# A Representation Framework for Continuous Dynamic Systems

Peter Raulefs
Artificial Intelligence Center
FMC Corporation
1205 Coleman Ave., Box 580
Santa Clara, CA 95052, U.S.A.

*Abstract.* By extending ensemble theory, we obtain a mathematical foundation for representing and reasoning about dynamic systems with continuous objects, such as liquids, and continuous processes, such as chemical reactions. This facility is embedded into the DREAM representation framework that, using object-oriented mechanisms, integrates a variety of representation approaches.

## 1. Introduction

We consider the problem of representing and reasoning about *continuous phenomena* in physical systems, such as the operation of a chemical plant schematically indicated in Fig. 1-1.

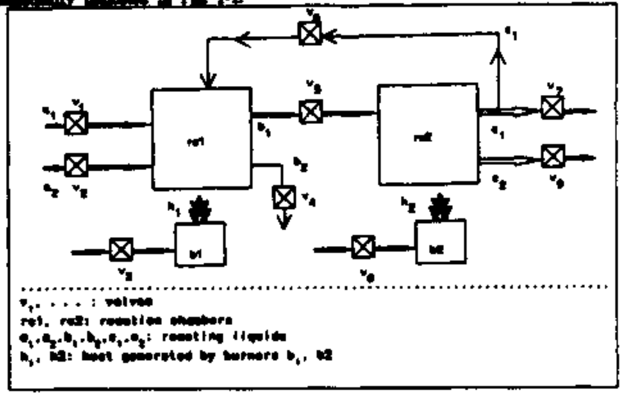
$v_1, \ldots$ : valves
re1, re2: reaction chambers
$a_1, a_2, b_1, b_2, c_1, c_2$: reacting liquids
$h_1$, h2: heat generated by burners $b_1$, b2

Figure 1-1: Chemical Plant Example

For continuous phenomena, we focus on two aspects. First, *continuous decomposability of objects and activities* is a property of many objects, such as liquids and gases and their reactions. Second, *continuous variability of properties*, such as temperatures, pressures and locations, reflect both continuous state spaces and continuity of actions causing state transitions.

Schmolze [7] has observed that previous representation theories do not describe how continuous physical objects are composed of parts. Schmolze describes such parts as consisting of "granules" that can be ascribed properties used for inferring properties of the whole from those of the parts.

We propose to found a representation theory for such phenomena on Bunt's ensemble theory [3, 4]. Ensemble theory extends set theory by introducing ensembles whose parts are ensembles (i.e., of the same 'type', unlike elements of sets). An ensemble is continuous iff it has proper continuous parts. One can describe objects with continuous *and* discrete parts, such as (liquid) water with pieces of melting ice. Another example is fine-grained sugar that is sometimes viewed as a liquid, e.g. when flowing through a pipe, and sometimes as a collection of discrete pieces, e.g. when discussing the grain size; both are even related, as grain size determines flow properties. Ensemble theory with the extensions of this paper provide a foundation to extend set theory, and to account for phenomena difficult to describe in terms of 'granules', unless granules are simply taken to be ensembles.

We apply these concepts to the representation framework DREAM ('Dynamic REAsoning and Modeling'). DREAM represents physical objects as agents denoting continuous, discrete or mixed ensembles. After extending ensemble theory with notions of continuous streams and merges, we show how such mathematical concepts can be transferred into DREAM. DREAM uses mechanisms from abstract data type specification and object-oriented languages, taking objects to be carriers of information (parts, states) *together* with the facilities to manipulate such information (time, actions, inferences), thus combining the terminological and assertional levels [2]. Processes are temporally extended patterns of situations and events describing dynamic changes.

## 2. The DREAM Representation Framework

### 2.1 Agents

DREAM represents objects as agents, adopting an object-oriented viewpoint where changes are brought about by agents carrying out operations. An operation may change the local state of an agent, and communicate information stimulating other agents to execute operations. This information *flow* represents cause-effect relationships, and *causal pathways* if it proceeds along specific routes. States comprise the information that, together with operation specifications, determine an agent's behavior. The behavior of an agent changes because its state changes. Operations quantize all potential activity of an agent.

From object-oriented computation, we also adopt the notion of classes, or types of agents with identical specifications of operations and state spaces. Agents of the same type may differ not only in state, but must differ in properties that remain unchanged as long as they exist, establishing an agent as an *individual*. A simple and efficient *individuation property* is a unique name. Type-wide constant properties and individuation concepts are specified in a CHARACTER clause.

