Swinging UML

How to Make Class Diagrams and State Machines Amenable to Constraint Solving and Proving

<<UML>> 2000

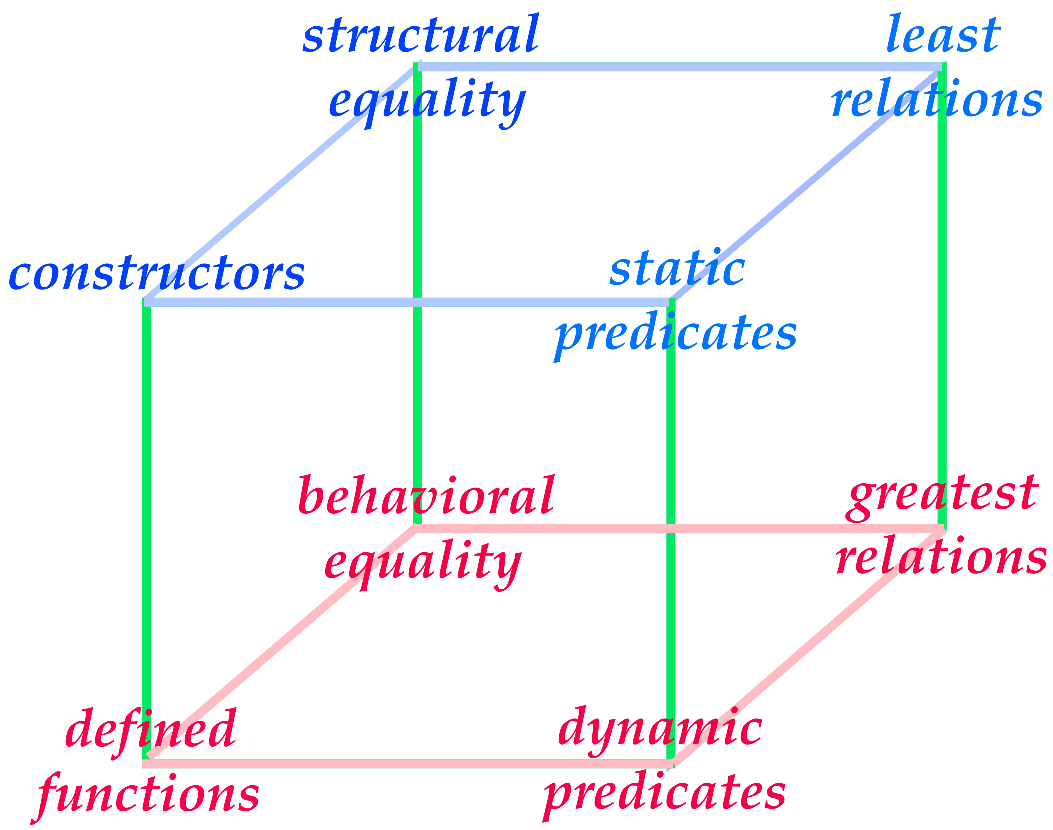
Peter Padawitz
University of Dortmund
Germany

`http://ls5.cs.uni-dortmund.de/~peter/Swinging.html`

Contents:

Swinging types, class diagrams and constraints

Swinging types are an axiomatic specification formalism, based on *many-sorted logic* and *Herbrand (= term) models*.



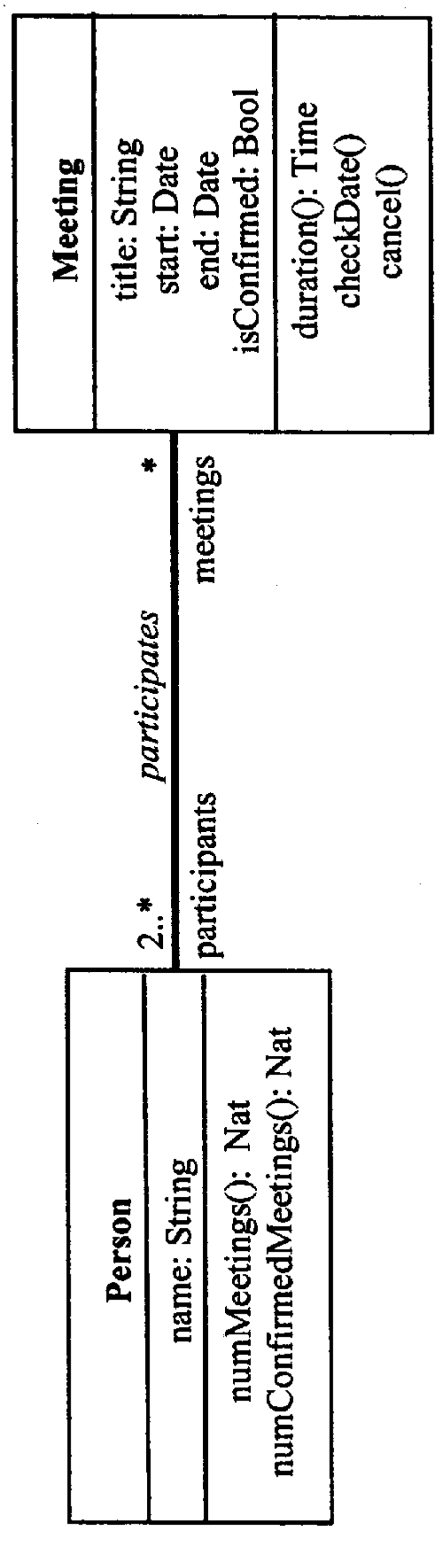
More characteristics:

