Cantor: a Tutorial and a User's Guide

(prototyping, set theory and all that)

Jean-Pierre Keller

volume I

Kepler *8 rue des haies, F-75020 Paris* *dec. 1994*

Cantor: a Tutorial and a User's Guide

(prototyping, set theory and all that)

Jean-Pierre Keller

Kepler

8 rue des haies, F-75020 Paris

> what is prototyping ?
deliver (i.e. give life and shape to) an abstraction
> why set theory?
because all the abstractions we may think of have models in set theory
> and what is all that?
that will be discussed now

Volume I: a Cantor Tutorial

Table of Contents

Volume I: a Cantor Tutorial

Jean-Pierre Keller, Kepler

8 rue des haies, F-75020 Paris

> what is prototyping ?
deliver (i.e. give life and shape to) an abstraction
> why set theory?
because all the abstractions we may think of have models in set theory
> and what is all that?
that will be discussed now

What is - and why - prototyping?

Prototyping is strictly speaking the art of developping prototypes. Prototypes are by definition experimental versions of presumably complex systems. They are developped to help in assessing the intricacies and virtues of projects : it is accepted that beyond very simple or well-known systems an abstract definition, though essential, is insufficient to understand and evaluate what a project represents, what are its consequences : an abstract system may not be understood or evaluated without being 'seen and touched and experimented'. Presented in this way, prototyping is just a cautious way of proceeding with projects and innovation.

This argument could be turned over: why not systematically look at every new idea with the eyes of a 'doctor' or an 'expert' ausculting a prototype? For those who, pushing such new ideas, would'nt be building a prototype, the affirmation of the feasibility and relevance of these ideas? Viewed in this way, prototypes rhyme with audacity, cleverness and seriousness. Ideally, prototypes should be low cost and developped swiftly : this is why often prototyping is assimilated with 'quick and dirty' and 'throw away' developments. Instead we suggest that prototyping is very much like experimenting, that it requires a scientific frame of reference, so that, when the experience comes to an end, valid and arguable conclusions can be made.

What are, in the software technologies, possible scientific frames of reference?

Whenever a craft develops into a method with a specific technology, there is a formal reference frame. Often, companies, laboratories, individuals develop skills which are partly build on established technologies, partly on their own personal experience.
Beyond 'sculpting' and refining with the possible assistance of powerful machinery (tools and established methods), prototyping is an attempt at formalizing the skills and their use of machinery into a method[1]. Such methods are domain dependent and attached to a class of applications. The most elementary examples are the so-called 'application generators'.By analogy, we call 'generic' these experimental methods or tools.

To illustrate this notion of generic tool, let us imagine the case of a company developping software for a specific chip on a custom made board. This software is developped on a standard host, say a Macintosh. There is a need for :
-compîling programs on the Macintosh
-transfering programs compiled on the Macintosh to the board,

[1]Very often these methods represent the integration of a relatively large number of standards, established methods and tools.

-emulating the board on the macintosh for preliminary debugging

The interface software meeting these essential needs is what is usually called an *application generator*. This is the lowest level generic tool. Let us suppose, that this board is a Digital Signal Processing board, for processing pictures in some standard format. A number of specific processing functions -using the hardware configuration ressources of the board- could be defined, and calls to these functions made possible, in the application generator. This extension of the interface software would be a more advanced generic tool, tailored to the DSP needs and the specific board. One could imagine many extensions expanding the capability of this interface software, but remaining *generic*, i.e. suited for a wide class of applications.

We picture the prototype production process as the 'pure prototyping' feed-back loop (cf. fig. 1):

-methods are represented by generic tools,

-prototypes are models obtained by application of these tools

Deficiencies in prototypes may be the result either of an inadequate application definition and/or parametrization of the available methods, or of a deficiency of the methods themselves. Developping prototypes therefore entails a progressive improvement of the methods' adequacy.

'Pure prototyping' instead of ad'hoc developments, means the adaptation and systematic use of generic tools. Thus prototyping becomes synonymous with capitalisation of know-how!

This requires therefore the ability of using existing formal methods as well as developping new ones. Whence prototyping tools are either powerful formal generic tools or tools for creating 'easily' new ones. It is for the sake of modeling complex abstractions and creating new formal methods, that set theory, the most expressive and general-purpose form of mathematics has been selected as the mathematical foundation for a software prototyping environment.

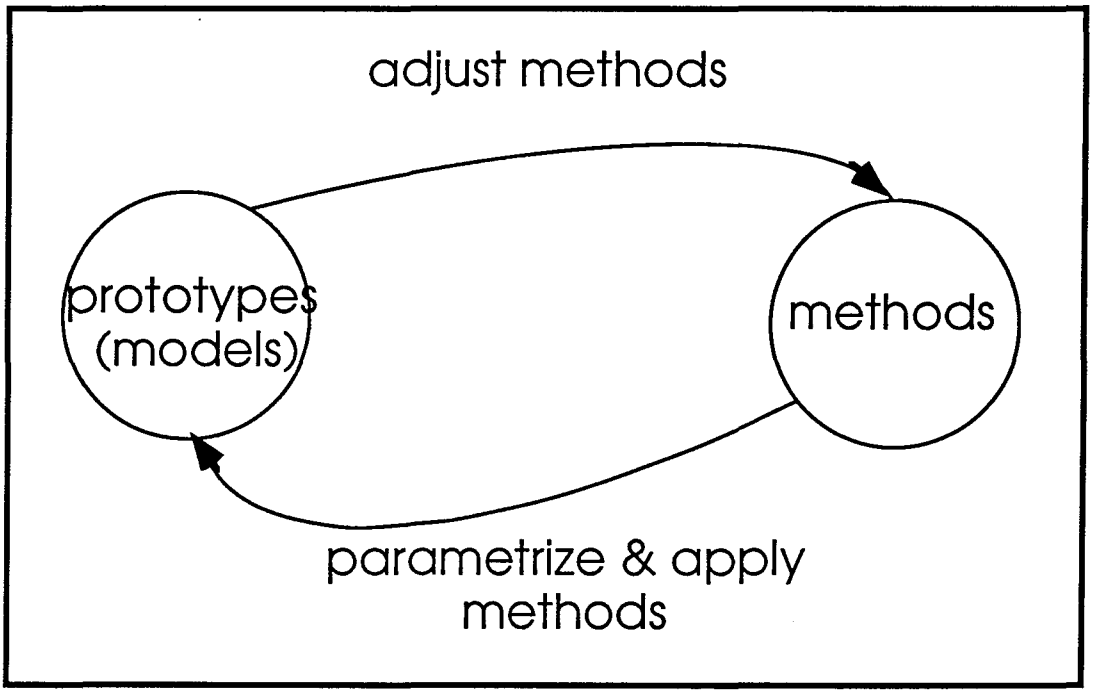


fig. 1 the 'pure prototyping' feed-back loop

Why sets and set theory?

Imagine an old man, late at night, on a sidewalk under the only street light of the block, watching carefully the water running in the curb. A neighbour, passing by recognizes the man: "what are you doing here so late in that dark street? -I am looking for my watch! -what happened to your watch? - It's lost, may be it slipped into the curb, will be pushed by the water and will show up here sometimes. -Where did you lose your watch? -Well, two blocks away, near the fruit-store... - why then look for it here? - you see, the other street is dark, there is no street light!"

This story describes a common situation when dealing with the search for a convenient and

appropriate solution for specific problems : you don't need to be an expert to recognize what is essential and what is secondary in a proposed solution, and that, when you need a watch, a watch-less street is just as good as a screw driver, a french english dictionnary or a street light, and is not the solution. In the programming world, you don't always need to be an expert to recognize whether a given data representation is adequate or not, even though it displays some relevant parts. We are not discussing at this point optimisation, just basic adequacy.

Often, programming problems deal with collections. Not all collection models will be adequate to represent the collection(s) you have to deal with. However, sets represent the most flexible known model for collections (all the imaginable models derive from that one; necessarily one of its derived model will adapt to your problem).

Set theory is an intricate mathematical theory, which will not be discussed here. Actually Cantor, has a bias toward a specific branch of it : hereditarily finite set theory. But this is beyond our point now: we are only concerned with providing a correct and rich basis for a simple, rigorous and clear formulation for a very wide class of abstract data models and their algorithms. Our objective now is to introduce the basic set theoretic data representations, and discuss their applicability.

We are now presenting a set of examples which should be seen as a gradual introduction to Cantor and its set-oriented constructs. The companion volume "a Cantor User's Guide" is refered here as the Cantor manual, since it contains a systematic presentation of Cantor essential features, as well as elemntary examples and illustration exercises : it is inseparable from this tutorial.

In the first example the most common set-oriented constructs are introduced: set, tuples and maps, including maps associating ordinary data-structures to user-defined functions. In the second example we introduce tabulation, a simple technique to speed-up many computations. In the next example, we discuss data representations in Cantor and compare them with the relational model -probably nowadays the most widespread representation of large data sets and databases. In the fourth example, we consider simple graph problems, and present elements of an algorithm design methodology based on fixed-points, finite differencing and a simple complexity model for a set machine. Finally, in the last example, we use Cantor constructs to represent unusual set organisations (hypergraphs) and their use in modeling real-time systems. We have not included other topics, quite relevant both to Software Engineering and set abstraction, like parsing and compiling, graphic representation of abstract structures (the converse of what is done here in example 5), data-flow analysis : these topics are perhaps too advanced for this modest tutorial introduction to the interplay of (set-oriented) mathematics, prototyping and programming. Indeed, as was stated in our introduction to prototyping[1], other branches of mathematics and other established methods would have to be integrated in the exposition of these subjects.

Example 1: a line at a bank or at the post office

This example deals with two collections:
1- the collection of customers, waiting on a line
2- the collection of service counters
The two collections are not organized along the same principles. The waiting line obeys a FIFO (First-In-First-Out) discipline : one joins the line at the tail, one is served at the head, one could quit the line 'in the middle' -this shortens the line- but one cannot join the line in the middle. This is typical of an ordered collection. The counters are organized along very different principles: a customer at the head of the waiting line is directed to any open and free (not busy) counter. If there is a 'counter discipline' it has nothing to do with the 'line discipline': it is based on availbility not on order.
To model this example one needs to formalize the following operations:

-line operations: joining a line, quitting a line 'in the middle' (elsewhere than at the head), quitting a line at the head,

-set operations on service counters: defining a subset by criterias (here 'available', 'busy','not operating', etc), selecting an arbitrary element in one of the subsets

Example 2: function tables

The table is a number table, for a function defined by a recurrence equation. This the case of the factorial function

$n! = n * (n-1)!$

or of the Fibonacci numbers defined by:

$F_0 = 0\ F_1 = 1\ F_{n+2} = F_{n+1} + F_n$

Computing 10!, involves computing 9!, which involves computing 8! etc. If after having computed 10! the user requests 11!, then again, the same values have to be computed again. It would be simpler to store them all in a function table, and retrieve their values rather then re-computing them, if possible.

Example 3: a data table

The table has four columns: item name, sales price, quantity sold since 1st of the month, quantity in stock. The 'key' of an entry into this table is the item name.

This example deals with a single collection. Each member of the collection may be regarded as a 4-uple (name, price, #sold, #in stock). This collection does'nt have a well defined ordering. It could be presented sorted by item name, or by price, or by quantity sold, or by quantity in stock, or by some other rule, e.g. by cash-flow (price * #sold). What determines its structure is more the fact that it should be seen as a mapping (a function) associating to each item name a 3-ple (price, #sold,#in stock). This model is that of a set of pairs where each pair is [item name, (price, #sold,#in stock)]. Note that the second element of each pair in this model is a 3-ple (price, #sold,#in stock). To actually formalize this model one needs to formalize the following operations:

-representing tuples (pairs, 3-ple, 4-uple, etc..) of various length, with items of many different types
-nesting of ordered sets (e.g. nesting a 3-ple within a pair)
-nesting of ordered sets within unordered sets
-computing the value associated to a 'key'
-accessing the i-th element of a tuple for reading or writing (i.e. update)

A possible visual presentation could be in a table, one row per item:

...			
$itemName_i$	$price_i$	nbr_sold_i	$nbr\_in\_stock_i$
...			

This example could be made more realistic in the following way:
Associated to the item name are not only the informations (price, #sold,#in stock) but for each item a list of suppliers, for each supplier its current price. That is, associated to the item name is a 4-uple (price, #sold,#in stock, supplier_table) where the 4-th item in the 4-uple is itself a table, associating to a supplier, its current price for this item. Since the same supplier may supply different items, one could assume that the same supplier name appears associated to different item names. A possible table presentation including nested supplier information tables could be:

...					
$itemName_i$	$price_i$	nbr_sold_i	$nbr\_in\_stock_i$	supplier_a supplier_b	price_a price_b
...					

The same information could be obtained by a different data organization, in which all the supplier prices are grouped in a supplier_price table, one per supplier, and the 4-th item associated to an item name is just a set of suppliers names. In this model we have two main collections, actually sets:

1-an item data table whose visual presentation could be :

...				
$itemName_i$	$price_i$	nbr_sold_i	$nbr\_in\_stock_i$	supplier_a supplier_b ....
...				

2-a supplier table :

....	
supplier_x	... $itemName_i$ $supplier\_price_i$ $itemName_{i+1}$ $supplier\_price_{i+1}$ ....
....	

We see here two solutions which are obviously adequate to the problem description, and for discriminating between them, an expert advice on 'normal forms' may be needed (normal forms are rules for organising data models in the relational approach). But abstractly these two solutions are almost equivalent.

Example 4: pre-requisites

Let us consider the following graph describing the dependencies among the chapters in a text book.