For example, agents of type 'valve' (Fig. 2-1) have a constant maximal opening area, and receive individuality by names. States are described by bindings of state variables constrained by properties in a WHERE-clause. For any constant or variable x, x:D indicates that x may only assume values in a dimension D defined elsewhere.

Operations are specified in terms of the following clauses (we indicate clauses by capitalized keywords): A PRECONDition is a proposition on states defining when the operation is applicable. The PRECONDition needs not be true after the operation has begun execution. A 'Transition INVariant' describes changes when the operation is being carried out, specifying conditions that persist as long as the operation is being executed. A TERMination CONDition is a proposition about states that, when it becomes true, immediately terminates execution. A POSTcondition describes resulting states as far as they are not implied by the other clauses. DURation provides information about how much time the operation spends on executing, especially when this depends on dynamically changing conditions. The PARTS-clause establishes a part-subpart hierarchy. Agents may be dynamically CREATEd, as in a chemical reaction. Information is communicated via typed I/O-ports.

To describe the operations of 'valve', we need only three of these clauses. A valve can only 'open' when its prior state is 'closed'. The first transition invariant takes the discrete, or qualitative part of the temporal derivative (we write $\partial$ for d/dt) and sets it to -1 to assert that the opening widens. State constraint (3) implies that the valve is in state 'opened' *immediately* after the operation 'open' has begun, even before it has terminated. Operation 'flow' is thereby enabled if the inport 'in' is not $\phi$ (i.e., empty). As agents will only execute operations when they are being told, the valve must tell itself to let the liquid flow, and it does so in the CAUSES-clause. 'flow' implies that a decrease of incoming liquid corresponds to an increase of outgoing liquid, expressing that liquid 'flows' through the valve. A physical constraint necessarily terminating the 'open'-operation is 'opening' reaching its maximal value. The specification does also allow an external event 'te' ("terminating event") to terminate the operation, where 'te' is explicitly communicated to the 'valve'-agent when being instructed to 'open'.

### 2.2 Elements of Change: Time, Situations and Events

We represent changes in a system as *events* that denote state transitions brought about by executing operations over time intervals. Situations are a dual notion describing the persistence of states over time intervals. Both rely on a temporal logic.

We briefly introduce those concepts of the dynamic temporal logic DTIL [6] that are subsequently needed. DTIL embeds the time interval logic TIL into a dynamic logic that connects time intervals with situations and events. TIL is a "temporal framework" extending an arbitrary first-order logic L=(T,F,R) of terms T, formulae F, and rules R inductively with interval terms TI and interval formulae FI, where

(1) (Intv, $\leq$) is an infinite, partially ordered set of time intervals.
(2) Intv$\subseteq$TI (intervals are the simplest interval terms).
(3) $\forall$i,j$\in$TI. i;j, (i,j), (i,j], beg(i), end(i)$\in$TI.
(4) $\forall$i,j$\in$TI. i|j, i$\leq$j, i=j, [illegible] $\in$ FI.

The statement below creates the 'valve'-agent v2, whose in- and out-ports are then connected to pipes p1 and p2.

```
AGENT-TYPE valve =
 CHARACTER        CONST    $max-opening:Area
                  INDIVID  name:Name

 STATES           closed,opened:Bool, opening:Area
                  WHERE (1) 0 ≤ opening ≤ $max-opening,
                        (2) opening = 0 <=> closed,
                        (3) opening>0 <=> opened
 INPORT  in:Liquid
 OUTPORT out:Liquid

 OPN    close =
        PRECOND  opened
        TINV     ∂opening' =_c -1
        TERMCOND closed'

 OPN    open(to:Event) =
        PRECOND  closed
        TINV     ∂opening' =_c +1 AND
                 ¬(in' = ϕ) => ∂in' = -1
        CAUSES   flow
        TERMCOND opening' = $max-opening OR to

 OPN    open-further(to:Event) =
        PRECOND  opened AND opening < $max-opening
        TINV     ∂opening' =_c +1 AND
                 ∂²in' =_c -1
        TERMCOND (opening' = $max-opening OR to) AND opening' > opening

 OPN    flow =
        PRECOND  opened AND ¬(in' = ϕ)
        TINV     ∂in' + ∂out' = 0
        TERMCOND in' = empty OR closed'


AGENT v2 := valve[name = v2, $max-opening = 10.0 sqin]

AGENT conduit3 := [p1 CONNECTS-TO in.v2, out.v2 CONNECTS-TO p2]
```