One-tiered approach: static and dynamic system aspects are treated within the same formalism.
Herbrand models provide a unique semantics.
State (or hidden) terms denote ☞ sets of attribute values, ☞ compositions of substates, ☞ object histories (= sets of past actions), ☞ processes (= sets of future actions) and/or ☞ infinite structures (of, e.g., an *uncountable* domain).
In the Herbrand model, a state of an object ❑ is ☞ a *normal form* consisting of *constructors* $\leadsto$ *initial-algebra semantics* or ☞ a *behavior*, i.e., an interpretation of all *context terms* consisting of *destructors* $\leadsto$ *final-coalgebra semantics* $\leadsto$ *"Kripke model"* or both: ☞ a *normal form* consisting of *constructors* some which are constants denoting *behaviors*.

normal form

interpretation of contexts

☞ *Local functions* admit the restriction of *normal forms* to those resulting from non-local-function calls. ☞ *Constraint axioms* admit the restriction of *behaviors*.
Transition systems and *I/O-relations* come as *dynamic predicates* $\leadsto$ integration of *modal logic*.
There are a *structural equality* and a *behavioral equality*. Both equalities coincide on *visible* terms.
Structural equality is a congruence and *inverse compatible* with *constructors*.
Behavioral equality is a *weak* congruence, i.e. only a *bisimulation* w.r.t. dynamic predicates, and *inverse compatible* with *object constructors*.
The associated proof rules exploit these properties as well as the interpretation of (μ-/ν-)predicates as *least*/*greatest* solutions of their *Horn*/*co-Horn* axioms: ☞ *term/atom unfolding, splitting* and *clash* for *simplification*, ☞ *induction* for proving *pre/postconditions* and *invariants*, ☞ *coinduction* for proving *properties of states*: reachability, fairness, behavioral equivalence,...



from: H. Hußmann, M. Cerioli, G. Reggio, F. Tort, Abstract Data Types and UML Models, Report DISI-TR-99-15, University of Genova 1999

OCL constraint:

```
context Meeting :: checkDate()
        post : isConfirmed =
               self.participants ->
               collect(meetings) ->
               forAll(m | m <> self and
                          m.isConfirmed implies
                            (after(self.end,m.start) or
                            (after(m.end,self.start)))
```

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{FINSET} = \mathbf{OBJECT}\ \texttt{and}\ \mathbf{OBJECT}[object \mapsto object']\ \texttt{and}} \\
\multicolumn{2}{l}{\qquad\qquad\ \mathbf{BOOL}\ \texttt{and}\ \mathbf{NAT}\ \texttt{then}} \\
\texttt{dessorts} & object \quad object' \\
\texttt{consorts} & set = set(object) \\
\texttt{local} & \\
\texttt{constructs} & nil : \to set \\
 & \_ :: \_ : object \times set \to set \\
\texttt{defuncts} & \emptyset : \to set \\
 & insert, remove : object \times set \to set \\
 & filter : (object \to bool) \times set \to set \\
 & map : (object \to object') \times set \to set \\
 & |\_| : set \to nat \\
 & forall : (object \to bool) \times set \to bool \\
\mu\texttt{-preds} & \_ \in \_ : object \times set \to bool \\
\texttt{vars} & x, y : object \quad s : set \quad f : object \to object' \\
 & g : object \to bool \\
\texttt{Horn axioms} & \emptyset \ \equiv\ nil \\
 & insert(x, nil) \ \equiv\ x :: nil \\
 & insert(x, x :: s) \ \equiv\ x :: s \\
 & insert(x, y :: s) \ \equiv\ x :: (y :: s) \quad \Leftarrow \quad x < y \\
 & insert(x, y :: s) \ \equiv\ y :: insert(x, s) \quad \Leftarrow \quad x > y \\
 & remove(x, nil) \ \equiv\ \emptyset \\
 & remove(x, x :: s) \ \equiv\ s \\
 & remove(x, y :: s) \ \equiv\ y :: remove(x, s)) \quad \Leftarrow \quad x \not\equiv y \\
 & filter(g, nil) \ \equiv\ \emptyset \\
 & filter(g, x :: s) \ \equiv\ filter(g, s) \quad \Leftarrow \quad g(x) \equiv false \\
 & filter(g, x :: s) \ \equiv\ x :: filter(g, s) \quad \Leftarrow \quad g(x) \equiv true \\
 & map(f, nil) \ \equiv\ \emptyset \\
 & map(f, x :: s) \ \equiv\ insert(f(x), map(f, s)) \\
 & |nil| \ \equiv\ 0 \\
 & |x :: s| \ \equiv\ |s| + 1 \\
 & forall(g, nil) \ \equiv\ true \\
 & forall(g, x :: s) \ \equiv\ g(x)\ and\ forall(g, s)
\end{array}
$$