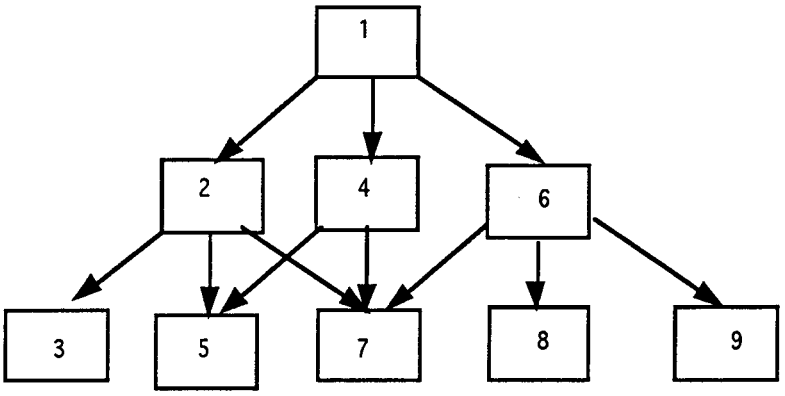


fig. 2 chapters dependency graph in a text book

Each arrow describes a direct dependency. For instance, since two arrows are ending at box 5 (the arrows 2->5, 4->5) this means: chapter 5 depends directly upon chapter 2 and chapter 4 : 2 and 4 are the ancestors of 5. Finding all the direct and indirect dependencies, involves accumulating into a single collection all the ancestry. Thus let us state our problem : determining all the pre-requisites to chapter 7.

Quickly, looking at the graph, one sees that {2,1,4,6} is the pre-requisite set for 7.

This presentation however misses a point: in what order should one read the pre-requisites to chapter 7? The ordering is certainly not (2,1,4,6). However, (1,2,4,6) or (1,6,2,4) seem equivalent.

Example 5: a simple real-time system: a digital watch

This example is inspired by D. Harel's presentation of hypergraphs and statecharts in the

Communication of ACM in 1988[2]. In this paper Harel proposes a formal representation for representing the behaviour of reactive systems, an important class of real-time concurrent systems. He illustrated his discussion with the case of a digital watch. The working of Harel's digital watch is described by the following diagram, which we will explain thereafter below:

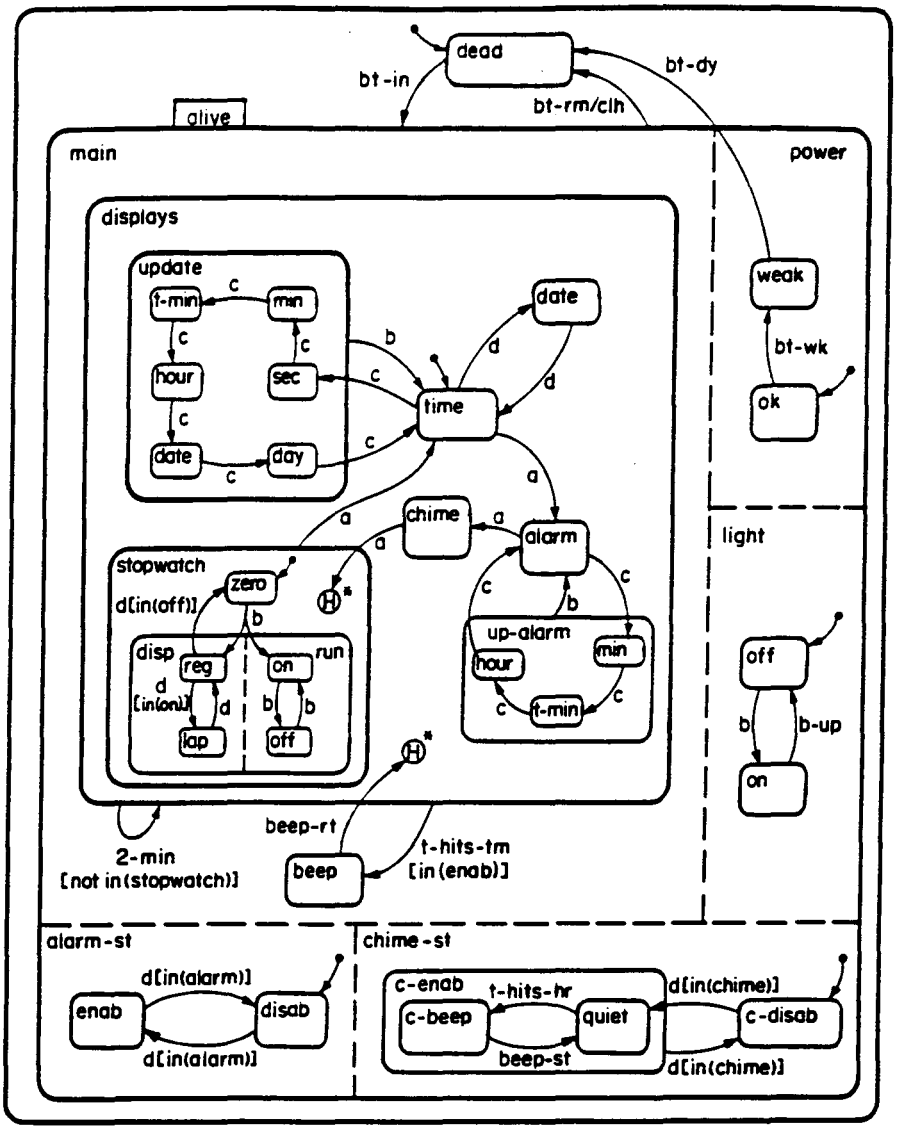


fig. 3 A State Chart for the Digital Watch (reproduced from CACM)

Before describing further this formalism, let us emphasize that (industrial) Software Engineering is using systematically visual formalisms to assist in the specification process. Harel's formalism is perhaps the most intricate one, it is quite widespread and anyway one of the most expressive. This is what motivated our choice.

The key feature of a Statechart, as a diagram, is the 'blob', a labelled rectangle: blobs are the main components of Statecharts; blobs may be nested; 'atomic' blobs contain no other blobs.

There are three main collections in Statecharts:
- the set of blobs, representing the individual states of the system being described, as well as the groupings of such states
- the set of subsystems, representing the breakdown of a blob into concurrent subsystems (a subsystem blob is a block in a partitioned blob; the boundary between two adjacent blocks in a

[2]D. Harel : On visual Formalisms, CACM, 31,5, 1988,pp 514-530.

partitioned blob is a dashed line)
- the transitions (labelled arrows)
Ordering by inclusion is the only natural ordering for the blobs and subsystems : these inclusions are represented as nestings on the Statechart diagram, inspired by Venn Diagrams.

Each of the rectangles is a 'blob' representing a set of states for the system. Nested blobs -within a given blob, at the same nesting depth- corresponds to mutually exclusive states: e.g. the whole system comprises two blobs *dead, alive*; the blob *up-alarm* comprises three mutually exclusive states: *hour, min, t-min.* These states represent different ways of displaying the time information. The same states may appear in another grouping: actually, *hour, min, t-min.* are also states in the *update* blob. Thus even though the time-update and the *alarm-update* states are distinct and mutually exclusive states, they are not disjoint as sets! The blob *alive* is subdivided too. This time, the subparts are separated by dashed lines: the dashed lines create a partition, here the partition is

alive -> main, power, light, alarm-st, chime-st

into five blocks. Each of these blocks may then be further subdivided into states or subsystems. In the example, the decompositions are:

main -> displays, beep
power -> ok, weak
light -> on, off
chime-st -> c-disab, c-enab
alarm-st -> disab, enab

Harel calls such partitions a cartesian product, since each element (state) of the partitionned blob *alive* is itself a collection of elements (states) containing a representative member of each block of the partition. Since 'alive' is partitionned into 5 subsystems, each state of *alive* is represented by its decomposition into 5 states. E.g. :

main	*power*	*light*	*alarm-st*	*chime-st*
displays	ok	off	disab	disab
beep	ok	off	disab	disab

are two examples of valid elements (states) of *alive* with for each one its state decomposition into *alive*'s member blocks. It is sufficient that they differ in only one of the member blocks components (i.e. subsystems) -here in the *main* block- to correspond to different states of *alive.* Note that, the same state (blob) may not appear in different blocks: the blocks are really disjoint sets. The arrows represent event-triggered transitions between states (i.e. between blobs).

The notion of default state is clear: if the device described by the statechart has to activate, for a given blob, any state among all the blob's members, it will select the default state for activation. When is it the device's responsability to activate a state: when a blob, subdivided into several states, is activated, the device has to select which state among its substates should be activated; the default state represents that active state by default. In the example of the watch, the top-level blob which describes the most general condition of the watch is supposed to be always active, it can be either in the *dead* state or in the *alive* state. How is the choice between those two states being made? In the Statechart in fig. 3 by default, the watch is in state *dead*: An arrow coming from a dot within the top-level blob, indicates the default state. Similarly, when the blob *alive* is activated -as a consequence of the *bt-in* event- then the watch device is simultaneously in subsystems *main, power, light, chime-st, alarm-st.* For each of these a decision has to be made: which state should be activated? In the Statechart in fig. 3, by default, the subsystem *power* is in state *ok*, the subsystem *light* is in state *off*, etc.
Notice that in the partition main, there is no default state: the system could arbitrarily be found in *beep* or in *displays* ! The same event, e.g. *b*, may occur in various subsystems (sub-partitions) simultaneously : partitioning a blob into subsystems means actually defining concurrent subsystems.

Is Cantor, equipped with its set constructs a convenient framework to describe this kind of formalism? Our problem is to simulate properly the watch described by this diagram: how should the blobs, their nesting and partitioning be represented, and how would the watch simulation of

the various concurrent subsystems go?

Giving life (prototyping) to sets

This is what Cantor is all about. In the above examples, several forms of set organization are used. What Cantor provides are immediate concrete models for all such set organizations. We will here prototype the above examples.

Example 1: a line at a bank or at the post office -revisited

This example, which we call the simpl_server example, will provide an opportunity to introduce the main Cantor collection representations: sets, tuples, maps.

Cantor represents both ordered and unordered collections . In Cantor, an ordered collection is called a tuple, an un-ordered collection is called a set. Curly brackets { } surround unordered collections, and square bracket [] ordered ones.

The abstractions of each problem modeled in Cantor, as in most programming languages, are represented by variables. They receive a value through *assignement* operations:

```
variable := expression;
```

which should not be confused with equality expressions like

```
variable = expression
```

which are either true or false. To observe a variable value, if the value is not hidden, (a hidden variable is a private object 'local' to the computation of a func; this will be discussed later) one simply requests its evaluation:

```
> variable;
```

Here '>' is the Cantor prompt. Type the variable name on the right of the prompt, followed by a semi-colon, and a carriage return. The value is displayed on the following line.

In this example we have at least the following non-hidden variables: theQ and theServers representing respectively the waiting line and the service counters.

A snapshot of a configuration with a waiting line of customers and 5 service counters could be:

```
> theQ;
[];
```

`theQ` represents here an empty waiting line.

```
> theServers;
{["b", ["avail", !6!]], ["e", ["avail", !3!]],
 ["c", ["busy", !5!]], ["d", ["busy", !7!]],
 ["a", ["busy", !8!]]};
```

theServers is a set whose elements are pairs [server_name, [status, customer_id]]. In this model, each individual customer is represented by a unique id number, e.g. !7!. Each service counter is similarly represented by a name and a status indication ('avail' or 'busy'), and the id of the (last) customer being served.

`theQ` is a variable representing the waiting line and `theServers` represents the set of service counters. At this point in the simulation, the line is empty, and two service counters (the ones named 'b' and 'e') are idle.

Another snapshot is:

```
> theQ;
[!26!, !27!];
> theServers;
{["e", ["busy", !19!]], ["b", ["busy", !24!]],
 ["c", ["busy", !13!]], ["d", ["busy", !25!]],
 ["a", ["busy", !21!]]};
```

The variable `theServers` is a set representing the collection of all service counters. It is also an association between a well-defined server name and information items: [server_status, cust_id].

Since this is a well-defined association -there is no server name which is associated with two different collections of information items-, one could consider `theServers` as a function to access the information items:

```
> theServers("a");
["busy", !21!];
```

Set of pairs are known as maps. When a map represents a well-defined association it is called a single-valued map or smap.
Similarly, `theQ` may be used as a function, associating to an integer representing a rank in the ordered collection, the element at that rank:

```
> theQ(1);
!26!;
> theQ(2);
!27!;
> theQ(5);
OM;
```

Note that since there is no one in rank 5 in `theQ`, the value returned is `OM` the undefined. The undefined could be called `om` too.

The development of a software model for this example is rather simple. And we will describe it. The basic data model should be defined and initialized:

```
$ create servers: initialized to 'avail' status,
$ and without customer
theServers   := {['a',['avail']], ['b',['avail']], ['c',['avail']],
['d',['avail']], ['e',['avail']]};
$ create a line, empty
theQ := [];
```

Note that Cantor allows a more elegant way of defining those servers:

```
theServers := {[x,['avail']]: x in 'abcde'};
```

A software model has to consider the possible events in this simple system:
- *join*: a customer joins the line
- *serve*: a service counter starts servicing the customer which is at the head of the line
- *complete service* : a service counter completed its task for a customer: that customer leaves the system, the service counter becomes 'avail', unless it has a customer to *serve*.

For this initial version we omit the possibility of a customer leaving the line before being serviced.

To each of these events is associated a corresponding function. We will explicitly define these functions in our model. But we need quickly to introduce function syntax and semantics.

Functions in Cantor are ordinary objects. They may be used as auxiliary in computations, as in traditional mathematics: e.g. a function to compute the speed of a sweet pea falling from a satellite as a function of the altitude above the sea level, or the installments for a loan at a given interest rate. A function, given a list of arguments returns a value. The list may be empty: i.e.the value returned is independant of any input value. A function may be invoked not only for computing a value, but also for 'doing things', e.g. sorting a collection, deleting a file, copying a file to another file, etc. When the value returned by a function is irrelevant, it is convenient to make this returned value OM, the undefined.

