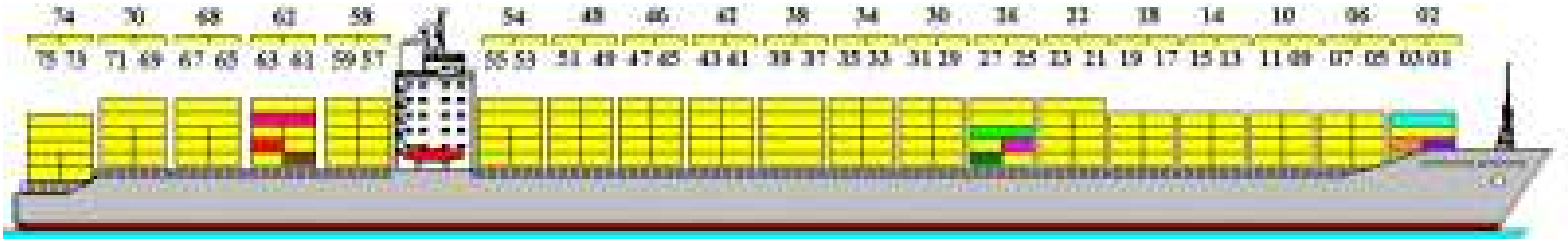
15. On A Theory of Container Stowage

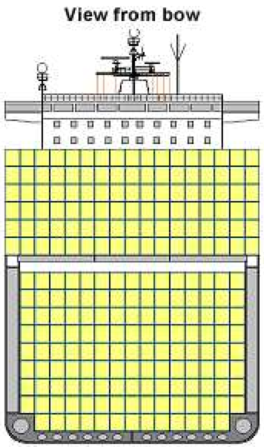
- This section is under development.
  - The idea of this section is
    - not so much to present a **container domain description**,
    - but rather to present fragments, "bits and pieces", of a theory of such a domain.
- The purpose of having a theory
  - is to "draw" upon the 'bits and pieces'
  - when expressing
    - properties of endurants and
    - definitions of
      - actions,
      - events and
      - behaviours.
- Again: this section is very much in embryo.

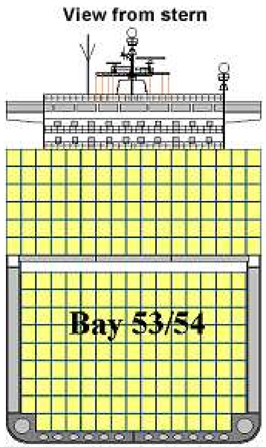
15.1. Some Pictures



A container vessel with 'bay' numbering

- Container vessels ply the seven seas and in-numerous other waters.
- They carry containers from port to port.
- The history of containers goes back to the late 1930s.
- The first container vessels made their first transports in 1956.
- Malcolm P. McLean is credited to have invented the container.
- To prove the concept of container transport he founded the container line `Sea-Land Inc.` which was sold to `Maersk Lines` at the end of the 1990s.





Bay numbers. Ship stowage cross section

- Down along the vessel, horisontally,
  - from front to aft,
  - containers are grouped, in numbered bays.

Row and tier numbers

- Bays are composed from rows, horisontally, across the vessel.
- Rows are composed from stacks, horisontally, along the vessel.
- And stacks are composed, vertically, from [tiers of] containers

15.2. Parts

15.2.1. A Basis

174. From a container vessel (cv:CV) and from a container terminal port (ctp:CTP) one can observe their bays (bays:BAYS).

type
174. CV, CTP, BAYS
value
174. obs_BAYS: (CV|CTP) $\rightarrow$ BAYS

175. The bays, bs:BS, (of a container vessel or a container terminal port) are mereologically structured as an (BId) indexed set of individual bays (b:B).

type
175. BId, B
175. BS = BId $\overrightarrow{m}$ B
value
175. obs_BS: BAYS → BS (i.e., BId $\overrightarrow{m}$ B)