$\mathbf{PERSON\_SET} = \mathbf{FINSET}[object \mapsto Person,\ < \mapsto <_{lex}]$ `and`
$\mathbf{STRING}$ `then`

`destructs`	$name : Person \rightarrow String$
`vars`	$p, p' : Person$
`Horn axioms`	$p < p' \ \Leftarrow\ name(p) <_{lex} name(p')$

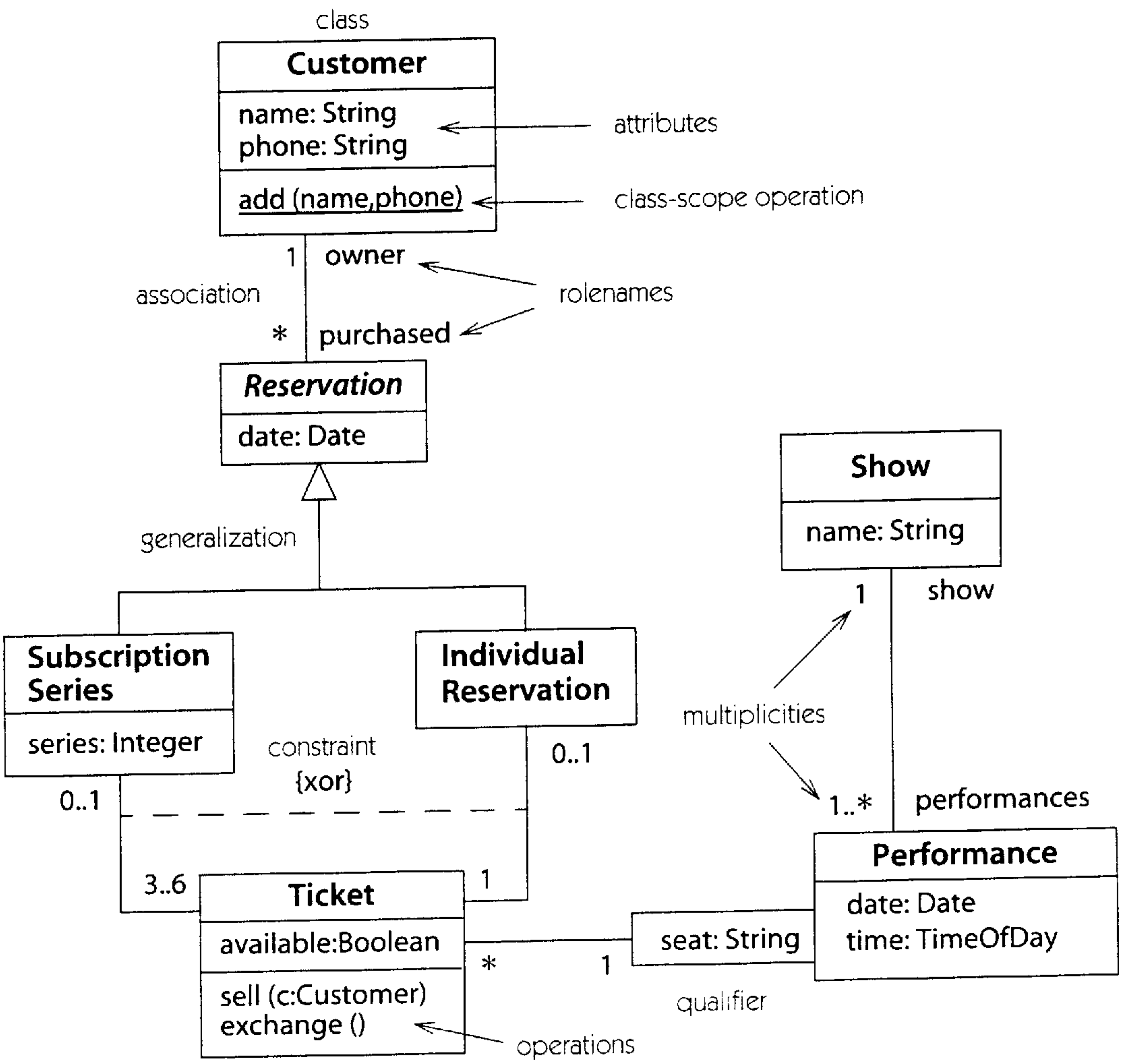
$\mathbf{MEETING\_SET} = \mathbf{FINSET}[object \mapsto Meeting,\ < \mapsto <_{lex}]$ `and`
$\mathbf{STRING}$ `then`