It is possible for the user to define her/his own functions. This is done with the following syntax:

```
func(list-of-parameters);
    local list-of-local-ids;      $ optional
    value list-of-global-ids;     $ optional
    statements;                   $ at least one statement
end
```

(Notice the use of comments: everything on a program line on the right of a $-sign is ignored by the Cantor interpreter.)

Since functions are ordinary objects, functions themselves can be the value of a variable. Thus the most common way of defining a function is as an assignment:

```
f := func(x); return x+1; end;
```

This assignment makes the function which returns its arguments augmented by 1, the value of the variable f. Then the assignment:

```
g := f;
```

makes that function the value of the variable g too. Thus:

```
> f(10);
11;
> g(-45);
44;
> f = g;
true;
```

In our functions we will not use `value` declarations. However we will often use `local` declarations. Variables declared local to a func, are accessible only within the scope of that func, i.e. to the program statements defining the func, including possibly other nested func definitions. However, they are not accessible (i.e. hidden) to other parts of the program, or to the console. For instance, a declaration:

```
local x;
```

within a func, creates a variable `x` to which assignements could be made. However, any variable called `x` in other parts of the program -outside the scope of that func - will not be changed, when the code associated to that func is executed. Parameters are considered local variables.
The first functions we will introduce have no arguments, i.e. they carry out a computation with no explicit argument. However they may work on -and modify- the common (global) data structures of the problem: `theQ` and `theServers`. Notice again the generous use of comments (everything on a program line on the right of a $-sign).

the join function:

```
func(); local cust;
    cust := newat;     $ identify a customer with a new atom
    theQ := theQ with cust; $ add the customer at the tail
end;
```

This function uses a single local variable: `cust`.
An *atom* is a data type, much like an *integer* or a *real number*. Its specificities are:
- each atom is created during a session by the built-in operation *newat*
- all the created atoms (within a given session) are distinct.
- the only operation on atoms are the comparison for equality or inequality of two atoms.

Atoms are like identification numbers. There is no arithmetics on identification numbers. What matters, is to be able to use them for identification and garantee that no two system generated id numbers are identical.

The variable `cust` is assigned as value an *atom*.

The `with` operation is the standard way to add an item to a collection. If the collection is an ordered collection that item is added at the end (as the last element) of the collection. If the collection is a set, and if that item is not already a member of the collection, that item is added to the collection. There is an inverse operation, for unordered collection, the `less` operation.

The variable `theQ` is an ordered list which is augmented with the value of `cust`.

the serve function:

```
func(); local srvr,cust;
    if #theQ = 0 then return end; $ no business: stay idle
$ select a server among the available ones
    srvr := arb({x: x in theServers | x(2)(1) = 'avail'});
    if is_om(srvr) then return end; $ all the servers are busy
$ cust is removed from the begining of theQ
```

```
        take cust fromb theQ;
        $ srvr(1) is the server name
        printf srvr(1)+' serves '+itoa(cust)+'\n';
        $ update the theServer map
        theServers(srvr(1)) := ['busy',cust];
    end;
```

This function uses two private (i.e. declared `local`) variables: `srvr`, which corresponds to a service counter, and `cust` which correponds to a customer, the one at the head of the line.
The # operation is the cardinality operation. Given a collection `coll`, `#coll` is the number of elements in the collection. In our case, the cardinality of `theQ` represents the waiting line length. If the line is empty, its cardinality is 0, and there is no service to perform: the serve function should return without doing anything. This is stated with an if-statement:

```
if #theQ = 0 then return end; $ no business: stay idle
```

We have already indicated that each server is a pair: [server_name, [server_status,cust]]. In an ordered collection s, s(i) represents the i-th element of the collection ($i \geq 1$). Therefore if x represents an element of theServers,

```
x(1) is a string (the server name)
x(2) is the pair [server_status,cust]
```

Since the status may be 'avail' or 'busy', either

```
x(2)(1) = 'avail'
```

or

```
x(2)(1) = 'busy'
```

Finally, x(2)(2) is either the current or the last customer. When the service counter starts working, there is no customer, whence

```
x(2)(2) = om, $ the undefined value
```

Later on, the server x having serviced a customer with id `cust`, x(2)(2) becomes `cust`.
The model of the serve function is: the customer at the head of the waiting line is sent to anyone of the available service counters. Therefore, to actually start a service, one has to compute the set of available service counters:

```
{x: x in theServers | x(2)(1) = 'avail'}
```

select arbitrarily an element, and assign this element to the variable representing the service counter:

```
srvr := arb({x: x in theServers | x(2)(1) = 'avail'});
```

The `arb` functions, a choice function, performs the arbitrary selection. As a result of this assignment, if the collection `{x: x in theServers | x(2)(1) = 'avail'}` is not empty, one of its members is assigned to the variable srvr. However if this collection is empty, `arb` returns OM, the undefined. If at this point srvr has value om, all the service counters are 'busy'. The serve function should return without doing anything.

```
if is_om(srvr) then return end; $ all the servers are busy
```

In

```
take cust fromb theQ;
```

take and **fromb** are Cantor keywords. This operation assigns the element at the 'begining' (the **b** in fromb) of the ordered collection theQ to the variable cust, and then removes that element from the collection, thus shortening it by one element. Similarly one could have

```
take var frome aTuple
take var from aSet
```

frome, acts on the end element of the ordered collection aTuple (assigns it to var and then removes it from aTuple), **from** acts by selecting an arbitrary element in the unordered collection aSet (assigns it to var and then removes it from aSet)
The next statement prints a message to the console:

```
printf srvr(1)+' serves '+itoa(cust)+'\n';
```

The message is obtained by concatenating several strings together by means of the + operation. For instance, itoa() is a built-in function which converts an integer or an atom into a string corresponding to the decimal representation of the number passed as argument (see in the user manual, if you have any difficulty with this, section 3.4 on strings). Note the new-line symbol: '\n', which is the equivalent to the key-board carriage return symbol. The **printf** command allows a programmable output (i.e. display on the console, or output to a file) format. The default **print** command has a default output format. (cf. section 7.4 of the user manual)

The data structure representing the service counter [server_name, [server_status,cust]] has to be updated to reflect the status change (from 'avail' to 'busy') and the association with the customer with id `cust`. This is done by the assignment statement:

```
theServers(srvr(1)) := ['busy',cust];
```

In absence of an explicit `return` from the *serve* function as the last statement, the function returns a value OM (undefined).

the complete service function:

```
func(); local srvr;
    srvr := arb({x: x in theServers | x(2)(1) = 'busy'});
    if is_om(srvr) then return end; $ all servers are idle
    $ srvr(1) is the server name
    printf srvr(1)+' completes service of '+
                itoa(srvr(2)(2))+'\n';
    $ update the theServer map
    theServers(srvr(1))(1) := 'avail';
end;
```

In our discrete event simulation model, a *complete service* event directs the program to stop service at an arbitrary service counter: this applies only to 'busy' servers. A busy server is selected among all the busy servers. If the selected server is OM, the undefined, the set of busy servers is empty. Otherwise, one should print a message and update the server status. This is very much like the *serve* function.

the mapping: event -> service:

We could now introduce the set of functions, associating an event name to its function:

```
event_map := {
    ['join', func(); local cust;
          cust := newat; $ identify a customer with a new atom
          theQ := theQ with cust; $ add the customer at the tail
    end],
    ['serve',func(); local srvr,cust;
          if #theQ = 0 then return end; $ no business: stay idle
          $ select a server among the available ones
          srvr := arb({x: x in theServers | x(2)(1) = 'avail'});
          $ are all the servers busy ?
          if is_om(srvr) then return end;
          $ cust is removed from the begining of theQ
          take cust fromb theQ;
          $ srvr(1) is the server name
          printf srvr(1)+' serves '+itoa(cust)+'\n';
          $ update the theServer map
          theServers(srvr(1)) := ['busy',cust];
    end],
    ['complete_service', func(); local srvr;
          srvr := arb({x: x in theServers | x(2)(1) = 'busy'});
          if is_om(srvr) then return end; $ all servers are idle
          $ srvr(1) is the server name
          printf srvr(1)+' completes service of '+
                                itoa(srvr(2)(2))+'\n';
          $ update the theServer map
          theServers(srvr(1))(1) := 'avail';
    end ]
};
```

The set `event_map` is a collection of pairs [anEventName, aFunc]. It is therefore a map, just like the collection theServers which is a set of pairs [server_name, [server_status,cust]] . The set `event_map` is a well-defined association, since the same anEventName is never associated with more than one function. It is therefore an smap, and `event_map` may be considered a function: e.g. the expression `event_map('serve')` is exactly the serve function. To execute the serve function, one has to invoke that function with the required argument list which, in our case, is the empty list; the invocation of the serve function is therefore `event_map('serve')()`. A map, being a set of pairs relates always two sets:

- the *domain* of the map, i.e. the set of all 1st elements in the pairs
- the *mage* or the *range* of the map , i.e. the set of all 2nd elements in the pairs (both names are used indifferently)

For instance, in the case of `event_map` :

```
> domain(event_map);
{'serve','complete_service','join'};
```

the domain is the set of event names.

the simulation scheme:

A simple simulation consists in generating (pseudo-)randomly events and requesting the program to execute each time the corresponding function. This may be achieved by this simple loop:

```
$ make customer arriving, and being served 'random'
events := domain(event_map);
$ events is {'join','serve','complete_service'};
nSteps := 150;
for i in [1..nSteps] do
     anEvent := arb(events);
     event_map(anEvent)();
end;
```

In the case of `event_map`, domain(`event_map`) is preciseley the set {'join', 'serve', 'complete_service'} of event names. and range(`event_map`) is the set of corresponding functions.

To run this simple simulation, one needs to decide the number of simulation steps, i.e. the number of events to generate. We consider here, a trial with 150 steps. To define a loop we define a collection [1..150], comprising all integers from 1 to 150. In the for-loop statement we name a loop index i which will take in turn all the values in that collection. Each time through the loop, all the statements between the for-loop opening and end statements will be executed. The first of these calls for the arbitrary selection of an event name, this name is then assigned to the variable anEvent:

```
anEvent := arb(events);
```

The second statement determines the function corresponding to that event name: event_map(anEvent), and then invokes that function without argument:

```
event_map(anEvent)();
```

the whole example as a text file:

We could regroup all of these pieces together into a single program text:

```
$ this portrays a queue and a mutliserver
$ configuration

$_____________________ global data
theServers := {};
possible_status := {'avail','busy'};
$ 'avail'(-able) means actually : idle

$ create servers: initialized to 'avail' status,
$ and without customer
theServers  := {['a',['avail']], ['b',['avail']],
    ['c',['avail']], ['d',['avail']], ['e',['avail']]};
$ create a line, empty
theQ := [];


$_____________________ model_simpl
$ model_simpl: a kind of simplfied discrete event simulation:
$       all the possible events have associated
$       processing functions;
$       they are all described in event_map.
$
$       A simulation consists in generating 'randomly' events
$       and processing the generated events as they arrive.
$

$ observe that the system memorizes always
$ he last customer of a given server
event_map := {
    ['join', func(); local cust;
        cust := newat;$ identify a customer with a new atom
        theQ := theQ with cust;   $ add the customer at the tail
    end],
    ['serve',func(); local srvr,cust;
        if #theQ = 0 then return end; $ no business: stay idle
        $ select a server among the available ones
        srvr := arb({x: x in theServers | x(2)(1) = 'avail'});
        $ are all the servers busy ?
        if is_om(srvr) then return end;
        $ cust is removed from the begining of theQ
        take cust fromb theQ;
        $ srvr(1) is the server name
        printf srvr(1)+' serves '+itoa(cust)+'\n';
        $ update the theServer map
        theServers(srvr(1)) := ['busy',cust];
    end],
    ['complete_service', func(); local srvr;
        srvr := arb({x: x in theServers | x(2)(1) = 'busy'});
        if is_om(srvr) then return end; $ all servers are idle
        $ srvr(1) is the server name
        printf srvr(1)+' completes service of '+
                itoa(srvr(2)(2))+'\n';
        $ update the theServer map
        theServers(srvr(1))(1) := 'avail';
    end]
};

$ make customer arriving, and being served 'random'
events := domain(event_map);${'join','serve','complete_service'};
nSteps := 20;
for i in [1..nSteps] do
    anEvent := arb(events);
    event_map(anEvent)();
end;

$ display a snapshot:
```

```
theQ;                                    theServers;
```

Note the last two statements, which call for the evaluation -and display of the values- of the two main collections `theQ` and `theServers`.

treating a text file as an include file = compiling and running a program

To actually run this program you should:
-type in the text into a file, which we could call 'server0'
-launch Cantor
-use the cmd-I command in Cantor to include the file 'server0'.
Including a file is precisely requesting the cantor system to take its input from that file instead of the console (see section 1 of the Cantor user manual).
You will get something like this (the pseudo-random generation of server names used may yield a different ordering for the server selections) :

```
> c serves 1
e serves 2
b serves 3
d serves 4
d completes service of 4
d serves 5
e completes service of 2
b completes service of 3
a serves 6
b serves 7
e serves 8
[];
{["a", ["busy", !6!]], ["b", ["busy", !7!]],
 ["d", ["busy", !5!]], ["e", ["busy", !8!]],
 ["c", ["busy", !1!]]};
!include server0 completed
```

Since *join* events are not reported via a message on the console, the output remains of limited size. To run more steps, say another 50 steps, type in at the prompt in the Cantor console:

```
> nSteps := 50;
```

Then copy from 'server0' the loop statements:

```
for i in [1..nSteps] do
    anEvent := arb(events);
    event_map(anEvent)();
end;
```

and paste them, at the prompt in the Cantor console. Then select the pasted text (the pasted text changes color) and key-in the ENTER-key or the RETURN-key on your keyboard. You direct in this way your Cantor system to execute the selected statements. (*Warning:* when selecting text in the console for execution, beware of avoiding all prompts: the Cantor parser will consider them as part of your command text, and will most certainly generate an error!). Here is what you'll get at the console:

```
> for i in [1..nSteps] do
    anEvent := arb(events);
    event_map(anEvent)();
end;

>>
>>
>> d completes service of 5
e completes service of 8
c completes service of 1
d serves 9
b completes service of 7
b serves 10
c serves 11
c completes service of 11
c serves 12
d completes service of 9
```

```
a completes service of 6
b completes service of 10
a serves 13
c completes service of 12
d serves 14
d completes service of 14
b serves 15
a completes service of 13
e serves 16
b completes service of 15
e completes service of 16
d serves 17
d completes service of 17
d serves 18
d completes service of 18
>
```

This completes the introduction to the fundamental set constructs: set, tuple, map. Other essential features of Cantor have been introduced: variables, atoms, and funcs.