Figure 2-1: Agent Type 'valve'

i;j is the sequential composition (starts with begin of i, ends with end of j), and (i,j), {i,j} are parallel compositions of time intervals (for {i,j}, i and j must overlap, for (i,j), i and j must not be ≤-comparable). beg(i) and end(i) denote begin and end of i. i|j iff i meets j [1], i||j iff i and j are concurrent (≤-incomparable), $i_\wedge j$ iff i and j are simultaneous (overlapping). TIL axiomatizes intervals to be open. ≤ is a chain-complete partial order so that a time instant, such as beg(i), is the supremum of all intervals k meeting i. $\curlyvee$ is the lattice-theoretic join induced by ≤ with instants as null-elements.

To represent temporal structures, especially for cyclic, recurrent activities, TIL has an interval type extension to the above framework. An interval type IT denotes an infinite supply of intervals all related to the same activity or property. Intervals in IT differ in duration and begins/ends, corresponding to all periods over which an operation may be carried out, or some state property persists. For example, all time periods of commuting to work each weekday morning will be associated with the same interval type IT-commute, although all time intervals of this type are *different* (different days, times of day vary, durations depend on traffic, etc.) A temporal structure is like an interval term, but with some interval types instead of intervals. For example,
T-workday = IT-sleep;IT-breakfast;IT-commute;IT-work;IT-commute;IT-eat

is the temporal structure of workdays, and IT-workday$^*$ that of work weeks. An interval term describing a particular workday is obtained by *admissible instantiations* of intervals for interval types, constrained by the requirement that it produces only admissible interval terms (with proper order-relationships among intervals). In our example, an interval substituting IT-sleep must meet the interval substituted for IT-breakfast.

A situation [s,i] is a proposition saying that state s persists throughout time interval i. An event <s|op|i|s'> is a proposition describing that operation op is carried out throughout time interval i and changes state s into state s'. We consider time not to exist apriori, but as being established by physical change. An event <s|op|i|s'> can be taken to *define* time interval i to begin when op starts executing and to end when op terminates. Two successive events <s|op|i|s'> and <s'|op'|i'|s''> delineate the situation [s',j] where i|j|i' (j begins when i ends, and ends when i' begins).

The inherent relationship between situations, events, and time intervals is exploited by extending the TIL-combinators for intervals and interval types to situations and events:
$[s_0,i_0]$ ; $\langle s_0|op_0|i_1|s_1\rangle$ ; $[s_1,i_2]$ describes that after state $s_0$ persists through interval $i_0$, executing $op_0$ during $i_1$ changes $s_0$ to $s_1$, and $s_1$ then persists until end($i_2$). Such situation/event-structures can provide a complete account of the history of a system. Representing concurrency and simultaneity by marked branching and joining arcs, however, yields *history networks* rather than sequences of situations and events.

## 2.3 Processes

A process specifies a temporally extended pattern of activity performed by a system of agents. Executing a process generates a history network of situations and events that result from executing operations at the times and in the order specified by the process. The key concepts to describe processes are:
**(1) Causal rules.** A causal rule relates classes of situations and events. As an event always terminates in a situation, the notion of events causing situations is already inherent in events. The other two types of causation exhibit different mechanisms:

- A causation rule <situation-pattern> → <event-pattern> denotes *spontaneous causation*, where just the occurrence of a situation causes an event.
- A causation rule <event-pattern> → <event-pattern> denotes *transmitted causation*, where executing an operation causes that of another one.

**(2) Situation/event structures and processes.** Although chaining causal rules generates history networks, causal rules do not describe properties of extended activity patterns explicitly. DREAM connects *situation/event-patterns* with causal links to represent situation/event structures, or *processes*. Processes document properties, constraints and contingencies in the assertions of situations and events. Often, a process can be collapsed into a single event or situation summarizing the overall effect at a higher level of abstraction, allowing to build abstraction hierarchies in process descriptions.

# 3. Ensembles: Mathematics for Continuous Objects

For underlying mathematical concepts, ordinary set theory is inadequate to explain phenomena related to objects with continuous parts. Bunt [3, 4] has developed *ensemble theory* to account for such concepts. The representation theory developed in this paper applies an extension (sect. 3.2 - 3.5) of Bunt's ensemble theory (sect. 3.1).