176. From a bay, b:B, one can observe its rows, rs:ROWS.

177. The rows, rs:RS, (of a bay) are mereologically structured as an (RId) indexed set of individual rows (r:R).

type

176. ROWS, RId, R

177. RS = RId $\overrightarrow{m}$ R

value

176. obs_ROWS: B → ROWS

177. obs_RS: ROWS → RS (i.e., RId $\overrightarrow{m}$ R)

178. From a row, r:R, one can observe its stacks, STACKS.

179. The stacks, ss:SS (of a row) are mereologically structured as an (SId) indexed set of individual stacks (s:S).

type

178. STACKS, SId, S

179. SS = SId $\overrightarrow{m}$ S

value

178. obs_STACKS: R → STACKS

179. obs_SS: STACKS → SS (i.e., SId $\overrightarrow{m}$ S)

180. A stack (s:S) is mereologically structured as a linear sequence of containers (c:C).

type
180. C
180. S = C*

- The containers of the same stack index across stacks are called the tier at that index, cf. photo on Page 509..

181. A container is here considered a composite part
    (a) of the container box, k:K
    (b) and freight, f:F.
182. Freight is considered composite
    (a) and consists of zero, one or more colli (package, indivisible unit of freight),
    (b) each having a unique colli identifier (over all colli of the entire world!).
    (c) Container boxes likewise have unique container identifiers.

type
181. C, K, F, P
value
181(a). obs_K: C → K
181(b). obs_F: C → F
182(a). obs_Ps: F → P-**set**
type
182(b). PI
182(c). CI
value
182(b). uid_P: P → PI
182(c). uid_C: C → CI

15.2.2. Mereological Constraints

183. For any bay of a vessel the index sets of its rows are identical.

184. For a bay of a vessel the index sets of its stacks are identical.

axiom

183. $\forall$ cv:CV ·
183. $\forall$ b:B·b $\in$ **rng** obs_BS(obs_BAYS(cv))$\Rightarrow$
183. **let** rws=obs_ROWS(b) **in**
183. $\forall$ r,r′:R·{r,r′}$\subseteq$**rng** obs_RS(b)$\Rightarrow$**dom** r=**dom** r′
184. $\wedge$ **dom** obs_SS(r) = **dom** obs_SS(r′) **end**

15.2.3. Stack Indexes

185. A container stack (and a container) is designated by an index triple: a bay index, a row index and a stack index.

186. A container index triple is valid, for a vessel, if its indices are valid indices.

type

185. $\mathrm{StackId} = \mathrm{BId}\times\mathrm{RId}\times\mathrm{SId}$

value

186. valid_address: BS $\rightarrow$ StackId $\rightarrow$ **Bool**
186. valid_address(bs)(bid,rid,sid) $\equiv$
186. bid $\in$ **dom** bs
186. $\wedge$ rid $\in$ **dom** (obs_RS(bs))(bid)
186. $\wedge$ sid $\in$ **dom** (obs_SS((obs_RS(bs))(bid)))(rid)

- The above can be defined in terms of the below.

type
  BayId = BId
  RowId = BId×RId
value
186. valid_BayId: V → BayId → **Bool**
186. valid_BayId(v)(bid) ≡ bid ∈ **dom** obs_BS(obs_BAYS(v))

186. get_B: V → BayId $\stackrel{\sim}{\to}$ B
186. get_B(v)(bid) ≡ (get_B(bs))(bid) **pre**: valid_BId(v)(bid)

186. get_B: BS → BayId $\stackrel{\sim}{\to}$ B
186. get_B(bs)(bid) ≡ (obs_BS(obs_BAYS(v)))(bid) **pre**: bid ∈ **dom** bs

186. valid_RowId: V → RowId → **Bool**
186. valid_RowId(v)(bid,rid) ≡ rid ∈ **dom** obs_RS(get_B(v)(bid))
186. **pre**: valid_BayId(v)(bid)