Exercises

- modify the simpl_server example by requesting that whenever a customer joins the waiting line, or a server completes the service of a customer, a *serve* event be generated and immediately executed
- modify the simpl_server example by requesting that, when a server completes its service with agiven customer, it removes any indication of the last customer being served. (N.B. usually, it is a good idea to keep a record of the last thing that was made, and not to erase or dispose of it. Only by keeping this information it is possible to undo a previous action!)
- modify the simpl_server example by allowing a customer to leave the line before being served. This implies that the line is re-organized. Hint-1: Introduce a new event. Hint-2: Use the following technique on tuples for removing the i-th element of a tuple t

```
if is_integer(i) and i >= 1 then
    t := t(..i-1)+t(i+1..); $ remove t's i-th element
end;
```

- Complexify the simpl_server model by introducing a new server status: 'non-operating', and new events for switching on or off a server from a 'non-operating' status to an 'avail' status.

Example 2: function tables - revisited

In all languages which support recursion -which is the case of Cantor- it is possible to write functions corresponding faithfully to recursive definitions:

```
fact := func(n);
    if not is_integer(n) or n<1 then return om; end;
    elseif n = 1 then return 1;
    else
        return n * fact(n-1); $ n! = n * (n-1)!
    end;
end; $ end fact
```

or similarly for the Fibonacci sequence:

```
Fibonacci  := func(n);
    if not is_integer(n) or n<0 then return om; end;
    elseif n = 0 then return 0;$ F0 = 0
    elseif n = 1 then return 1;$ F1 = 1
    else
        return Fibonacci(n-1)+Fibonacci(n-2);$ Fn+2 = Fn+1 + Fn
    end;
end; $ end Fibonacci
```

But observe, that since computing f(n) requires the values of some or all f(j) for j<n, it would be simpler to keep them actually in a table -i.e. in a map- and consult the table -a collection again. As we will see Cantor supports function tables in a transparent and efficient way, derived from maps.

redefining a function at a point

Defining function tables is possible in Cantor, with the 'function redefinition at a point feature' (see section 5.2 of the user manual): given a func f, it is possible to assign a specific value to f for a specific value of its argument. This is similar to a map assignment:

```
f(n0) := q0;
```

assigns the value q0 to f when its argument is n0 : this is recorded in an auxiliary map owned by the func f, called the *override map* or table for f. When a func expression is evaluated, Cantor always attempts to see if the function argument is in the domain of the *override map* for that function: if the argument is found, the func returns the corresponding image value, otherwise the code for f is executed.

tabulating a function

In our case, we could store in the *override map* precisely the correct value: upon a subsequent call, the recursive function, instead of going again through the whole computation will attempt to retrieve the value from the override map, if this value has already been computed, otherwise it will compute the missing elements in the override table. The changes to the previous code is minimal:

```
fact := func(n);
     if not is_integer(n) or n<1 then return om; end;
     elseif n = 1 then return 1;
     else  $ one arrives here only if fact(n) has not been
           $ recorded yet in the override table
           fact(n) := n * fact(n-1); $ update the override table
           $ return the value stored in the override table
           return fact(n);
     end;
end; $ end fact
```

and for Fibonacci:

```
Fibonacci  := func(n);
     if not is_integer(n) or n<0 then return om; end;
     elseif n = 0 then return 0;
     elseif n = 1 then return 1;
else       $ one arrives here only if Fibonacci(n) has not been
           $ recorded yet in the override table
           Fibonacci(n) := Fibonacci(n-1)+Fibonacci(n-2);
           return Fibonacci(n);$ return the value in the override table
end;
end; $ end Fibonacci
```

This technique is described in the Cantor manual, and example runs are given there (see sections 5.2 and 8.2 of the user manual).

function tabulation is not always meaningful

Not all recursive function lend themselves to tabulation. It makes sense only if the functions keep re-using previously computed values again and again. It is easy to exhibit an example where that is not the case:

consider the merge function. It takes as input two ordered collections (say, of numbers). And that function merges them into a single ordered collection:

```
merge := func(t1,t2);
     if #t1 = 0 then return t2;
     elseif #t2 = 0 then return t1;
     elseif t1(1) < t2(1) then                  $ put the smallest
          return [t1(1)] + merge(t1(2..),t2); $ in front
     else                                       $ and merge what's left
          return [t2(1)] + merge(t1,t2(2..)); $ behind
     end;
end; $ end merge
```

It is quite 'legal' to introduce tabulation in this func:

```
merge := func(t1,t2);
     if #t1 = 0 then return t2;
     elseif #t2 = 0 then return t1;
     elseif t1(1) < t2(1) then      $ put the smallest in front
          merge(t1,t2) := [t1(1)] + merge(t1(2..),t2);
          return merge(t1,t2);
     else                           $ and merge what's left
          merge(t1,t2) := [t2(1)] + merge(t1,t2(2..));$ behind
          return merge(t1,t2);
     end;
end; $ end merge
```

but it is unlikely to result in any speed-up, since the pairs (t1,t2) exhibit no a priori pattern which could be re-used, unless this is a specific consequence of a given problem definition.

Exercise

- create a func to compute all the permutations of a given set

Example 3: a data table - revisited-1

We have already seen the visual presentation in a table, one row per item:

...			
$itemName_i$	$price_i$	nbr_sold_i	$nbr\_in\_stock_i$
...			

The generic term for that table could be formally written:
{ ..., [$itemName_i$, [$price_i$, nbr_sold_i, $nbr\_in\_stock_i$]], ... }
where curly brackets { } surround unordered collections, and square bracket [] ordered ones, as in:

```
food_store :=
{    ['lettuce',        [0.75,24,21]],
          ['sour cream',     [0.63,27,23]],
          ['milk',           [0.82,82,46]],
          ['skimmed milk',   [0.64,76,52]],
          ['grapefruit',     [0.25,45,68]]
};
```

the data table as a map

This map representation lends itself to data retrieval forms in the style of function calls:

```
> food_store('milk'); $ milk data
[0.820, 82, 46];
```

food_store is a s-map, i.e. a function associating to each element of its domain (here the keys i.e. the collection of item_names) a well defined value, actually the 3-ple [price, #sold, #in_stock]. The information may be analyzed, as in the following examples:

```
> $ all the food store items have a price less than 1.0 ?
> forall item in domain(food_store) | food_store(item)(1) < 1.0;
true;
> $ there is a food store item whose price is less than 0.5 ?
> $ each food_store(item) is a tuple [price, #sold, #in_stock]
> exists item in domain(food_store) | food_store(item)(1) < 0.5;
true;
```

Note that outside the quantifier expressions `item` has retained its previous value (or lack of, if it was undefined in the first place). This variable is a 'bound' variable in the quantifier expressions: it is known only within the scope of the expression where it is introduced.
Derived computations may be defined. In the following example, we define a collection of expressions using one of the iterative syntactic forms, on the entire data set:

```
{price*nbr_sold: [price,nbr_sold,~] = food_store(item)};
```

consisting of the values of products:

```
{17.010, 11.250, 67.240, 48.640, 18.000};
```

Then we add all of these values. The sum operation, over a collection is denoted *%+aColl*. Similarly a product operation could be defined *%*aColl* . For instance the factorial function n! could be defined %* [1..n]. The desired operations may be described in a single expression :

```
> $ generated cash-flow is:
> %+{price*nbr_sold: [price,nbr_sold,~] = food_store(item)};
162.140;
```

Derived computations may be defined on a subcollection:

```
> $ generated cash-flow for the products selling less than 0.80
> $ and at more than 40 units is:
> %+{price*nbr_sold: [price,nbr_sold,~] = food_store(item) |
>>   price < 0.80 and nbr_sold > 40};
 59.890;
```

The iterator [price,nbr_sold,~] = food_store(item) requests that each food_store item be identified with a triple, and auxiliary variables are assigned the corresponding values: price for the 1st, nbr_sold for the second.Observe the role played by the tilda sign (~) : it substitutes for an irrelevant item. In the above example, we don't need the 3rd item (#in_stock): thus avoiding an assignement to an unused variable, we indicate the presence of an item, and its uselessness by the tilda. Note that outside the scope of the set expression the variables `price, nbr_sold, item` have retained their previous value (or lack of, if they were undefined in the first place). These variables are the 'bound' variables in the set former.

These formers are extensively described in sections 4.4 and 7.6 of the Cantor user manual. When a food store customer buys an extra lettuce, the food_store map should be updated by incrementing by one the nbr_sold field associated to lettuce, and decrementing the corresponding stock. This is illustrated as follows:

```
> $ update following the sale of a lettuce
> food_store('lettuce')(2); $ the current value of nbr_sold
24;
> $ increment nbr_sold by 1
> food_store('lettuce')(2) := food_store('lettuce')(2)+1;
> food_store('lettuce')(2); $ the current value of nbr_sold
25;
> $ decrement the nbr in stock by 1!
> food_store('lettuce')(3) := food_store('lettuce')(3)-1;
```

A new delivery of 45 bottles of milk is registered by:

```
> $ update following the delivery of 45 milk bottles
> food_store('milk')(3); $ the current value of #in_stock
46;
> food_store('milk')(3) := food_store('milk')(3)+45;
> food_store('milk')(3); $ the current value of #in_stock
91;
```

Now the cash value of the stock is evaluated at:

```
> %+{price*stock: [price,~,stock] = food_store(item)};
154.390;
```

There is more than one way of defining this expression. Here is another one:

```
> %+{price*stock: [~,[price,~,stock]] in food_store};
154.390;
```

The iterator in the last expression emphasizes the set representation of food_store: each member in that set should match a pair, the 1st element in the pair is ignored because of the ~, and the second element should match a triple - and we ignore the 2nd element of that triple.

saving the data table on a file and restoring it

This data set may then be saved in a file - let us call it 'food_store_01_nov_90', for later retrieval:

```
> save('food_store','food_store_01_nov_90');
! Compiling on 'food_store_01_nov_90'
OM;
```

In that case a new session, proceeding with the saved values will start by re-installing that data set:

```
> restore('food_store_01_nov_90');
food_store_01_nov_90 loaded!
{["milk", [0.820, 82, 91]], ["lettuce", [0.750, 25, 20]],
 ["skimmed milk", [0.640, 76, 52]],
 ["sour cream", [0.630, 27, 23]],
 ["grapefruit", [0.250, 45, 68]]};
```

This restore() instruction, wipes out any existing *food_store* variable and defines a new one with the given value. Indeed, the save() instruction records not only the value but the variable name (which is the 1st argument in the save function call) too.

the example as an include file

You may reproduce this session by typing in a Cantor console the following instruction script, or by typing in this script into a text file and, using the !echo on switch, include this text file into a Cantor console (e.g. by means of the *cmd-I* menu invocation):

```
food_store :=
{   ['lettuce',   [0.75,24,21]],
        ['sour cream',    [0.63,27,23]],
        ['milk',          [0.82,82,46]],
        ['skimmed milk',  [0.64,76,52]],
        ['grapefruit',    [0.25,45,68]]
};

food_store('milk'); $ milk data

$ all the food store items have a price less than 1.0 ?
forall item in domain(food_store) | food_store(item)(1) < 1.0;

$ there is a food store item whose price is less than 0.5 ?
$ each food_store(item) is a tuple [price, #sold, #in_stock]
exists item in domain(food_store) | food_store(item)(1) < 0.5;

$ generated cash-flow is:
{price*nbr_sold: [price,nbr_sold,~] = food_store(item)};
%+{price*nbr_sold: [price,nbr_sold,~] = food_store(item)};

$ generated cash-flow for the products selling less than 0.80
$ and at more than 40 units is:
%+{price*nbr_sold: [price,nbr_sold,~] = food_store(item)|
    price < 0.80 and nbr_sold > 40};

$ update following the sale of a lettuce
food_store('lettuce')(2); $ the current value of nbr_sold
$ increment nbr_sold by 1
food_store('lettuce')(2) := food_store('lettuce')(2)+1;
food_store('lettuce')(2); $ the current value of nbr_sold
$ decrement the nbr in stock by 1!
food_store('lettuce')(3) := food_store('lettuce')(3)-1;

$ update following the delivery of 45 milk bottles
food_store('milk')(3); $ the current value of #in_stock
food_store('milk')(3) := food_store('milk')(3)+45;
food_store('milk')(3); $ the current value of #in_stock

%+{price*stock: [price,~,stock] = food_store(item)};
%+{price*stock: [~,[price,~,stock]] in food_store};

save('food_store','food_store_01_nov_90'); $ adapt with the proper date
$ start a new session with:
$ restore('food_store_01_nov_90');
```

the relational version of the data table

Getting familiar with data processing in Cantor is one thing. What would be more interesting is to compare this kind of data representation with for example the relational model. In that model, *food_store* is a 4-ary relation, i.e. a relation with 4 columns: item_name, price, nbr_sold, nbr_in_stock.

itemName	price	nbr_sold	nbr_in_stock
'lettuce'	0.75	24	21
'sour cream'	0.63	27	23
'milk'	0.82	82	46
'skimmed milk'	0.64	76	52
'grapefruit'	0.25	45	68

This relation, is a set of quadruple, with generic term:
{...., [itemName,price, nbr_sold,nbr_in_stock], ...}
instead of being a map from item names into triples [price, nbr_sold,nbr_in_stock]. I.e. the relational version of this data table example is

```
{    ['lettuce',       0.75,24,21],
          ['sour cream',    0.63,27,23],
          ['milk',          0.82,82,46],
          ['skimmed milk',  0.64,76,52],
          ['grapefruit',    0.25,45,68]
};
```

By definition, the relational representation of a n-ary relation is by a set of n-uples.