## 3.1 Basic Ensemble Theory

Basic notions of ensemble theory are (x,y,... stand for ensembles):

1. **Part-whole relation** ⊆. x⊆y denotes 'x is part of y'. ⊆ is transitive, antisymmetric, and reflexive. There is exactly one ensemble, the empty ensemble ϕ, that is part of every ensemble.
   *Notation.* ⊂ denotes the 'non-empty part' relation. A proper non-empty part of an ensemble is also called a genuine part.
2. **Merge and overlap functions** ∪, ∩. For a collection C of ensembles, the merge ∪(C) is the least ensemble s.t. every non-ϕ part of ∪(C) has at least one part that is also a part of some member of C, i.e. ∪(C) is the minimal ensemble that has all members of C as its parts. The overlap ∩(C) of C is the least ensemble containing all common parts of all members of C.
   For an ensemble X, and a part x⊆X, $x^c$ is the completion of x w.r.t. X, defined as the minimal ensemble with x∪ $x^c$ = X.
   For an ensemble X, (C(X),∪ ,∩ ,$^c$,ϕ,X) is a Boolean algebra, where C(X) is the set of all parts in X.
3. **Atomic ensembles and the unicle-whole relation** $\underline{\in}$. A non-empty ensemble is atomic iff it has no other parts than ϕ and itself. For an atomic object x, there is a unique non-ϕ part of x, called the unicle of x. The relationship between an atomic ensemble and its unicle is called the unicle-whole relation $\underline{\in}$:
   $\forall x(\text{atom}(x)\rightarrow\exists_1 y. y\,\underline{\in}\, x)$, and
   $\forall x(\neg\,\text{atom}(x)\rightarrow\neg\exists\, y. y\,\underline{\in}\, x)$.
   The unicle-whole relation admits an extensionality axiom stipulating that x∈y iff all parts of x are parts of y and, if x is atomic, the unicle of x is the unicle of some part of y.
4. **Member-whole relation** ∈. x∈y iff x is the unicle of some part of y. Hence the unicle of an ensemble x is a member of x, relating ensemble to set theory.
5. **Continuous ensembles.** A non-empty ensemble x is continuous if every genuine part of x has a proper genuine part. Hence, a continuous ensemble cannot have any minimal parts, and continuous ensembles have no members.
6. **Discrete ensembles.** An ensemble is discrete iff it is equal to the merge of its atomic parts. Clearly, the empty ensemble ϕ is discrete.

Continuous ensembles contain no atomic parts, and discrete ensembles consist only of atomic parts (if any). Ensembles that are neither continuous nor discrete are called mixed ensembles. Every mixed ensemble can be decomposed into a discrete and a continuous part.

## 3.2 Continuous Merge

Consider merging two continuous ensembles w of water W and v of vodka V to form an ensemble d = w∪v of diluted booze D. By our definition, each part of d contains parts w', v'⊆d of water and vodka. But in reality, there is no part of 'pure' water or vodka in d, where a pure part of water is one that has no part of vodka. As this is not guaranteed by the above merge operation, we capture this concept in a specialization:

Definition 1: [Continuous Merge]

The continuous merge of two continuous ensembles x:X and y:Y is the least ensemble z with

(*) $\forall u\subseteq z\,\exists w\subseteq x, v\subseteq y.\, u = w \uplus v.$

We denote the set of all ensembles obtained by continuously merging ensembles from X and Y as $X \uplus Y$, i.e. $z{:}Z$ with $Z = X \uplus Y$.

*Note.* We omit the proof that z in (*) is unique (i.e., $\uplus$ is well-defined).

An important application of continuous merges is establishing a 'closeness' constraint for a chemical reaction by having liquid or gaseous materials merge continuously to react.

### 3.3 States

How can we describe the temperature distribution in a lake in summer? This is the phenomenon that, especially for deep lakes, the bottom region is much cooler than regions closer to the surface, and there is usually a gradual increase of temperatures from bottom to surface, unless disturbed by underwater streams. Likewise, two continuously merged materials may not start reacting unless their temperatures exceeds some minimal value. In such cases, we would want to ascribe a 'temperature'-property to ensembles that may change over time.