186. get_R: V → RowId $\xrightarrow{\sim}$ R
186. get_R(v)(bid,rid) ≡ get_R(obs_BS(v))(bid,rid) **pre**: valid_RowId(v)(bid

186. get_R: BS → RowId $\xrightarrow{\sim}$ R
186. get_R(bs)(bid,rid) ≡ (obs_RS(get_RS(bs(bid))))(rid)
186. **pre**: valid_RowId(v)(bid,rid)

186. get_S: V → StackId $\stackrel{\sim}{\rightarrow}$ S
186. get_S(v)(bid,rid,sid) ≡ (obs_SS(get_R(get_B(v)(bid,rid))))(sid)
186. **pre**: valid_address(v)(bid,rid,sid)

186. get_C: V → StackId $\xrightarrow{\sim}$ C
186. get_C(v)(stid) ≡ get_C(obs_BS(v))(stid) **pre**: get_S(v)(bid,rid,sid) ≠ ⟨⟩

186. get_C: BS → StackId $\xrightarrow{\sim}$ C
186. get_C(bs)(bid,rid,sid) ≡ **hd**(obs_SS(get_R((bs(bid))(rid))))(sid)
186.   **pre**: get_S(bs)(bid,rid,sid) ≠ ⟨⟩

186. valid_addresses: V → StackId-**set**
186. valid_addresses(v) ≡ {adr|adr:StackId•valid_address(adr)(v)}

187. The predicate non_empty_designated_stack checks whether the designated stack is non-empty.

187. non_empty_designated_stack: V → StackId → **Bool**
187. non_empty_designated_stack(v)(bid,rid,sid) ≡ get_S(v)(bid,rid,sid) ≠ ⟨⟩

188. Two vessels have the same mereology if they have the same set of valid-addresses.

value

188. unchanged_mereology: BS × BS → **Bool**

188. unchanged_mereology(bs,bs′) ≡ valid_addresses(bs) = valid_addresses(b

189. The designated stack, s', of a vessel, v' is popped with respect the "same designated" stack, s, of a vessel, v
    (a) if the ordered sequence of the containers of s' are identical to the ordered sequence of containers of all but the first container of s.

189. popped_designated_stack: BS $\times$ BS $\rightarrow$ StackId $\rightarrow$ **Bool**
189. popped_designated_stack(bs,bs′)(stid) $\equiv$
189(a). **tl** get_S(v)(stid) = get_S(bs′)(stid)

190. For a given stack index, valid for two bays (**bs**, **bs′**) of two vessels or two container terminal ports, and say **stid**, these two bays enjoy the unchanged_non_designated_stacks(bs,bs′)(stid) property

(a) if the stacks (of the two bays) not identified by **stid** are identical.

190. unchanged_non_designated_stacks: BS × BS → StackId → **Bool**
190. unchanged_non_designated_stacks(bs,bs′)(stid) ≡
190(a). ∀ adr:StackId•adr ∈ valid_addresses(v)\{stid}⇒
190(a). get_S(bs)(adr) = get_S(bs′)(adr)
190. **pre**: unchanged_mereology(bs,bs′)

15.2.4. Stowage Schemas

191. By a stowage schema of a vessel we understand a “table”

(a) which for every bay identifier of that vessel records a bay schema

(b) which for every row identifier of an identified bay records a row schema

(c) which for every stack identifier of an identified row records a stack schema

(d) which for every identified stack records its tier schema.

(e) A stack schema records for every tier index (which is a natural number) the type of container (contents) that may be stowed at that position.