If we assume that each tuple is well identified by its itemName, without ambiguity, then itemName may be considered as the *primary key* of this relation. When a primary key is insufficient, it is possible to introduce a 2nd-ary key, a ternary key, ... to arrive at a single-valued map [primary-key, 2nd-ary key,...] -> tuple in the relation : each tuple should be uniquely associated with its keys.

Exercises

-write a func fo transform `food_store` into its corresponding relational version
-write a func to transform the relational version of the food store data table into a map having as domain the prices
-write the expressions or funcs which in the relational representation would compute the same values or updates as the ones shown in the above `food_store` example
-write a func to transform the Cantor map representation of a data table into the relational representation. Do this for an arbitrary n-ary relation, assuming that each of the columns is a plain data item (a string, a boolean or a number, but not a set, a tuple, a func)
-we indicated that the generated cash-flow was the value of the expression:

```
%+{price*nbr_sold: [price,nbr_sold,~] = food_store(item)}
```

Is this correct? Would'nt

```
%+[price*nbr_sold: [price,nbr_sold,~] = food_store(item)]
```

be better? What is the correct expression for the cash-value of the stock?

Example 3: a data table - revisited-2

We intend to show in this section that the relational data model is strictly a subset of the possible data representations available in Cantor.

We discussed earlier an extended model of the food store example; we have two main collections, actually sets:
1-an item data table whose visual presentation could be:

...				
$itemName_i$	$price_i$	nbr_sold_i	$nbr\_in\_stock_i$	supplier_a supplier_b ....
...				

and whose generic presentation as a map could be:
{ ..., [$itemName_i$, [$price_i$, nbr_sold_i, $nbr\_in\_stock_i$, {supplier_a, supplier_b, ..}]], ...}
Observe that in this representation, the 5th column is not made of basic data items, instead, each is a collection 'supplier_list' represented by a set of supplier names.
2-a supplier table which could have a visual representation as follows:

....	
supplier_id	... $itemName_i$ $supplier\_price_i$ $itemName_{i+1}$ $supplier\_price_{i+1}$ ....
....	

which corresponds, in Cantor, to a map having as generic term:
{ ..., [supplier_id, {...,[$itemName_i$, $price_i$],.....}],.....}
representing a map with a range of maps.

In this extended example the item data table could be:

```
food_store_ext :=
{    ['lettuce',        [0.75,24,21,{'joe','max','mike'}]],
          ['sour cream',     [0.63,27,23,{'liu','mike'}]],
          ['milk',           [0.82,82,46,{'liu','mike'}]],
```

```
            ['skimmed milk',   [0.64,76,52,{'liu','mike'}]],
            ['grapefruit',     [0.25,45,68,{'joe','max','mike'}]]
};
```

Assuming that ordinary names are sufficiently well-defined - and uniquely defined - to qualify as identifications for suppliers, a snapshot of the supplier table could be represented by:

```
suppliers := {
    ['joe', {   ['lettuce', 0.30],
                ['grapefruit', 0.18]}
    ],
    ['max', {   ['lettuce', 0.32],
                ['turnip', 0.11],
                ['apple', 0.05],
                ['grapefruit', 0.19]}
    ],
    ['liu', {   ['milk', 0.30],
                ['skimmed milk', 0.18],
                ['butter 1st qual', 0.40],
                ['salty butter', 0.32],
                ['whipped butter', 0.45],
                ['whipped cream', 0.29],
                ['sour cream', 0.23]}
    ],
    ['mike', {  ['milk', 0.31],
                ['skimmed milk', 0.18],
                ['lettuce', 0.29],
                ['banana', 0.12],
                ['grapefruit', 0.20],
                ['sour cream', 0.24]}
    ]
};
```

The domain of *suppliers* is a set of supplier_names. Its range is a set of s-maps, each associating to an item name its (supplier-) price. To know the price used by 'mike' on the banana item one evaluates the expression `suppliers('mike')('banana')`. Indeed:

```
> suppliers('mike');
{["skimmed milk", 0.180], ["milk", 0.310], ["lettuce", 0.290],
 ["banana", 0.120], ["sour cream", 0.240], ["grapefruit", 0.200]};
> suppliers('mike')('banana');
0.120;
```

To find the best price on milk:

```
> %min {x('milk'): [~,x] in suppliers};
0.300;
```

Note that if no one sells milk, x('milk') is always undefined, and the set {x('milk'): [~,x] in suppliers} is empty, therefore its min is also undefined: the best price for 'milk' is undefined if 'milk' is not available!

This could be turned into a function:

```
best_price := func(item);
            return %min {x(item): [~,x] in suppliers};
end; $ end best_price
```

and tested:

```
> best_price('milk');
0.300;
> best_price('cow');
OM;
```

This func may be improved to compute the suppliers which deliver at the best price:

```
$the suppliers giving the best price on a given item
best_suppliers_price := func(item);
            local best;
            best := %min {x(item): [~,x] in suppliers};
            if is_om(best) then return; end;
            $ return both the best price and the list of suppliers
```

```
            $ selling at that price
            return [best, {supl: [supl,x] in suppliers | [item,best] in x}];
  end; $ end best_suppliers_price
```
and tested:
```
  > best_suppliers_price('milk');
  [0.300, {"liu"}];
  > best_suppliers_price('cow');
  OM;
```
In relational calculus, the suppliers is a 3-ary relation with three columns whose generic term is
{..., [supplier_name,item_name,item_price], ...}
and whose visual representation is

supplier_name	item_name	item_price
'joe'	'lettuce'	0.30
'joe'	'grapefruit'	0.18
'max'	'lettuce'	0.32
'max'	'turnip'	0.11
'max'	'apple'	0.05
'max'	'grapefruit'	0.19
'liu'	'milk'	0.30
'liu'	'skimmed milk'	0.18
'liu'	'butter 1st qual'	0.40
'liu'	'salty butter'	0.32
'liu'	'whipped butter'	0.45
'liu'	'whipped cream'	0.29
'liu'	'sour cream'	0.29
'mike'	'milk'	0.31
'mike'	'skimmed milk'	0.18
'mike'	'lettuce'	0.29
'mike'	'banana'	0.12
'mike'	'grapefruit'	0.20
'mike'	'sour cream'	0.24

in contrast with the nested map representation we have used in our examples.

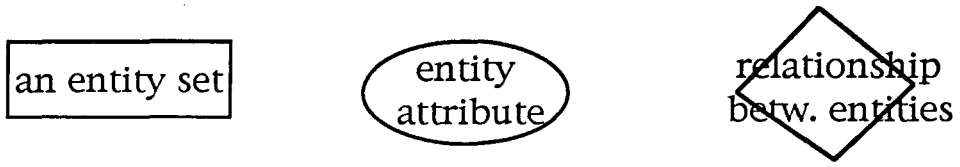
Exercises

- what is the arity of the relation associated to the item data table in the extended food store example and represented by food_store_ext ? what is its relational representation?
- are relations s-maps? m-maps?
- what is the generic term representation for an integrated food store model with visual representation:

...				
$itemName_i$	$price_i$	nbr_sold_i	$nbr\_in\_stock_i$	supplier_a price_a supplier_b price_b ....
...				

provide its explicit map representation, corresponding to the data in `food_store_ext` and `suppliers` in the above examples; what is the arity of this relation? provide its relational representation.
- write the expressions or funcs which in the relational representation would compute the same values as the ones shown in the above `suppliers` example
-how could `best_suppliers_price` be modified to avoid re-traversing the entire data set each time it is invoked? (Hint: use function tabulation)
- write a func for updating suppliers, when a supplier changes its price. How should this be

reflected in `best_suppliers_price` ?
- the above example represents a food store a the current supplier prices. Since suppliers item prices may vary from supplier to supplier and from day to day, how should one represent the supplier price history? the stock and the stock value?
- write a func which will transform a homogeneous map into a relation. Beware that a map could have nested sets and tuples in its range or domain. Apply this to transform the maps food_store, food_store_ext and suppliers in the above examples into relational representations
- write a func which transforms a relational representation of a data table into a map-oriented representation, where the domain is the primary key.
- write a func which transforms a relational representation of a data table into a map-oriented representation, taking into account the primary, the secondary, ... the n-ary keys.
- the Entity Relationship Diagrams (ERD) are used to describe conceptual data models. An ERD consists of the following objects:



Unlabelled edges link the entity attributes to their entity, and a relationship to its entities. An entity set is in the relational data model an n-ary relation, and its entity attributes are the column names. A relationship represents a relation between two or more relations . Draw the Entity Relationship Diagrams for the food store and its suppliers, corresponding to the two examples of 'extended food store'.

Example 3: data tables - revisited-3

We have seen that Cantor allows data representations which are more expressive than those allowed by the relational model. However, what is gained in expressiveness, is lost in uniformity. This point will be made clear in the systematic review of the operations in the relational algebra presented in this section.

In the previous discussion of the data tables we never investigated the questions relating to the interaction between data tables. A good introduction to this is to expose, and illustrate, the relational operations.

Therefore in what follows, in this section, we assume that n-ary relations are sets of n-tuples (tuples of cardinality n). Then the arity of a relation R is #arb(R). Indeed, all the tuples in R are assumed to have the same cardinality.

The main relational operations[3] on relations are: union, set difference, cartesian product, projection, selection, intersection, quotient, join, natural join.

formal Cantor definion of the relational operations

Each of these has a relatively simple definition.
1- union : the union of relations R and S is R+S, the standard set union. Note that the union makes sense only if both relations have the same arity
2- set difference : again, if R and S have the same arity, this is R-S, the standard set difference
3- cartesian product : the cartesian product of R and S is

```
{r+s: r in R, s in S};
```

thus if R has arity k1 and S has arity k2, the cartesian product is made of (k1+k2)-tuples, the 1st k1 components of which form a tuple in R and the last k2 components, a tuple in S.
4- projection : a projection is defined by a pair (R, sub_col) where R is a relation and sub_col is a tuple of distinct integers in the range [1..arity(R)]. The projection defined by (R,sub_col) is

[3]See for instance Principles of Database Systems, J.D. Ullman, Computer Science Press

{ %+[[x(i)]: i in sub_col]: x in R}

The compound concatenation operation %+[[x(i)]: i in sub_col] concatenates together the components x(i) for all the i's in sub_col, in the specified order. Sometimes, instead of a list of component number, a relation is specified by its column names or 'attributes'. In that case there is a map col_attr -> col_rank Lets us call such a map, col_map. If sub_col is specified by column names, the projection is defined, by recomputing the column rank from its name :

{ %+[[x(col_map(name))]: name in sub_col]: x in R}

5- selection : let R be a relation and let F be a formula involving

i) operands that are string constants constants or component numbers

ii) the arithmetic comparison operators >,<,=, >=, <=, /=

iii) the logical operators and, or, not

We assume that F is a string and has a valid syntax.

A selection defined by the pair (R,F) is the set of all tuples in R satisfying F_R, where F_R is obtained from F by replacing i by t(i): for instance if F is

```
'1>3 or "joe" = 2'
```

then F_R is

```
't(1)>t(3) or "joe" = t(2)'
```

and the selection is

{t: t in R | F_R}

in our example this would be:

{t: t in R | t(1)>t(3) or "joe" = t(2)}

Let us examine step by step how this may be obtained in Cantor.

Let F be as follows:

```
> F := '1>3 or "joe" = 2';
```

i) scan a decompose F into its token stream:

```
>tf := scan(F+"",1,1);
>tf;
[1, ">", 3, "or", "\q", "joe", "\q", "=", 2];
```

Note that adding an empty string to F does not change the value of the 1st argument to the scan function. However, since scan 'destroys' its arguments. It is the copy F+"" of F which will be destroyed, but not the original ...

ii) replace each integer i, which represents a column number by x(i) as a string

```
> tf :=
  [if is_integer(u) then 'x('+itoa(u)+')' else u end: u in tf];
```

Since itoa(u) convert an integer into its decimal string representation 'x('+itoa(u)+')' is just the concatenation of three strings

```
> tf;
["x(1)", ">", "x(3)", "or", "\q", "joe", "\q", "=","x(2)"];
```

iii) reconstitute a transformed formula by just concatenating all the pieces, (add a space in between each part):

```
> zf := %+ [u+' ': u in tf];
> zf;
"x(1) > x(3) or \q joe \q = x(2) ";
```

iv) this is a parsable expression, let us parse it

```
> Fr := analyze(zf);
> Fr;
x(1) > x(3) or  joe  = x(2);;
```

Here Fr is not a string, it is an abstract syntax tree. Its structure may be revealed by an ugly-print (default output is by pretty-print) (see section 6 of the user manual) :

```
> ugly(Fr);
( CALL :
  ( or :
     ( > :
          ( SELECTOR :
               ( T_Id :    x )
               ( ( :
                    ( :
                         ( T_Integer :     1 )
          ( SELECTOR :
               ( T_Id :    x )
```

```
                ( ( :
                      (  :
                            ( T_Integer :      3 )
      ( = :
            ( T_String :        joe   )
            ( SELECTOR :
                  ( T_Id :     x )
                  ( ( :
                        (  :
                              ( T_Integer :      2 )OM;
  >
```

v) in fact we are not concerned with the top node in this parse tree, which CALL's for evaluation. This is a technicality!

```
  > Fr := Fr(1); $ let af be its own 1st (i.e. left) subtree
  > Fr;
  x(1) > x(3) or  joe  = x(2);
```

vi) the selection is just

{x: x in R | eval(Fr) }

We use here the built-in *eval* function which evaluates (i.e. computes the set value, or the integer value, etc. of-) the abstract syntax tree form of an expression.

```
  > R;
  {[54, 64, 65, 16], [90, 3, 58, 1], [41, 90, 23, 25],
   [5, 16, 86, 0], [42, 24, 42, 39], [36, 54, 49, 43],
   [90, 2, 50, 49], [63, 88, 25, 63], [78, 86, 28, 98],
   [95, 42, 53, 58]};
  > {x: x in R | eval(Fr)};
  {[95, 42, 53, 58], [78, 86, 28, 98], [41, 90, 23, 25],
   [90, 3, 58, 1], [90, 2, 50, 49], [63, 88, 25, 63]};
```

The selection operation involves the formula transformation operations i) - v) captured by the following func:

```
  fla_transf := func(F);
      local tf,Fr;
      tf := scan(F+"",1,1);
      tf := [if is_integer(u) then 'x('+itoa(u)+')' else u end:
                  u in tf];
      F := %+ [u+' ': u in tf];
      Fr := analyze(F)(1);
      return Fr;
  end; $ end fla_transf
```

Under those conditions, selection (R,F) is defined by

{x: x in R | eval(fla_transf (F)) }

6- intersection : (if R and S have the same arity) this is just

R*S

7- quotient : Let R and S be relations of arity r and s, respectively, with r > s. The quotient R:S is

{t(..r-s): t in R | forall u in S | t(..r-s)+u in R}

Given an arbitrary tuple t, the expression t(i..j) represents the tuple of length j-i consisting of all the elements of t between ranks i and j (included). If the lower bound -here i - is omitted, it is assumed to be 1. If the upper bound -here j- is omitted, it is assumed to be #t. Therefore the quotient expression is equivalent to {t(1..r-s): t in R | forall u in S | t(1..r-s)+u in R}.