**Definition 2:** [States of ensembles]

A *state* of an ensemble is a property that may change its value over time. We use two specific notions of states:

- A **propositional state** is a proposition about an ensemble.
- A **first-order state** is a function from state variables to values in a state space.

*Notation.* For an ensemble e, state(e) denotes its state. If we describe first-order states, we write $x.e = v$ for the fact that e is in a state where state-variable x assumes the value v. For a type X of ensembles, State(X) is the state space of its members.

Applying a notion of states to ensembles requires a definition of how states of parts relate to the whole ensemble.

**Definition 3:** [State combination function]

For a type X of continuous ensembles, a **state combination function** $co_X$ maps the state of the parts of partitions of any X-ensemble e to the state of e:

$co_X$ is a state combination function for ensemble type X iff for any partitioning $e_I$ of an ensemble e:X,

(1) $\mathrm{state}(e) = co_X(\{\mathrm{state}(e_i) \mid i \in I\})$, and

(2) for any X-ensemble e', $\quad \mathrm{state}(e \uplus e') = co_X(\{\mathrm{state}(e), \mathrm{state}(e')\})$.

{$co_X$ combines the states of all partitions to the state of the whole, and exchanging any partition with another part having the same state will not change the state of the whole (substitutivity property of states). Clearly, the combined state of an ensemble is independent of the chosen partitioning}}.

*Example.* For ensembles w:W of water, let state(w) be a function that assigns numbers between 0 and 100 to the state variable 'temp'. Then, for any partitioning $P(w) = \{w_1, \ldots, w_n\}$ of w into water ensembles, a useful state combination function would determine the average temperature of w from those of the packages it consists of:

$$co_X(\{t_1, \ldots, t_n\}) = \Sigma_{i=1}^{n} \frac{\mathrm{mass}(w_i) \times t_i}{M},$$

where $M = \mathrm{mass}(w)$. $co_X$ would, however, not be a state combination function if it took just the maximum element of its argument set.

### 3.4 Streams

Imposing structure on continuous ensembles requires making parts distinguishable without relinquishing continuity. Ascribing states to parts accomplishes just that. Given a way of distinguishing parts, one can impose order among them. For studying concepts of flow, we need a notion of streams as specialized partitions. A partition of an ensemble e is a set p so that each element of it is an ensemble of e-parts with

(1) $\forall x,y \in p. \cap(x,y) = \phi$, and

(2) $\cup(p) = e$.

The elements of p are the blocks of the partition.

**Definition 4:** A **stream partition** of an ensemble e is a partition of e together with a linear order of its blocks. Two stream partitions $(p_1,<_1)$ and $(p_2,<_2)$ of an ensemble e are compatible iff, for i=1,2,

```
∀x_i,y_i ∈ p_i. IF x_1 <_1 y_1 and x_2 <_2 y_2
          THEN for U_2(x_1) :⇔ {u ∈ p_2 | ∃ z⊆x_1.z⊆u}, and
               for U_2(y_1) :⇔ {u ∈ p_2 | ∃ z⊆y_1.z ∈ u}:
          ∀u ∈ U_2(x_1), v ∈ U_2(y_1).u ≤_2 v.
```

{$U_2(x_1)$ is the set of all $p_2$-blocks containing parts of $x_1$ and $U_2(y_1)$ is the set of all $p_2$-blocks containing parts of $y_1$}

The **stream product** $p = p_1 {*} p_2$ of two compatible stream partitions $(p_1,<_1)$ and $(p_2,<_2)$ of an ensemble e is the coarsest stream partition of e that refines both, as illustrated in Fig. 3-1. If a stream partition $(p_1,<_1)$ of an ensemble e refines a stream partition $(p_2,<_2)$ of e, then both are compatible and $p_1 {*} p_2 = p_1$. Stream partitions "chop" ensembles into pieces. This appears to defy intuitive notions of liquid streams, where any such "chopping" is artificially imposed, yet still part of the concept of a stream. By relativizing stream partitions, we obtain continuous streams:

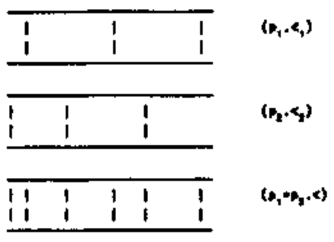

Figure 3-1: Stream Product

**Definition 5:** A continuous ensemble e is a **continuous stream** iff each stream partition of e has a proper refinement. A set P(e) of stream partitions of e so that each stream partition in P(e) has a proper refinement in P(e) is a **continuous stream refinement system** for e.