(f) The tier indexes of a stack schema form a set of natural numbers from one to the maximum number in the index set.

value

191. obs_StoSchema: V $\rightarrow$ StoSchema

type

191(a). StoSchema = BId $\overrightarrow{m}$ BaySchema

191(b). BaySchema = RId $\overrightarrow{m}$ RowSchema

191(c). RowSchema = SId $\overrightarrow{m}$ StaSchema

191(d). StaSchema = **Nat** $\overrightarrow{m}$ C_Type

191(e). C_Type

axiom

191(f). $\forall$ stsc:StaSchema $\cdot$ **dom** stsc = {1..**max** **dom** stsc}

192. One can define a function which from an actual vessel "derives" its "current stowage schema".

```
192. cur_sto_schema: V → StoSchema
192. cur_sto_schema(v) ≡
192.    let bs = obs_BS(obs_BAYS(v)) in
192.    [ bid ↦ let rws = obs_RS(obs_ROWS(bs(bid))) in
192.            [ rid ↦ let ss = obs_SS(obs_STACKS(rws)(rid)) in
192.                    [ sid ↦ ⟨ analyse_container(ss(i))|i:Nat•i ∈ inds ss ⟩
192.                    | sid:SId•sid ∈ ss ] end
192.            | rid:RId•rid ∈ dom rws ] end
192.    | bid:BId•bid ∈ dom ds ] end

192. analyse_container: C → C_Type
```

193. Given a stowage schema and a current stowage schema one can check the latter for conformance wrt. the former.

```
193. conformance: StoSchema × StoSchema → Bool
193. conformance(stosch,cur_stosch) ≡
193.     dom cur_stosch = dom stosch
193. ∧ ∀ bid:BId · bid ∈ dom stosch ⇒
193.         dom cur_stosch(bid) = dom stosch(bid)
193.     ∧ ∀ rid:RId · rid ∈ dom(stosch(bid))(rid) ⇒
193.             dom(cur_stosch(bid))(rid) = dom(stosch(bid))(rid)
193.         ∧ ∀ sid:SId · sid ∈ dom(cur_stosch(bid))(rid)
193.                 ∀ i:Nat · i ∈ inds((cur_stosch(bid))(rid))(sid) ⇒
193.                     conform((((cur_stosch(bid))(rid))(sid))(i),
193.                                 (((stosch(bid))(rid))(sid))(i))

193. conform: C_Type × C_Type → Bool
```

194. From a vessel one can observe its mandated stowage schema.
195. The current stowage schema of a vessel must always conform to its mandated stowage schema.

value

194. obs_StoSchema: V → StoSchema

195. stowage_conformance: V → **Bool**
195. stowage_conformance(v) ≡
195. **let** mandated = obs_StoSchema(v),
195. current = cur_sto_schema(v) **in**
195. conformance(mandated,current) **end**

15.3. Actions

15.3.1. Remove Container from Vessel

20. The remove_Container_from_Vessel action applies to a vessel and a stack address and conditionally yields an updated vessel and a container.

20(a). We express the 'remove from vessel' function primarily by means of an auxiliary function remove_C_from_BS, remove_C_from_BS(obs_BS(v))(stid), and some further post-condition on the before and after vessel states (cf. Item 20(d)).

20(b). The remove_C_from_BS function yields a pair: an updated set of bays and a container.

20(c). When obs_erving the BayS from the updated vessel, v′, and pairing that with what is assumed to be a vessel, then one shall obtain the result of remove_C_from_BS(obs_BS(v))(stid).

20(d). Updating, by means of remove_C_from_BS(obs_BS(v))(stid), the bays of a vessel must leave all other properties of the vessel unchanged.

21. The pre-condition for remove_C_from_BS(bs)(stid) is

21(a). that stid is a valid_address in bs, and

21(b). that the stack in bs designated by stid is non_empty.