It may be more efficient to define the quotient in several steps:

```
  R_quot := {t(..r-s): t in R}; $ this is just a projection
  quotient := {u: u in R_quot | forall v in S | u+v in R};
```

Indeed the forall is evaluated once for each element in R_quot, which may have a cardinality much smaller than that of R.

8- join : (often called the θ-join) Let θ be an arithmetic comparison operator (<,>, <=,>=, =, /=) . Let R, S be relations of arity r and s respectively. Let i,j be positive integers i ≤ r and j ≤ s Then the join is defined by (R,S, i θ j) to be the selection on the cartesian product R x S defined by the formula i θ r+j :

{x+y: x in R, y in S | eval(fla_transf ('i θ r+j')) }

When q is '=' then this called an equijoin. Column names are sometimes used instead of ranks, requiring the interposition of the col_maps for both R and S to transform the names into column numbers. For instance the join (R,S, A θ B) is defined as (R,S, col_map_R (A) θ col_map_S (B)).

9- natural join : Let R, S be relations with named columns. Let col_map_R, col_map_S be their respective col_maps. Consider all the columns with the same names in R and S, i.e.

```
common_attr := domain(col_map_R ) ∩ domain(col_map_S );
```

The natural join is best defined in two steps:

i) compute the intersection of all the equijoins (R,S, A = A) over all the attributes A in common_attr :

```
equi_joins := %* {{x+y: x in R, y in S |
     eval(fla_transf ('col_map_R (A) = r+col_map_S (A)')) } :
          A in common_attr };
```

ii) project out all the columns of S correponding to the common attributes

```
S_common_rnks := {col_map_S (attr) : attr in common_attr };
```

is the set of column numbers in S for the attributes common to R and S, and

```
S_cols := [j: j in [1..s] | j notin S_common_rnks ];
```

is the ordered set of all other column numbers in S, correponding to the attributes of S not in R. Therefore the projection list, is the concatenation of all r columns of R, and of the columns in S_cols, shifted by r in the cartesian product, and therefore in equijoins as well:

```
sub_col := [1..r]+[j+r: j in [1..s] | j notin S_common_rnks ];
```

The natural join is then

{ %+[[x(i)]: i in sub_col]: x in equijoins }

We could put this together in a simple func:

```
natural_join := func(R, col_mapR, S, col_map_S);
     local common_attr, equi_joins, S_common_rnks, sub_col;
     common_attr := domain(col_map_R ) ∩ domain(col_map_S );
     equi_joins := %* {{x+y: x in R, y in S |
          eval(fla_transf ('col_map_R (A) = r+col_map_S (A)'))}:
               A in common_attr };
     S_common_rnks := {col_map_S (attr) : attr in common_attr };
     sub_col := [1..r]+[j+r: j in [1..s] |
               j notin S_common_rnks ];
     return { %+[ [x(i)]: i in sub_col]: x in equijoins };
end; $ end natural_join
```

Exercises

- let R be {[1,2,3], [4,1,6], [3,2,4]}, and S be {[2,7,1], [4,1,6]}. Compute R x S, the projection (R, [1,3]), the selection (R, 2 = 2).
- let R be {[1,2,3,4], [1,2,5,6], [2,3,5,6], [5,4,3,4], [5,4,5,6], [1,2,4,5]}, and let S be {[3,4], [5,6]}. Compute the quotient R:S.
- let R be {[1,2,3], [4,5,6], [7,8,9]}, let S be {[3,1],[6,2]}. Compute the join (R,S, 2 < 1).
- let R be {[1,2,3], [4,2,3], [2,2,6], [3,1,4]}, and S be {[2,3,4], [2,3,5], [1,4,2]}. Let the attribute map for R and S be defined by:

col_map_R := {['A',1], ['B',2], ['C',3]};
S by col_map_S := {['B',1],['C',2]};
Compute the natural join of R and S.

- express by means of the relational operations on the `food_store`, and the `suppliers` (assumed to be in relational presentation, instead of map-oriented presentation) the following:
  i- the two suppliers who could supply most if not all the products sold by the food store
  ii- the margin computed from the difference between the food store sales price and the worse

supplier price
iii- the stock value computed from the suppliers best price for each item

Example 4: pre-requisites - revisited-1

To address the pre-requisites problem a convenient representation of the dependency structure represented in fig. 2 is needed. A common way is to represent this graph as the collection of its edges, each edge being the representation of an arrow. An arrow is an ordered collection, e.g., 1->2 may be represented by the pair [1,2], and the whole dependency graph by the set (unordered collection) of all these arrows:

```
dependence := {  [1,2], [1,4], [1,6],
                 [2,3], [2,5], [2,7],
                 [4,5], [4,7],
                 [6,7], [6,8], [6,9]
             };
```

`dependence` is a set of pairs, therefore it is a map. However, this is clearly not a s-map: for instance, three possible values are associated to (chapter) 1. This map is a multi-valued map or m-map having as domain (set of 1st elements in the pairs) {1,2,4,6} and as range (set of second elements in the pairs) {2,3,4,5,6,7,8,9}

```
> domain(dependence);
{4, 6, 1, 2};
> range(dependence);
{7, 9, 8, 5, 6, 2, 3, 4};
```

The set of all the dependents or successors of a given chapter is

```
dependence{aChapter};
```

a naive problem definition

To establish the set of prerequisites, one uses the following idea: a chapter ch_1 is a pre-requisite to the chapter ch_0 if

$ch_0 \in$ `dependence{`ch_1`}` or `dependence{`ch_1`}` $\cap$ `prerequisites` $\neq \emptyset$

where `prerequisites` is the set of already computed prerequisites. This computation stops when no new chapter is added to `prerequisites` .I.e., `prerequisites` satisfies the equation:

`prerequisites` = {ch : ch in domain(dependence) |
$ch_0 \in$ `dependence{ch}` or `dependence{ch}` $\cap$ `prerequisites` $\neq \emptyset$ }

where ch_0 is in our case 7.

a naive fixed-point solution

There are many ways to formalize the search for a solution to this equation. The above equation is a fixed-point equation of type:

$X = \{u: u \in S \mid K(u,X)\}$

A standard solution consists in computing a sequence

$X_0 = \{\}$

....

$X_{n+1} = \{u: u \text{ in } S \mid K(u,X_n)\} \cup X_n$

This sequence forms a increasing chain of subsets of S:

$X_n \subseteq X_{n+1}$

This chain has an upper bound: S. Therefore, eventually this sequence will stop growing: either X_n becomes the whole set S, or $X_{n+1} = X_n$.

This could be programmed in Cantor as follows:

```
X := {};
T := {u: u in S | K(u,X)};
while T /= X do
          X := X + T;
          T := {u: u in S | K(u,X)};
end;
```

In our example S is domain(`dependence`) and K(u,X) is:

```
7 in dependence{u} or dependence{u} * X /= {}
```

Actually running this program yields:

```
> K := func(u,X);
```

```
>>   return 7 in dependence{u} or
>>   dependence{u} * X  /= {};
>> end;
>
> X := {};
>
> S := domain(dependence);
> T := {u: u in S | K(u,X)};
> while T /= X do
>>         X := X + T;
>>         T := {u: u in S | K(u,X)};
>> end;
> X;        $ the fixed point
{6, 4, 1, 2};
```

This solution is an example of least fixed point solution. Actually, all the iterative problems in computer science may be stated as fixed point problems.

Here we did not really touched the specifics of the problem: we translated a naive problem definition into a naive fixed point solution.

an ideal complexity model for a set machine

We are using here the usual notation for complexity -in its simple version. We are concerned with size complexity: e.g. the number of elements in a collection, and with time complexity, i.e. the number of time units necessary to perform a given computation. When we say:
the (asymptotic) size complexity of a problem is O(f(n))
we mean for a specific kind of size , e.g. memory or disk space,
there is a constant c > 0 such that, for all sufficiently large n,
the problem size < c*f(n)
When we say:
the (asymptotic) time complexity of an algorithm or function is O(f(n))
we mean for any specific specific measure of time , e.g. seconds, micro-seconds, years, used to evaluate the duration of the computation of that algorithm, operation or function evaluation
there is a constant c > 0 such that, for all sufficiently large n,
the (asymptotic) time < c*f(n)
For instance a time complexity of O(1) characterizes a process which has a constant time upperbound. When a time complexity estimate O(f(n)) is used, generally, n characterizes the problem size.

In the table below we present the definition of the asymptotic time cost for executing some elementary operations on an ideal set machine. It is understood that most set machine operations could be coded cleverly as collections of these elementary operations.

operation	**description**	**complexity**
ɔ s	arbitrary choice	O(1)
∃ x ∈ s | K(x)	existential quantifier	#s * cost(K(x))
∀ x ∈ s | K(x)	universal quantifier	#s * cost(K(x))
for x ∈ s do .. end	for-loop supervision	O(#s)
m := ...	map or set assignment	O(1)
t := ...	tuple assignment	O(1)
s with x	collection addition	O(1)
s less x	set element deletion	O(1)
x ∈ s	membership test	O(1)
f(x)	value of f (only if f is a smap)	O(1)
f(x) := ...	index assignment to a function or a map of a pre-computed term	O(1)

The following simple example will illustrate these complexity notions.

Given the above complexity table, one could infer that the complexity for computing a slice {1..n} or [1..n] is O(n) . Indeed, this is the complexity of the following program, which could be considered as the micro-code for slices:

```
aSlice := {}; $ resp ..:= [];
for x in [1..n] do
    aSlice := aSlice with x;
end;
```

The initialization cost for aSlice is O(1), i.e. c0. The loop supervision cost is proportional to the number of iterations in the loop: it is O(n) c1*n. The loop is repeated n times. Each time the executed code requires O(1) i.e. c2 for an element addition to the collection aSlice, and time O(1) i.e. c3 for assignment of this new value to aSlice. Summarizing the execution time for this micro-code is bounded, for sufficiently large n by
c0 + c1*n + (c2+c3)*n
Taking C = c0+c1+c2+c3 it is easy to verify that for sufficiently large n the execution time is bounded by C*n, i.e. is O(n).

We said, that these are complexities of an ideal machine. Indeed, most implementations do not meet these requirements. For instance, if a set has N elements, in practice, the complexity of a membership test is O(log N). Very often actual complexities are even worse: if a data structure can accomodate upto N elements, then each access has complexity O(log N). But if N is much larger than all other size parameters in the problem, N is actually a problem constant: in that case, O(log N) is a uniform bound for each individual access and this may be considered as a constant overhead, i.e. a O(1) complexity!!!

Conversely, it is possible to prove that every set algorithm of a large class may be implemented with suitable data structures which will meet the requirements indicated in the above table[5]. But this implementation is not a Cantor implementation : it is an implementation in an actual machine language, e.g. C, Pascal, an assembler language.

Despite the fictitious character of 'the ideal set machine' it is a very useful model for elaborating algorithms and seeking optimisations.

designing a more efficient algorithm with finite differencing

To achieve a more efficient solution, it seems reasonable to replace the costly expression evaluations by simpler or 'more economical' ones, according to the just exposed complexity model. The following observations are major guiding principles:

- just like tabulation could be a remarkable speed-up in recursive function evaluation, one should separate in a fixed-point loop all the data which need no longer be re-examined, from the 'new' data to process, to avoid reprocessing always the same data. Since we are constructing a chain of sets $X_0 \subseteq ... \subseteq X_n \subseteq X_{n+1} \subseteq ...$, at step n, the new elements are those in $\partial X_n = X_{n+1} - X_n$. Since $X_{n+1} = X_n + \partial X_n$, we are looking for a method to involve only these new elements in the next computation step.
- while computing the re-assignment $X_{n+2} := f(X_{n+1})$, we want to exhibit an incremental computation for $f(X_n+\partial X_n)$, e.g. of incremental cost $O(\#\partial Xn)$.
- complex operations are those involving the processing of at least one entire collection each time: these operations cannot be carried out at a fixed cost! For instance $A \cap B$ has a cost of O(min(#A,#B)) , $A \cup B$ has a cost of O(#A+#B) and testing A /= B (or A = B) has a cost of O(min(#A,#B)) . As abstract operations they play an essential role in defining a solution; however, they should be eliminated, if possible from optimized versions, and replaced by loops of incremental operations each with a fixed cost

To apply these principles one operate on the symbolic definition of the (set and loop) expressions, rewriting them and substituting in provably equivalent ones. The search for incremental operations is in essence inspired by the XVII century technique of formal polynomial differentiation, used in the making of polynomial tables, and called finite differencing.