Every continuous stream refinement system provides an infinite supply of arbitrarily refined stream partitions. This is the facility we need for merging continuous streams:

The **smerge** $e = e_1 \oplus e_2$ of two continuous streams $e_1$ and $e_2$ is the least ensemble e s.t. for all continuous stream refinement systems $P(e_1)$ and $P(e_2)$, there is a continuous stream refinement system P(e) with

$\forall p_1 \in P(e_1), p_2 \in P(e_2) \exists p \in P(e).$

(1) $\forall x \in p.\ \exists x_1 \in p_1, x_2 \in p_2.\ x_1 \subseteq x \wedge x_2 \subseteq x.$

{each p-block contains parts that are $p_1$- and $p_2$-blocks}

(2) $\forall x_1,y_1 \in p_1, x_2,y_2 \in p_2.$
$(x_1 <_1 y_1 \wedge x_2 <_2 y_2) \rightarrow$
$\exists x,y \in p.\ x<y \wedge x_1,x_2 \subseteq x \wedge y_1,y_2 \subseteq y.$

{blocks in p preserve the orders on parts from $p_1$ and $p_2$}

**Examples**

1. **Metal alloy bar.** b is a cylindrical brass bar, made of copper and zinc, where one end is pure copper, the other pure zinc, and b is a "continuous" Cu/Zn-alloy in between. An obvious approach is describing b with a function $f_{Cu}$ giving the differential Cu-concentration at distance r from its left end, where the rest is zinc:

$df_{Cu}(r)/dr$ = const. with $f_{Cu}(0)=0$, $f_{Cu}(r_0)=100\%$.

We represent b as a continuous stream, where every continuous stream refinement system P(b) has the property

$\forall p \in P(b). \forall x,y \in p.\ x<y \Rightarrow r.x<r.y$
$\qquad \forall x,y \in p.\ x<y \Rightarrow \%Cu.x < \%Cu.y$

2. **Merging water and alcohol streams.** Consider merging two continuous streams w of water and a of alcohol to form a mixture $m = w \oplus a$. We ascribe any part of a liquid a state that contains values for temperature and volume; each part of m has the additional state variable %W (water contents). Then, the smerge of continuous water and alcohol streams *requires* a definition for determining the states of m-parts from those of the smerged w- and a-parts.

# 4. Representation of Continuous Objects

### 4.1 Example: Liquids

To describe liquids, we assume a simple view by saying that an object is considered a liquid iff its temperature ranges between a freezing and a boiling temperature that depend on pressures only, its volume depends on its temperature, and unless it is in a "stable" state, it flows until it becomes stable:

A Liquid0-liquid is characterized by linear dependencies of volume and temperature, and freezing/boiling temperatures and pressure. Its states consist of volume, pressure and temperature, constrained as indicated. It has Liquid0-parts that can be partitioned into Liquid0-parts with an obvious state-combination function. It is only stable when all of its parts are stable.

self is a self-reference of any agent of type Liquid0. For brevity, POSTconditions contain some meta-logical notation: (1) ∧ (2) ... stands for the formula obtained by ANDing the indicated formulas, and {u/self} denotes a substitution in the following formula. By its PARTS-clause, a Liquid0-agent is composed of Liquid0-agents consisting of Liquid0-agents, etc., with the state composition function defined in the WHERE-clause. As continuous agents can be partitioned in infinitely many ways, the state composition function must be independent of any particular partitioning. This is obviously accomplished in the WHERE-clause, where Partition(self) denotes the infinite set of all partitions of a non-$\phi$ Liquid0-agent:

$X \in \mathrm{Partition}(u) :\Longleftrightarrow \exists I \subseteq N. X = \{x_i \mid i \in I\},$

and for any permutation g of I, $u = \ldots\mathrm{merge}.\mathrm{merge}.x_{g(1)}(x_{g(2)})\ldots$

A 'piece' of liquid is stable, and thereby not flowing, if all its parts are stable.