`destructs`	$title : Meeting \rightarrow String$
`vars`	$m, m' : Meeting$
`Horn axioms`	$m < m' \ \Leftarrow\ title(p) <_{lex} title(p')$

$$
\begin{array}{ll}
\textbf{PERSON\&MEETING} = & \textbf{PERSON\_SET}\ \texttt{and} \\
& \textbf{MEETING\_SET}\ \texttt{and} \\
& \textbf{DATE\&TIME}\ \texttt{then}
\end{array}
$$

$$
\begin{array}{ll}
\texttt{dessorts} & Person\quad Meeting \\
\texttt{constructs} & checkDate : Meeting \to Meeting \\
& cancel : Meeting \to Meeting \\
\texttt{destructs} & meetings : Person \to set(Meeting) \\
& participants : Meeting \to set(Person) \\
& start, end : Meeting \to Date \\
& isConfirmed : Meeting \to bool \\
\texttt{defuncts} & numMeetings : Person \to Nat \\
& numConfirmedMeetings : Person \to Nat \\
& duration : Meeting \to Time \\
& consistent : Meeting \times Meeting \to bool \\
\texttt{vars} & p : Person \quad m, m' : Meeting \quad ms : set(Meeting) \\
& ps : set(Person) \\
\texttt{constraints} & is_{Meeting}(m) \ \Rightarrow\ |participants(m)| \geq 2 \\
\texttt{Horn axioms} & numMeetings(p) \ \equiv\ |meetings(p)| \\
& numConfirmedMeetings(p) \\
& \quad \equiv\ |filter(isConfirmed, meetings(p))| \\
& duration(m) \ \equiv\ end(m) - start(m) \\
& isConfirmed(checkDate(m)) \\
& \quad \equiv\ forall(\lambda m'.consistent(m, m'), \\
& \qquad\qquad remove(m, ms)) \\
& \quad \Leftarrow\ participants(m) \equiv ps\ \wedge \\
& \qquad map(meetings, ps) \equiv ms \\
& isConfirmed(cancel(m)) \ \equiv\ false \\
& consistent(m, m') \ \equiv\ not(isConfirmed(m'))\ or \\
& \qquad end(m) < start(m')\ or \\
& \qquad end(m') < start(m)
\end{array}
$$