22. The post-condition for remove_C_from_BS(bs)(stid) wrt. the updated bays, bs′, is

22(a). that the yielded container, i.e., c, is obtained, get_C(bs)(stid), from the top of the non-empty, designated stack,

22(b). that the mereology of bs′ is unchanged, unchanged_mereology(bs,bs′). wrt. bs. ,

22(c). that the stack designated by stid in the "input" state, bs, is popped, popped_designated_stack(bs,bs′)(stid), and

22(d). that all other stacks are unchanged in bs′ wrt. bs, unchanged_non_designated_stacks(bs,bs′)(stid).

value

20. remove_C_from_V: V $\rightarrow$ StackId $\stackrel{\sim}{\rightarrow}$ (V$\times$C)
20. remove_C_from_V(v)(stid) **as** (v$'$,c)
20(c). (obs_BS(v$'$),c) = remove_C_from_BS(obs_BS(v))(stid)
20(d). $\wedge$ props(v)=props(v$''$)

20(b). remove_C_from_BS: BS $\rightarrow$ StackId $\rightarrow$ (BS$\times$C)
20(a). remove_C_from_BS(bs)(stid) **as** (bs$'$,c)
21(a). **pre**: valid_address(bs)(stid)
21(b). $\wedge$ non_empty_designated_stack(bs)(stid)
22(a). **post**: c = get_C(bs)(stid)
22(b). $\wedge$ unchanged_mereology(bs,bs$'$)
22(c). $\wedge$ popped_designated_stack(bs,bs$'$)(stid)
22(d). $\wedge$ unchanged_non_designated_stacks(bs,bs$'$)(stid)

15.3.2. Remove Container from CTP

- We define a remove action similar to that of the previous section.

196. Instead of vessel bays we are now dealing with the bays of container terminal ports.

We omit the narrative — which is very much like that of narrative Items 20(c) and 20(d).

value

196. remove_C_from_CTP: CTP $\rightarrow$ StackId $\stackrel{\sim}{\rightarrow}$ (CTP×C)
196. remove_C_from_CTP(ctp)(stid) **as** (ctp′,c)
20(c). (obs_BS(ctp′),c) = remove_C_from_BS(obs_BS(ctp))(stid)
20(d). $\wedge$ props(ctp)=props(ctp″)

15.3.3. Stack Container on Vessel

197. Stacking a container at a vessel bay stack location

   (a)

   (b)

   (c)

value

197. stack_C_on_vessel: BS $\rightarrow$ StackId $\stackrel{\sim}{\rightarrow}$ C $\stackrel{\sim}{\rightarrow}$ BS

197(a). stack_C_on_vessel(bs)(stid)(c) **as** bs′

197(a). **comment:** bs is bays of a v:V, i.e., bs = obs_BS(v)

197(b). **pre**:

197(c). **post**:

15.3.4. Stack Container in CTP

198.

199.

200.

201.

value

198. stack_C_in_CTP: CTP → StackId → C $\stackrel{\sim}{\rightarrow}$ CTP
199. stack_C_in_CTP(ctp)(stid)(c) **as** ctp′
200.   **pre**:
201.   **post**:

15.3.5. Transfer Container from Vessel to CTP

202.

203.

204.

205.

value

202. transfer_C_from_V_to_CTP: V→StackId $\xrightarrow{\sim}$ CTP→StackId $\xrightarrow{\sim}$ (V×CTP)
203. transfer_C_from_V_to_CTP(v)(v_stid)(ctp)(ctp_stid) ≡
204. **let** (c,v′) = remove_C_from_V(v)(v_stid) **in**
204. (v′,stack_C_in_CTP(ctp)(ctp_stid)(c)) **end**

15.3.6. Transfer Container from CTP to Vessel

206.

207.

208.

value

206. transfer_C_from_CTP_to_V: CTP→StackId $\xrightarrow{\sim}$V→StackId $\xrightarrow{\sim}$(CTP×V)
207. transfer_C_from_CTP_to_V(ctp)(ctp_stid)(v)(v_stid) ≡
208. **let** (c,ctp′) = remove_C_from_CTP(ctp)(ctp_stid) **in**
208. (ctp′,stack_C_in_CTP(ctp)(ctp_stid)(c)) **end**

Any Questions ?