The operations are specified in a first-order language that includes part-of and equality relations representing the corresponding relations of ensemble theory:

$\forall x,y{:}\mathrm{Liquid0}.\quad x \subseteq y :\Longleftrightarrow \exists z{:}\mathrm{Liquid0}.\ \mathrm{merge}(z) = y\ \wedge$
$\qquad\qquad x = y :\Longleftrightarrow x \subseteq y \wedge y \subseteq x$

Both relations are well-defined for Liquid0-agents because 'merge' is defined inductively. Obviously, $\forall x{:}\mathrm{Liquid0}.\ \mathrm{merge}.\phi(x) = \mathrm{merge}.x(\phi) = x$, and $\subseteq$ is a partial order. POSTconditions (4) for merge, and (2) for overlap ensure minimality, and thereby single-valuedness of these operations. By the above equality relation, two LiquidO-agents x and y are equal, x = y, iff they are identical.

We consider two specializations of LiquidO: Superfluid liquids that *always flow*, and drinkable liquids. As the semantics of an agent type is that of an abstract data type denoting a class of algebras (or, equivalently, a theory), we adopt the notion of data type *combinations* for specializing an agent type:

```
AGENT-TYPE LiquidS = Liquid0 + [STATE WHERE stable = false]

AGENT-TYPE LiquidD = Liquid0 RELATIVE-TO Human
                     WITH ∀h:Human.[alive.h]drink.h(:self)[alive.h']
```

LiquidD-agents are considered drinkable iff any human drinking it stays alive. This is a different kind of specialization as restricting the state space in LiquidS, as LiquidD-agents are constrained by their environment. Most specializations of LiquidO will specify substances of particular chemical composition and associated properties:

```
AGENT-TYPE Water = Liquid0 +
  CHARACTER      INDIVID chemcomp = H2O
                 CONST   name:Name ∧
                         $extent = "function describing how volume depends
                                 on temperature" ∧
                         $freeze-temp = 0oC ∧
                         $boil-temp = 100oC
                         ... etc. ...
```

```
A particular water-agent is obtained by instantiating as yet unbound
$-prefixed state identifiers, and specifying an initial state:
AGENT w = Water[$name=pool-4;STATE vol=60cuft, temp=70o,
                                press=1atm, stable]
```

```
AGENT-TYPE Liquid0 =
  CHARACTER      CONST   $extent:Temperature -> Volume IS linear,
                         $freeze-temp, $boil-temp:Pressure -> Temperature
                                                    IS linear

  STATE vol:Volume, press:Pressure, stable:Bool, temp:Temperature, mass:Mass
        WHERE vol=$extent(temp) ∧
              (∂temp=0 → ∂vol=0) ∧
              $freeze-temp(press)<temp<$boil-temp(press)  ∧
              vol=0 ∨ mass=0 → :self=ϕ

  PARTS :Liquid0
        WHERE ∀X ∈ Partition(self).

              vol = Σ{vol.x|x∈X} ∧

              temp = (Σ{mass.x × temp.x|x∈X})/mass.x ∧
              stable <=> ∀x∈X. stable.x

  OPN   flow
        PRE  ¬stable
        TERM stable


  OPN   merge(x:Liquid0)
        POST (I) :self=ϕ → :self'=x ∧
             (II)       (1) ∀x⊊self'.∃y⊊x.y⊊x ∧
                        (2) ∀x⊊x.∃y⊊self'.y⊊x ∧
                        (3) x' ⊊ :self' ∧
                        (4) ∀u:Liquid0. [u/self][(1) ∧ (2) ∧ (3)]
                            → u = :self'


  OPN   overlap(x:Liquid0)
        POST (1) ∀x⊊self'.x⊊self ∧ x⊊x
             (2) ∀u:Liquid0. [u/self](1) → u = :self'


  OPN   separate(<predicate>, n:Name)
        PRE     ∃x⊊:self. <predicate>(x)
        POST    (1) ∀x⊊:self.<predicate>(x)  → ¬x⊊:self' ∧
                (2) ∃u:Liquid0.name.u'=n ∧
                                ∀x⊊:self'.<predicate>(x) <=> x⊊u
```

**Individuation**

The above discussion gives an example of a combined specialization/instantiation/parts-hierarchy (see Fig. 4-1). As these notions were adopted from similar ideas in object-oriented data type specification languages, we omit all further details about inheritance, encapsulation, specialization, and instantiation. A crucial feature of this approach is that it provides a *framework* for integrating vastly different concepts and styles.