Inheritance

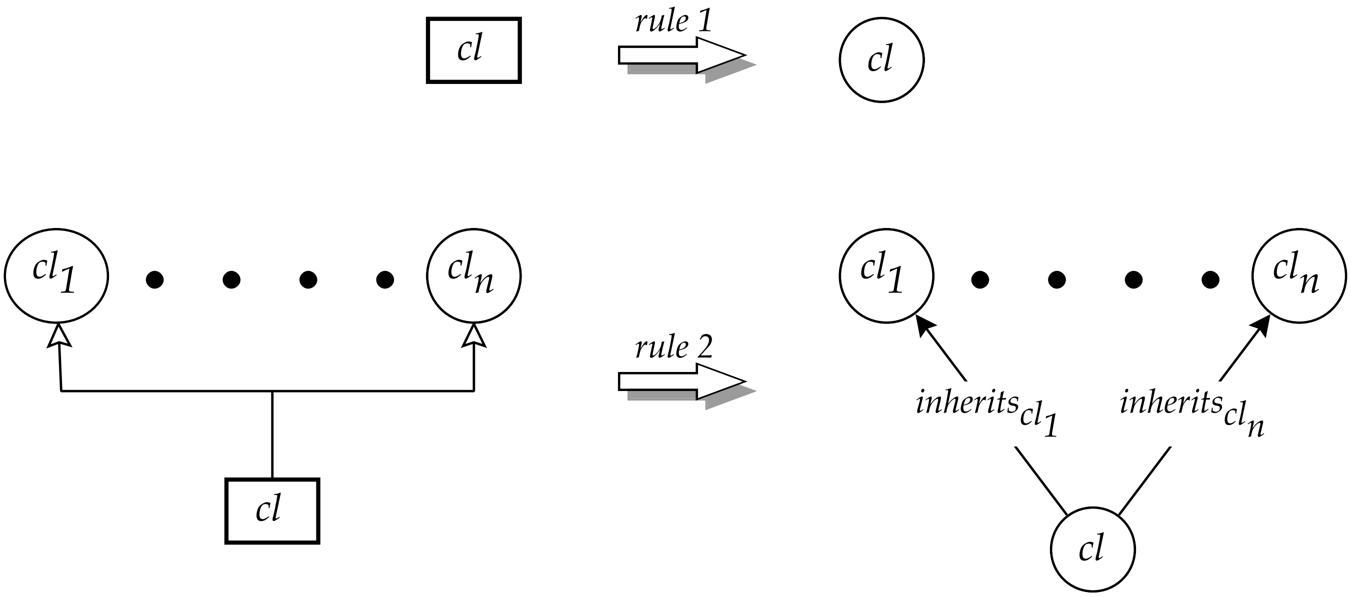


from: J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley 1999

$$
\begin{array}{lll}
\mathbf{RESERVATION} = & \multicolumn{2}{l}{\mathbf{INT}\ \texttt{and}\ \mathbf{STRING}[String]\ \texttt{and}} \\
& \multicolumn{2}{l}{\mathbf{TIME}[Date, TimeOfDay]\ \texttt{and}} \\
& \multicolumn{2}{l}{\mathbf{FINSET}[object \mapsto Reservation + SS + RI]\ \texttt{and}} \\
& \multicolumn{2}{l}{\mathbf{FINSET}[object \mapsto Ticket]\ \texttt{and}} \\
& \multicolumn{2}{l}{\mathbf{FINSET}[object \mapsto Performance]\ \texttt{then}}
\end{array}
$$

$$
\begin{array}{ll}
\texttt{dessorts} & Customer\quad Reservation\quad SubscriptionSeries\quad Ticket \\
& IndividualReservationTicket\quad Show\quad Performance \\
& CustomId = String \\
& TicketId = Date \times (int + 1) \times String \times TimeOfDay \\
\texttt{consorts} & SS = Reservation \times SubscriptionSeries \\
& IR = Reservation \times IndividualReservation \\
\texttt{constructs} & new : CustomId \times String \to Customer \\
& new : TicketId \to Ticket \\
& new : Date \times TimeOfDay \to Performance \\
& new : int \to SubscriptionSeries \\
& new : 1 \to IndividualReservation \\
& buy : Customer \times Ticket \to Customer \\
& sell : Ticket \times Customer \to Ticket \\
& exchange : Ticket \to Ticket \\
\texttt{destructs} & name : Customer \to String \\
& phone : Customer \to String \\
& purchased : Customer \to set(Reservation + SS + IR) \\
& date : Reservation \to Date \\
& owner : Reservation \to Customer \\
& series : SubscriptionSeries \to int \\
& tickets : SubscriptionSeries \to set(Ticket) \\
& ticket : IndividualReservation \to Ticket
\end{array}
$$

$$
\begin{array}{ll}
 & available\colon Ticket \to bool \\
 & ID\colon Ticket \to TicketId \\
 & subscriptionSeries\colon Ticket \to 1 + SS \\
 & individualReservation\colon Ticket \to 1 + IR \\
 & qperformance\colon Ticket \to String \times Performance \\
 & name\colon Show \to String \\
 & performances\colon Show \to set(Performance) \\
 & date\colon Performance \to Date \\
 & time\colon Performance \to TimeOfDay \\
 & show\colon Performance \to Show \\
 & seat\colon String \times Performance \to String \\
 & ticket\colon String \times Performance \to Ticket \\
\texttt{vars} & t : Ticket \quad ss : SS \quad ir : IR \quad sh : Show \\
\texttt{constraints} & is_{SS}(ss) \ \Rightarrow \ 3 \le |tickets(ss)| \le 6 \\
 & is_{Ticket}(t) \ \Rightarrow \ (subscriptionSeries(t) \equiv \kappa_1() \ \stackrel{+}{\vee} \\
 & \qquad\qquad individualReservation(t) \equiv \kappa_1()) \\
 & is_{Show}(sh) \ \Rightarrow \ |performances(sh)| > 0 \\
\texttt{Horn axioms} & \ldots
\end{array}
$$



Generalizations are translated into product and sum sorts

Refinement: Model change via signature morphisms

Given signatures $\Sigma = (S, F, P)$ and $\Sigma' = (S', F', P')$, a *signature morphism* $rep : \Sigma \to \Sigma'$ consists of functions

$$rep_S : S \to S', \quad rep_F : F \to F', \quad rep_P : P \to P'$$

such that for all $f : s_1 \dots s_n \to s \in F$ and $r : s_1 \dots s_n \in P$,

$$rep_F(f) : rep_S(s_1) \dots rep_S(s_n) \to rep_S(s),$$
$$rep_P(r) : rep_S(s_1) \dots rep_S(s_n).$$

Given a Σ'-structure B, the *rep-reduct* $B|_{rep}$ is the Σ-structure that is defined by:

$$(B|_{rep})_s = B_{rep(s)} \text{ for all } s \in S,$$
$$f^{B|_{rep}} = rep(f)^B \text{ for all } f \in F \cup P.$$

B_{rep} denotes the *least Σ-substructure* of $B|_{rep}$.