Rather than providing here a systematic presentation of this technique, we will just introduce the

scheme, and show its effectiveness. This technique has been established and developped by R. Paige[4].

step-wise refinement guided by finite differencing

We could consider the inverse graph of dependence:
dependence_inv := {[x,y]: [y,x] in dependence};
Then the problem is to compute the reacheability set of node 7, in this (inverse) graph, i.e. all the nodes in the graph which could be reached, following dependence_inv's edges starting from the source node 7. This set could be defined as a fixed-point :

X = dependence_inv[X]

where the notation g[A] means, as usual, the union of all the image sets g{a}: a in A, that is in Cantor notation:

%+{g{a}: a in A} = (by definition) g[A]

A fixed-point sequence converging to the set X satisfying X = g[X] is

X0 := {s}; $ s is the source, here it is 7
Xi+1 := Xi + g[Xi]; $ where g[Xi] = %+{g{x}: x in Xi}

This may be implemented in the following loop:

```
X := {s};
f := func(d); return d%+{g{x}: x in d}; end; $ f(D) is D+g[D]
T := f(X);
while X /= T do
     X := T;
     T := f(X);
end;
$ at this point X is the fixed-point
```

where s is 7 and g is dependence_inv . Actually running this program yields:

```
> $ the inverse relation
> dependence_inv := {[x,y]: [y,x] in dependence};
> dependence_inv;
{[8, 6], [9, 6], [7, 2], [7, 6], [7, 4], [6, 1], [5, 2],
[5, 4], [4, 1], [3, 2], [2, 1]};
> g := dependence_inv;
> X := {7};
> f := func(d); return d%+{g{x}: x in d}; end; $ f(D) is D+g[D]
> T := f(X);
> while X /= T do
>>   X := T;
>>   T := f(X);
>> end;
> $ at this point X is the fixed-point
> X;
{6, 7, 2, 1, 4};
```

This loop may be optimized. Note that within this loop the expressions T and f(X) are kept equal, this is why one says 'the invariant T= f(X) is maintained throughout that loop'. In fact to maintain that invariant, it is not necessary to fully recompute f(X) each time.
We introduce intermediate expressions to clarify the steps in the computation, and possibly uncover incremental steps:

```
X := {s};
T := g{s} with s;       $ T = X%+{g{x}: x in X} i.e. T = X + g[X]
dN := T-X;              $ introducing the invariant dN = T-X
while dN /= {} do       $ this is equiv to X /= T
     N0 := {y: u in T, y in g{u} | y notin T}; $ this is g[T] - T
     X := T;      $ X changes to X + dN
     T := T+ N0; $ maintain T = X + g[X], after X's change
     dN := N0;          $ maintain the invariant dN = T-X
end;
```

[4]See for instance, Paige R., Kœnig S. : Finite Differencing of Computable Expressions, ACM Trans. Prog. Lang. Syst. 4,3,1982 pp 402-454. Or the chapter on Program Transformation, strongly influenced by that paper, pp 130-185 in the remarkable book: Software Prototyping mit SETL, by E.E. Doberkat and D. Fox, Teubner pub. Stuttgart, 1989.

```
$ at this point X is the fixed point
```

Within the expression N0, T is identical with X+dN, i.e. N0 is just

```
{y: u in X+dN, y in g{u} | y notin T}
```

Whenever u in X, then, by construction g{u} is a subset of T = X+dN. Therefore:

```
{y: u in X, y in g{u} | y notin T} = {}
```

Therefore N0 may be simplified to

```
{y: u in dN, y in g{u} | y notin T}
```

The fixed-point loop then becomes:

```
X := {s};
T := g{s} with s;       $ T = X%+{g{x}: x in X} i.e. T = X + g[X]
dN := T-X;                    $ introducing the invariant dN = T-X
while #dN /= 0 do $ X /= T iff T-X /= {} iff g[X] - X /= {}
     $      achieve dN = T-X
     $      achieve T = X + g[X], upon X := X+dN
     N0 := {y: u in dN,y in g{u} | y notin T}; $ g[T] - T
     X := T; $ X+dN;
     T := T + N0;
     $      assert T = X + g[X]
     dN := N0;
     $      assert dN = g[X] - X = T-X
end;
$ at this point X = X + g[X], T = X
```

mechanical refinement by finite differencing

We have been 'lucky' in identifying a computational increment dN. In general, one proceeds as follows, for a fixed point computation:

```
X := {s};
while  f(X)-X /= {} do
     z := arb(f(X)-X);
     X := X with z;
end;
$ at this point X = X + g[X]
```

We introduce an invariant for each subexpression which occur within the loop and would need to be recomputed as a consequence of the change of one of its terms. We thus have an invariant for the expression f(X)-X, to avoid recomputing it each time through the loop

E := f(X)-X;

We will store this value upon entry to the loop, and we will update E, within the while loop just before X is modified, so that the invariant E = f(X)-X is maintained, at the point where the expression f(X)-X is needed. This update code is called difference code for E with respect to the modification X := X with z;

```
X := {s};
E := %+{g{u}: u in X};        $ introduce E = g[X]
while E-X /= {} do      $ while g[X] - X /= {} iff #(E-Z) /= 0
     z := arb(E-X);
     X := X with z;
     E := E + g{z};      $ maintain E = g[X]
end;
$ at this point X = X + g[X]
```

Mechanical maintenance of an invariant of kind U = V+S, where S is subject to changes, involves difference code of the following kind, for each increment dS

```
U := U + {x: x in dS | x notin U}
```

Therefore the difference code for E should actually be:

```
E := E + {x: x in g{z} | x notin E};
```

or the strictly equivalent loop

```
for x in g{z} | x notin E do
     E := E with x;
end;
```

Clearly another invariant becomes necessary: N = E-X. The result is now

```
X := {s};
E := %+{g{u}: u in X};        $ introduce E = g[X]
N := E-X;          $ introduce N = E-X
while  #N /= 0  do      $ while g[X] - X /= {}
     z := arb(N);
     X := X with z;
     E := E + {u: u in g{z} | u notin E}; $ maintain E = g[X]
          $ maintain N = E-X :
     N := (N+ {u: u in g{z} | u notin X}) less z;
end;
$ at this point X = X + g[X]
```

In the above code it is easy to see that the invariant E is never used after N's initialization, and should be removed:

```
X := {s};
N := g{s} less s; $ %+{g{u}: u in X}-X == g[X] - X
while  #N /= 0  do      $ while g[X] - X /= {}
     z := arb(N);
     X := X with z;
          $ maintain N = g[X]-X :
     N := (N + {u: u in g{z} | u notin X}) less z;
end;
$ at this point X = X + g[X]
```

This 'mechanical' version of the reachability algorithm is slightly less performant -in Cantor- than the previous 'hand coded' one, also inspired by the finite differencing method.
The finite differencing technique, consisting of replacing costly repeated computations by simpler ones, may have an initial overhead which masks its real effect on small data sets. This technique has been extensively studied by R. Paige and is now automated in his APTS system. The tests sets used at the end of this section show the remarkable improvement derived from this technique.

the include file for the prerequisite problem

To compare all of these versions, the following include file has been created

```
$*******************************
!memory 2000000
!echo off
!recordOutput fd_reach.tr

[t1,t2,t3,t4,t5,t6] := [[],[],[],[],[],[]] ;
x_size := [];
n_size := [];
e_size := [];
m_size := [];

$----demo_fd, prepare_data
$   given a graph gr, an experiment index i
$   and optional sources, and maximum arity
$   compute, under various refinements
$   the same set: the set of all nodes
$   reachable from source s in graph gr
demo_fd := func(gr,i opt s,m);
local X,T,E,z;
local N,dN,N0;
local in_time,out_time;
local g,h,prepare_data;

$   prepare_data : compute gr's inverse graph
$   and the maximum arity m. If source s
$   is not defined, let it be any node having
$   maximum arity
prepare_data := func(gr,i opt s,m);
    g := {[x,y]: [y,x] in gr};
    if is_om(m) then
        h := {[x,#g{x}]: x in domain(g)};
        m := %max range(h);
    end;
    if is_om(s) then
        s := arb({x: x in domain(g) | h(x) = m});
    end;
    n_size(i) := #(domain(g)+domain(dependence));
    e_size(i) := #g;
    m_size(i) := m;
    printf '\nn_size(i) : '; print n_size(i);
    printf 's : '; print s;
    printf 'm : '; print m;
    printf 'g{s} : '; print g{s};
    return [s,m];
end; $ end prepare_data

[s,m] := prepare_data(gr,i,s,m);

$   -1   the naive fixed point definition
in_time := clock();

X := {s};
T := X%+{g{x}: x in X};        $ T = X + g[X]
while X /= T do $ X /= T iff T-X /= {} iff g[X] - X /= {}
    X := T;
    T := X%+{g{x}: x in X};   $ maintain T = X + g[X]
end;
$ at this point X = X + g[X], T = X
out_time := clock();
X1 := X;
printf 'duration: ', (out_time-in_time)/60,'\n';
printf 'size: ', #X,'\n';      $ the fixed point size
print X;
t1(i) := (out_time-in_time)/60;
x_size(i) := #X;

$   -2   let us exhibit the main iteration condition in detail
in_time := clock();

X := {s};
T := g{s} with s;      $ T = X%+{g{x}: x in X} i.e. T = X + g[X]
dN := T-X;      $ introducing the invariant dN = T-X
while dN /= {} do      $ this is equiv to X /= T
    N0 := {y: u in T, y in g{u} | y notin T};      $ this is g[T] - T
        $ observe that T is X+N
        $ by construction u in X ==> g{u} subset X+N
        $ whence {y: u in X,y in g{u} | y notin T} = {}
        $ therefore: {y: u in T,y in g{u} | y notin T} is
        $       {y: u in N , y in g{u} | y notin T}
    X := T;  $ X changes to X + dN
```

```
    T := T+ N0;       $ maintain T = X + g[X], after X's change
    dN := N0;         $ maintain the invariant dN = T-X
end;
$ at this point X is the fixed point

out_time := clock();
X2 := X;
printf 'duration: ', (out_time-in_time)/60,'\n';
printf 'size: ', #X,'\n';       $ the fixed point size
if X2 /= X1 then
    printf '\ndiscrepancy between 2 and 1\n';
    print X;
end;
t2(i) := (out_time-in_time)/60;

$   -3   the naive fixed point re-definition
$        includes maintenance of N, X, T
in_time := clock();

X := {s};
T := g{s} with s;       $     T = X%+{g{x}: x in X} i.e.  T = X + g[X]
N := T-X;
while #N /= 0 do $ X /= T iff T-X /= {} iff g[X] - X /= {}
    N0 := {y: u in N,y in g{u} | y notin T}; $ g[T] - T
    $    achieve T = X + g[X]
    X := T; $ X+N;
    T := T + N0;
    $    assert T = X + g[X]
    N := N0;
    $    assert N = g[X] - X = T-X
end;
$ at this point X = X + g[X], T = X
out_time := clock();
X3 := X;
printf 'duration: ', (out_time-in_time)/60,'\n';
printf 'size: ', #X,'\n';       $ the fixed point size
if X2 /= X3 then
    printf '\ndiscrepancy between 2 and 3\n';
    print X;
end;
t3(i) := (out_time-in_time)/60;


$   -4   maintain as invariant %+{g{u): u in X}
$   note that E +g{z} is E + {u: u in g{z} | u notin E}
in_time := clock();

X := {s};
E := %+{g{u}: u in X};
while E-X /= {} do    $ while g[X] - X /= {} iff #(E-Z) /= 0
    z := arb(E-X);
    X := X with z;
    E := E + {u: u in g{z} | u notin E}; $ maintain E = g[X]
end;
$ at this point X = X + g[X]

out_time := clock();
X4 := X;
printf 'duration: ', (out_time-in_time)/60,'\n';
printf 'size: ', #X,'\n';       $ the fixed point size
if X4 /= X3 then
    printf '\ndiscrepancy between 4 and 3\n';
    print X;
end;
t4(i) := (out_time-in_time)/60;

$   note that E +g{z} is E + {u: u in g{z} | u notin E}

$   -5   maintain as invariant %+{g{u}: u in X} - X
in_time := clock();

X := {s};
E := %+{g{u}: u in X};
N := E-X;
while #N /= 0 do    $ while g[X] - X /= {}
    z := arb(N);
    X := X with z;
    E := E + {u: u in g{z} | u notin E};       $ maintain E = g[X]
    N := (N+ {u: u in g{z} | u notin X}) less z;       $ maintain N =
E-X
end;
$ at this point X = X + g[X]
out_time := clock();
X5 := X;
printf 'duration: ', (out_time-in_time)/60,'\n';
printf 'size: ', #X,'\n';       $ the fixed point size
if X5 /= X4 then
    printf '\ndiscrepancy between 4 and 5\n';
    print X;
end;
t5(i) := (out_time-in_time)/60;

$   -6   eliminate E, which is never used, except for its own
maintance
in_time := clock();

X := {s};
N := g{s} less s; $ %+{g{u}: u in X}-X == g[X] - X
while #N /= 0 do    $ while g[X] - X /= {}
    z := arb(N);
    X := X with z;
    N := (N + {u: u in g{z} | u notin X}) less z;       $ maintain N =
g[X]-X
end;
$ at this point X = X + g[X]
out_time := clock();
X6 := X;
printf 'duration: ', (out_time-in_time)/60,'\n';
printf 'size: ', #X,'\n';       $ the fixed point size
if X6 /= X5 then
    printf '\ndiscrepancy between 5 and 4\n';
    print X;
end;
t6(i) := (out_time-in_time)/60;

end; $ end demo_fd

display_results := func();
    all_fddata := [n_size,e_size,m_size,x_size, t1,t2,t3,t4,t5,t6];
    printf '\n','_'*63,'\n';
    printf '\n n_size,e_size,m_size,x_size, t1,t2,t3,t4,t5,t6\n';
    print all_fddata;

    printf '\n','_'*60,'\n';
    printf '\n',['n','e','m','x', 't1','t2','t3','t4','t5','t6']:6, '\n';

    printf '\n','_'*60,'\n';
    printf '\n', [[n_size(i),e_size(i),m_size(i),x_size(i),
        t1(i),t2(i),t3(i),t4(i), t5(i), t6(i)]: i in [1..7]] : 10*([6.02,' ']
with '\n') ;

    printf '\n','_'*60,'\n';
end; $ end display_results


!echo on

dependence := {     [1,2], [1,4], [1,6],
            [2,3], [2,5], [2,7],
            [4,5], [4,7],
            [6,7], [6,8], [6,9]
        };

domain(dependence);
range(dependence);
s := 7;
m := 3;
demo_fd(dependence,1,s,m);


dd1 := {[random(100),random(100)]: i in [1..100]};
demo_fd(dd1,2);

dd2 := {[random(200),random(200)]: i in [1..200]};
demo_fd(dd2,3);

dd3 := {[random(50),random(50)]: i in [1..200]};
demo_fd(dd3,4);

dd4 := {[random(150),random(150)]: i in [1..500]};
demo_fd(dd4,5);

dd5 := {[random(450),random(450)]: i in [1..1000]};
demo_fd(dd5,6);

dd6 := {[random(1000),random(1000)]: i in [1..1000]};
demo_fd(dd6,7);

display_results();


$ save('all_fddata','all_fddata');
!recordOutput
!echo off
```