Individuation is a procedure identifying entities such as concepts (i.e., agent types) and objects (agents) as unique individuals distinguishable from others. Individuation includes an equality notion, provides a decision procedure for equality, and a choice mechanism for picking one out of a collection of entities. Approaches to individuation supported in DREAM are:
(1) *Individuation by intrinsic property.* 'Water' is individuated from other liquids by its chemical composition.The simplest way is giving objects unique intrinsic names (not references). DREAM supports this by the INDIVIDuation clause in the CHARACTER clause.
(2) *Conceptual individuation.* LiquidO is identifiable by its character, state space, parts, and operations. In general, two agent-types are equal iff they denote isomorphic algebras. This is often undecidable.
(3) *Referential individuation* consists of someone attaching a unique name to an entity. The difference to (1) and (2) is that someone else provides the individuating information.

The discussion of Hayes' car [5] shows the importance of being aware of the kind of individuation one is talking about: The identity of the car is threatened by gradually exchanging all its parts, including registration numbers. Individuation that is conceptual or by intrinsic property would distinguish two different cars. "X's" car is identified by referential individuation with the reference being maintained by a name. Hayes' requirement of spatiotemporal continuity of conceptual or intrinsic-property individuation is no solution: Under these equality relations, the original car and the 'same' car after exchanging all its parts are different. However, the *reference* remains the same.

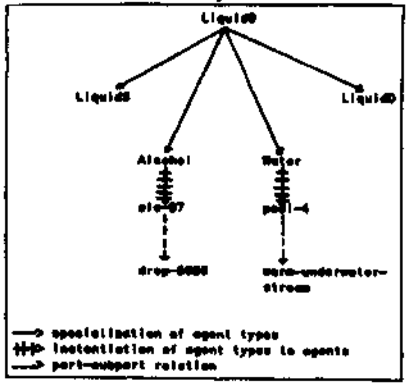

Figure 4-1: Specialization/Instantiation/Parts-Hierarchy

## 4.2 Streams

We illustrate the DREAM-representation of the stream concepts developed in sect. 3 in terms of specializations of LiquidO:

Partitions of an agent l:LiquidO are agents of type

```
AGENT-TYPE Liquid0-partition(l) =
  CHARACTER      CONST I:N-set
  PARTS          l_i:Liquid0
                 WHERE i∈I ∧  l = merge{l_i|i∈I}
```

Any partition of l is characterized by its parts and an index set I of natural numbers that enumerates its blocks. merge$(l_i|i \in I)$ stands for . . . merge.merge.$l_{i1}(l_{i2})$ . . . , using the fact that merge.x(y) = merge.y(x).

Stream partitions of an agent l:LiquidO are similar to partition-agents for l with the addition that the linear order on index sets imposes a linear order on its parts.

Continuous streams are specified by transferring our definition into proper DREAM-syntax to specify PARTS-constraints and an smerge-operation.

*Acknowledgement.* I would like to thank N. S. Sridharan for helpful comments on an earlier draft of this paper.

## References

[1] Allen, J.F.
Maintaining knowledge about temporal intervals.
*Communications of the ACM* 26(11):832-843, 1983.

[2] Brachman, R.J., V.P. Gilbert, H.J. Levesque.
An essential hybrid reasoning system: Knowledge and symbol level accounts of KRYPTON.
In *Proc. 9th IJCAI*, pages 532-539. IJCAI, Los Angeles, CA, August, 1985.

[3] Bunt, H.C.
*Mass terms and model-theoretic semantics.*
Cambridge University Press, Cambridge, England, 1986.

[4] Bunt, H.C.
The formal representation of (quasi-) continuous concepts.
In J.R. Hobbs, R.C. Moore (editor), *Formal Theories of the Commonsense World*, chapter 2, pages 37-70. Ablex Publ. Co., Norwood, N.J., 1986.

[5] Hayes, P. J.
Naive Physics I: Ontology for Liquids.
In J.R. Hobbs, R.C. Moore (editor), *Formal Theories of the Commonsense World*, chapter 3, pages 71-108. Ablex Publ. Co., Norwood, N.J., 1985.

[6] Raulefs, P.
A dynamic logic for temporal reasoning about dynamic systems.
*in preparation* , 1987.

[7] Schmolze, J.
Physics for robots.
In *Proc. 5th Nat'l Conf. on Artificial Intelligence*, pages 44-50. AAAI, Philadelphia, PA, August, 1986.