Let A be the Herbrand model of an *abstract* specification $SP = (\Sigma, AX)$ and B be the Herbrand model of a *concrete* specification $SP' = (\Sigma', AX')$. *SP' refines SP along rep* if

(1) the *refinement model* B_{rep} satisfies AX,

(2) SP' is *consistent w.r.t.* SP (i.e., negation-free formulas that are valid in B_{rep} are also valid in A).

If SP *has complement predicates* and rep *preserves negation*, then (2) follows from (1).

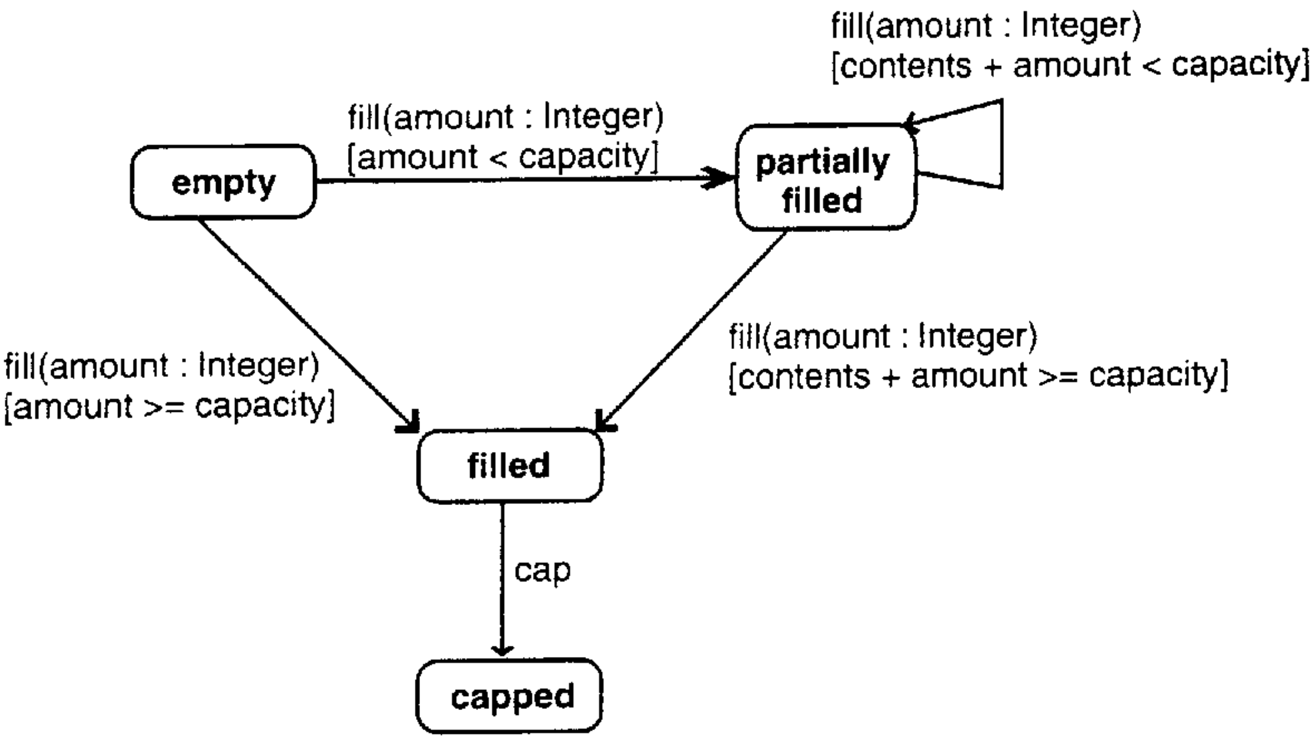
☞ Refinements make a *loose semantics* obsolete.

☞ Structural equalities may be refined to behavioral ones. $\leadsto$ Visible sorts are mapped to hidden ones.

☞ Global operations are refined to message passing primitives by mapping the operations' — *static* — I/O relations to — *dynamic* — transition relations.

State machines

$$state_1 \xrightarrow{event(x)[guard(x)]/action(x)} state_2.$$



from: J.B. Warmer, A.G. Kleppe, The Object Constraint Language, Addison-Wesley 1999

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{BOTTLESTATE} = \textbf{NAT}\ \texttt{then}} \\
\texttt{consorts} & Bottle \\
\texttt{constructs} & new : nat \to Bottle \\
 & fill : Bottle \times nat \to Bottle \\
 & cap : Bottle \to Bottle \\
\texttt{destructs} & capacity, contents : Bottle \to nat \\
 & capped : Bottle \\
\texttt{static}\ \mu\texttt{-preds} & empty, filled, partially\_filled : Bottle \\
\texttt{vars} & n : nat \quad b : Bottle \\
\texttt{Horn axioms} & capacity(new(n)) \equiv n \\
 & contents(new(n)) \equiv 0 \\
 & capacity(fill(b,n)) \equiv capacity(b) \\
 & contents(fill(b,n)) \\
 & \quad \equiv \ min(contents(b) + n, capacity(b)) \\
 & capacity(cap(b)) \equiv capacity(b) \\
 & contents(cap(b)) \equiv contents(b) \\
 & capped(cap(b)) \\
 & empty(b) \quad \Leftarrow \quad contents(b) \equiv 0 \\
 & filled(b) \quad \Leftarrow \quad contents(b) \equiv capacity(b) \\
 & partially\_filled(b) \\
 & \quad \Leftarrow \quad 0 < contents(b) < capacity(b)
\end{array}
$$

☞ Here the states are *attribute (= destructor) values*.

☞ What distinguishes *transition guards* from (properties of) states?

Introducing events and transitions:

BOTTLETRANS = BOTTLESTATE `then`

`vissorts`	$event$
`constructs`	$Fill : nat \to event$
	$Cap : \to event$
`dynamic preds`	$\_ \xrightarrow{\_} \_ : Bottle \times event \times Bottle$
`Horn axioms`	$b \xrightarrow{Fill(n)} fill(b, n) \quad \Leftarrow \quad empty(b)$
	$b \xrightarrow{Fill(n)} fill(b, n) \quad \Leftarrow \quad partially\_filled(b)$
	$b \xrightarrow{Cap} cap(b) \quad \Leftarrow \quad filled(b)$

Introducing communication and modalities:

MUTEX = **NAT** `and` **BOOL** `and` **FINSET**$[object \mapsto user]$ `then`

`vissorts`	$event \quad label$
`dessorts`	$user \quad resource$
`consorts`	$environment = set(user) \times resource$
`oconstructs`	$free, used : \to resource$
`constructs`	$new : int \to user$
	$access, release : user \to user$
	$New : nat \to event$
	$Request, Access, Release : \to event$
	$\_.\_ : int \times event \to event$
	$\_ : event \to label$
	$\_/\_ : event \times event \to label$
`destructs`	$id : user \to nat$
	$uses : user \to bool$
`dynamic preds`	$\_ \xrightarrow{-} \_ : user \times label \times user$
	$\_ \xrightarrow{-} \_ : resource \times label \times resource$
	$\_ \xRightarrow{-} \_ : set(user) \times label \times set(user)$
	$\_ \xRightarrow{-} \_ : environment \times label \times environment$
ν`-preds`	$mutex, \Box mutex : environment$
`vars`	$i : nat \quad u, u' : user \quad r, r' : resource \quad s : set(user)$
	$e, a : event \quad l : label \quad env, env' : environment$

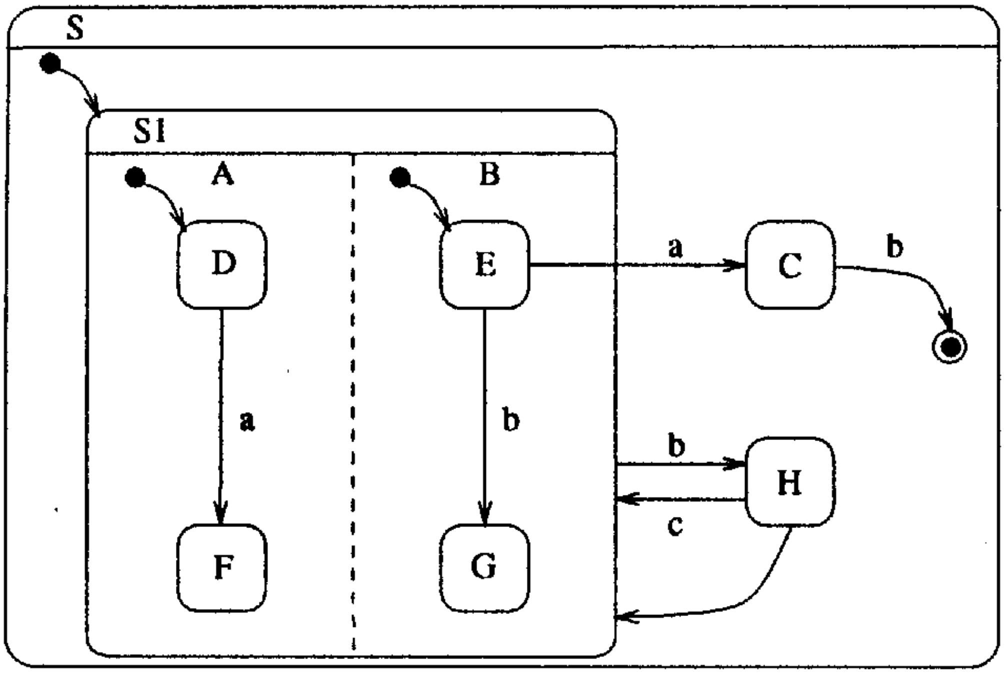
`Horn axioms`

$$
\begin{aligned}
&id(new(i)) \equiv i\\
&uses(new(i)) \equiv false\\
&id(access(u)) \equiv id(u)\\
&uses(access(u)) \equiv true\\
&id(release(u)) \equiv id(u)\\
&uses(release(u)) \equiv false\\
&u \stackrel{Request}{\longrightarrow} u \quad \Leftarrow \quad uses(u) \equiv false\\
&u \stackrel{Access}{\longrightarrow} access(u) \quad \Leftarrow \quad uses(u) \equiv false\\
&u \stackrel{Release}{\longrightarrow} release(u) \quad \Leftarrow \quad uses(u) \equiv true\\
&free \stackrel{i.Request/i.Access}{\longrightarrow} used\\
&used \stackrel{i.Release}{\longrightarrow} free\\
&s \stackrel{New(i)}{\Longrightarrow} insert(new(i), s)\\
&s \stackrel{i.e}{\Longrightarrow} insert(u', remove(u, s))\\
&\qquad \Leftarrow \quad u \stackrel{e}{\longrightarrow} u' \ \wedge \ id(u) \equiv i\\
&(s, r) \stackrel{e/a}{\Longrightarrow} (s', r') \quad \Leftarrow \quad s \stackrel{e}{\Longrightarrow} s' \ \wedge \ r \stackrel{e/a}{\longrightarrow} r'\\
&(s, r) \stackrel{e/a}{\Longrightarrow} (s', r') \quad \Leftarrow \quad r \stackrel{e/a}{\longrightarrow} r' \ \wedge \ s \stackrel{a}{\Longrightarrow} s'\\
&(s, r) \stackrel{e}{\Longrightarrow} (s', r') \quad \Leftarrow \quad s \stackrel{e}{\Longrightarrow} s' \ \wedge \ r \stackrel{e}{\longrightarrow} r'\\
&mutex(s, r) \quad \Rightarrow \quad ((u \in s \wedge u' \in s \wedge uses(u) \wedge uses(u'))\\
&\qquad\qquad\qquad\qquad\qquad \Rightarrow \quad id(u) \equiv id(u'))\\
&\Box mutex(env) \quad \Rightarrow \quad mutex(env)\\
&\Box mutex(env) \quad \Rightarrow \quad (env \stackrel{l}{\Longrightarrow} env' \Rightarrow \Box mutex(env'))
\end{aligned}
$$

Invariant to be proved:

$$\Box mutex(\emptyset, free).$$

Introducing composite states:



from: J. Lilius, I.P. Paltor, The Semantics of UML State Machines, TUCS Technical Report 273, Turku Centre for Computer Science 1999

ALPHABET

`vissorts`	$event$
`consorts`	$outer \quad middle \quad A \quad B$
`constructs`	$a, b, c :\to event$
	$\_ :\to event$
`objconstructs`	$S : middle \to outer$
	$S1 : A \times B \to middle$
	$C, H, final :\to middle$
	$D, F :\to A$
	$E, G :\to B$
`dynamic preds`	$\_ \xrightarrow{\_} \_ : outer \times event \times outer$
`vars`	$x : A \quad y : B \quad z : middle$
`Horn axioms`	$S(S1(D, y)) \xrightarrow{a} S(S1(F, y))$
	$S(S1(x, E)) \xrightarrow{b} S(S1(x, G))$
	$S(S1(x, E)) \xrightarrow{a} S(C)$
	$S(C) \xrightarrow{b} S(final)$
	$S(S1(x, y)) \xrightarrow{b} S(H)$
	$S(H) \xrightarrow{c} S(S1(D, E))$
	$S(H) \longrightarrow S(S1(D, E))$

☞ A *sequential* composite state SS is a constructor whose domain sort agrees with the sort of the substates of SS.

☞ A *concurrent* composite state CS is a constructor whose domain sort is the *product* of sorts each of which represents a thread of CS.

Conclusion

☞ Combined translation of class diagrams and state machines into (coalgebraic) swinging types that integrate Horn, equational and modal logics. ☞ In particular: inheritance relationships are reduced to product and sum sorts and their projections resp. injections.

☞ Unique term models are both natural and make UML models amenable to a prover-supported analysis.

☞ Model refinement via signature morphisms *rep* and *rep*-reduct models. ☞ In particular: refinements of static to dynamic predicates and of structural to behavioral equalities.

☞ Other UML concepts to be translated?

☞ Benefit from the translation for detecting inconsistencies and – confusing – redundancies in UML concepts?

☞ *Future work:*

(1) classification of proof obligations for UML models

(2) design of a tailor-made *compile-time* prover complementing the existent *run-time* provers checking model *instances* against given constraints (such as the Dresden OCL compiler)

(3) What may UML gain from the π-calculus or other process calculi? Can all relevant LTS be reduced to sum- or set-sorted destructors?

☞ Who is willing to support projects aiming at (1) and (2)?