the execution trace

The execution is quite instructive. The first observation is that these fixed-point methods compute exactly the desired set augmented with the source node. The second observation is the noticeable improvement of the efficiency of the algorithm, from the naive version to the elaborate finite differencing solution, both in the mechanical and the intuitive approaches. Even though Cantor's implementation of the collection data structures is far from meeting the criterias of that of an ideal set machine, the experimental results indicate, an 'average behaviour' compatible with that of such a machine. The lack of talent of the Cantor designers is not the only thing to blame: there are some significant theoretical limitations; a fixed data structure for representing sets cannot satisfy always the requirements of the ideal set machine. Actually, each algorithm may require a different implementation of that ideal set machine, since each algorithm uses sets in specific ways. Cantor being an interpreted language, has to interpret each instruction, each expression evaluation request, without taking in consideration the algorithm as a whole and possible optimisation information. However Paige technique of Real-Time Simulation of a Set Machine on a RAM[5] may be in the not too distant future available to compile Cantor programs into an appropriate target development language (e.g. C or C++) using the appropriate set implementation, meeting each time the requirements of an ideal set machine.

In what follows, durations are in seconds. They have been evaluated on a Macintosh LC475 by dividing a tick count by their frequency (60 per sec). The reported execution time includes the time spent by the Cantor system 'garbage collecting', i.e. performing essential asynchronous dynamic memory management functions, not specific to any particular algorithm. Here are the execution results:

[5]See for instance Paige R. : Real Time Simulation of a Set machine on a RAM in ICCI `89, ed. W. Koczodaj, Computing and Information, 2, 1989.

```
> dependence := {  [1,2], [1,4], [1,6],
>>              [2,3], [2,5], [2,7],
>>              [4,5], [4,7],
>>              [6,7], [6,8], [6,9]
>>          };
>
> domain(dependence);
{4, 6, 1, 2};

> range(dependence);
{7, 9, 8, 5, 6, 2, 3, 4};

> s := 7;
> m := 3;
> demo_fd(dependence,1,s,m);
n_size(i) :
9;
s : 7;
m : 3;
g{s} : {4, 2, 6};
duration:          0.03333
size:      5
{7, 6, 2, 1, 4};
duration:          0.06667
size:      5
duration:          0.01667
size:      5
duration:          0.01667
size:      5
duration:          0.06667
size:      5
duration:          0.03333
size:      5
OM;

>
>
> dd1 := {[random(100),random(100)]: i in [1..100]};
> demo_fd(dd1,2);
n_size(i) :
57;
s : 57;
m : 5;
g{s} : {42, 48, 49, 12, 25};
duration:          0.26667
size:      20
{70, 59, 80, 88, 57, 53, 52, 49, 43, 48, 46, 12, 14, 15, 25, 29,
 35, 38, 41, 42};
duration:          0.20000
size:      20
duration:          0.05000
size:      20
duration:          0.41667
size:      20
duration:          0.16667
size:      20
duration:          0.23333
size:      20
OM;

>
> dd2 := {[random(200),random(200)]: i in [1..200]};
> demo_fd(dd2,3);
n_size(i) :
124;
s : 152;
m : 5;
g{s} : {158, 103, 118, 6, 86};
duration:          1.18333
size:      38
{114, 118, 105, 123, 129, 96, 103, 87, 94, 137, 142, 152, 131,
 135, 186, 184, 192, 187, 158, 160, 179, 163, 9, 11, 7, 6, 26,
 38, 44, 47, 75, 81, 86, 84, 69, 65, 64, 54};
duration:          1.30000
size:      38
duration:          0.26667
size:      38
duration:          1.53333
size:      38
duration:          0.70000
size:      38
duration:          0.46667
size:      38
OM;

>
> dd3 := {[random(50),random(50)]: i in [1..200]};
> demo_fd(dd3,4);
n_size(i) :
50;
s : 5;
m : 7;
g{s} : {31, 11, 9, 1, 50, 35, 33};
duration:          0.88333
size:      50
{34, 35, 33, 32, 38, 36, 37, 50, 49, 48, 47, 46, 43, 44, 45, 41,
 42, 39, 40, 11, 10, 12, 9, 8, 6, 7, 2, 3, 5, 4, 1, 0, 31, 30,
 29, 21, 22, 23, 26, 28, 24, 25, 20, 19, 18, 17, 16, 14, 15, 13};
duration:          1.08333
size:      50
duration:          0.51667
size:      50
duration:          2.96667
size:      50
duration:          1.68333
size:      50
duration:          0.96667
size:      50
OM;

>
> dd4 := {[random(150),random(150)]: i in [1..500]};
> demo_fd(dd4,5);
n_size(i) :
147;
s : 55;
m : 11;
g{s} : {82, 113, 126, 127, 138, 129, 49, 71, 75, 18, 1};
duration:          4.15000
size:      143
{147, 148, 149, 150, 145, 143, 144, 142, 141, 140, 139, 127,
 128, 129, 131, 132, 133, 134, 138, 137, 135, 136, 117, 116,
 115, 114, 119, 118, 123, 121, 120, 124, 125, 126, 109, 110,
 112, 113, 108, 107, 106, 105, 104, 101, 102, 103, 96, 98, 97,
 100, 99, 93, 94, 95, 83, 82, 80, 78, 79, 77, 76, 84, 85, 86,
 87, 90, 89, 88, 91, 92, 48, 49, 50, 52, 51, 54, 53, 55, 45, 47,
 46, 43, 44, 42, 40, 41, 38, 39, 56, 57, 58, 59, 62, 63, 61, 60,
 72, 73, 75, 74, 70, 71, 69, 67, 68, 66, 64, 0, 1, 2, 6, 7, 8,
 9, 5, 3, 4, 18, 16, 17, 14, 15, 10, 11, 12, 13, 26, 27, 25, 24,
 23, 21, 22, 20, 19, 31, 30, 28, 29, 34, 33, 35, 36};
duration:          5.18333
size:      143
duration:          1.53333
size:      143
duration:          38.98333
size:      143
duration:          5.33333
size:      143
duration:          2.86667
size:      143
OM;

>
> dd5 := {[random(450),random(450)]: i in [1..1000]};
> demo_fd(dd5,6);
n_size(i) :
398;
s : 375;
m : 7;
g{s} : {35, 174, 278, 222, 375, 372, 305};
duration:          22.28333
size:      391
{113, 112, 107, 106, 111, 109, 120, 121, 119, 118, 115, 117,
 116, 90, 91, 89, 88, 92, 93, 95, 94, 98, 97, 103, 104, 101,
 102, 99, 100, 124, 125, 122, 123, 128, 127, 126, 132, 131, 130,
 133, 134, 135, 139, 138, 137, 136, 164, 163, 162, 161, 160,
 159, 167, 166, 165, 171, 170, 169, 168, 172, 174, 173, 142,
 143, 140, 141, 145, 144, 146, 147, 149, 148, 150, 151, 154,
 152, 153, 158, 157, 155, 156, 75, 76, 73, 74, 71, 72, 70, 69,
 67, 66, 61, 60, 63, 64, 62, 77, 78, 81, 80, 82, 84, 86, 87, 85,
 23, 25, 22, 31, 28, 26, 27, 13, 15, 17, 16, 19, 18, 20, 21, 12,
 11, 10, 9, 7, 6, 0, 1, 5, 3, 2, 57, 59, 56, 55, 52, 51, 53, 45,
 46, 42, 44, 43, 48, 47, 50, 49, 36, 37, 38, 39, 40, 35, 34, 33,
 32, 298, 297, 300, 299, 301, 302, 305, 304, 303, 312, 311, 314,
 310, 309, 308, 306, 316, 317, 320, 321, 319, 322, 323, 325,
 324, 283, 282, 279, 280, 281, 287, 285, 286, 291, 289, 290,
 295, 296, 294, 293, 292, 274, 272, 275, 278, 277, 269, 271,
 268, 266, 267, 252, 250, 251, 249, 248, 246, 245, 257, 256,
 255, 262, 261, 265, 264, 258, 260, 259, 176, 177, 178, 182,
 184, 183, 180, 181, 186, 185, 187, 190, 189, 188, 194, 192,
 196, 197, 195, 198, 199, 201, 200, 207, 206, 203, 204, 202,
 210, 208, 212, 213, 215, 230, 231, 229, 228, 227, 226, 225,
 223, 224, 222, 221, 219, 220, 218, 216, 217, 239, 238, 242,
 241, 243, 244, 232, 233, 235, 236, 237, 372, 370, 371, 367,
```

```
 369, 368, 359, 358, 361, 360, 352, 353, 355, 364, 363, 366,
 365, 350, 349, 346, 342, 341, 343, 345, 340, 339, 338, 337,
 331, 332, 333, 334, 335, 336, 326, 327, 329, 330, 328, 373,
 375, 379, 377, 389, 388, 387, 386, 383, 380, 384, 385, 391,
 390, 392, 393, 395, 394, 399, 398, 396, 397, 400, 401, 402,
 403, 404, 405, 407, 406, 409, 408, 411, 412, 410, 413, 414,
 415, 417, 416, 429, 428, 426, 427, 424, 425, 419, 418, 421,
 422, 420, 432, 435, 431, 430, 437, 436, 441, 444, 438, 440,
 439, 449, 450, 448, 446, 445, 447};
duration:        25.21667
size:     391
duration:         3.76667
size:     391
duration:       187.05000
size:     391
duration:        10.43333
size:     391
duration:         6.73333
size:     391
OM;

>
> dd6 := {[random(1000),random(1000)]: i in [1..1000]};
> demo_fd(dd6,7);
n_size(i) :
629;
s : 614;
m : 5;
g{s} : {106, 172, 522, 754, 807};
duration:         2.50000
size:     94
{774, 782, 771, 763, 765, 544, 555, 561, 560, 562, 565, 622,
 614, 638, 636, 703, 715, 754, 875, 871, 886, 895, 908, 909,
 911, 931, 939, 932, 961, 949, 988, 974, 813, 832, 814, 807,
 794, 858, 857, 847, 849, 281, 283, 280, 296, 284, 188, 182,
 232, 210, 197, 252, 263, 270, 273, 97, 78, 34, 106, 98, 33, 30,
 4, 28, 168, 170, 172, 156, 163, 125, 121, 110, 112, 300, 304,
 311, 369, 370, 344, 350, 397, 391, 382, 411, 427, 459, 476,
 474, 515, 522, 496, 514, 493, 485};
duration:         2.40000
size:     94
duration:         0.40000
size:     94
duration:         4.03333
size:     94
duration:         1.20000
size:     94
duration:         0.80000
size:     94
OM;

>
> display_results();
```

```
n_size,e_size,m_size,x_size, t1,t2,t3,t4,t5,t6

[[9, 57, 124, 50, 147, 398, 629],
 [11, 99, 200, 191, 493, 996, 1000], [3, 5, 5, 7, 11, 7, 5],
 [5, 20, 38, 50, 143, 391, 94],
 [0.033, 0.267, 1.183, 0.883, 4.150, 22.283, 2.500],
 [0.067, 0.200, 1.300, 1.083, 5.183, 25.217, 2.400],
 [0.017, 0.050, 0.267, 0.517, 1.533, 3.767, 0.400],
 [0.017, 0.417, 1.533, 2.967, 38.983, 187.050, 4.033],
 [0.067, 0.167, 0.700, 1.683, 5.333, 10.433, 1.200],
 [0.033, 0.233, 0.467, 0.967, 2.867, 6.733, 0.800]];
```

n	e	m	x	t1	t2	t3	t4	t5	t6
9	11	3	5	0.03	0.07	0.02	0.02	0.07	0.03
57	99	5	20	0.27	0.20	0.05	0.42	0.17	0.23
124	200	5	38	1.18	1.30	0.27	1.53	0.70	0.47
50	191	7	50	0.88	1.08	0.52	2.97	1.68	0.97
147	493	11	143	4.15	5.18	1.53	38.98	5.33	2.87
398	996	7	391	22.28	25.22	3.77	187.05	10.43	6.73
629	1000	5	94	2.50	2.40	0.40	4.03	1.20	0.80

```
OM;

>
>
> $ save('all_fddata','all_fddata');
> !recordOutput
! Recording Output is off
> !echo off
>
```

comments on this execution trace

Just for the sake of legibility, we have used for displaying the formatted results derived from invoking *display_results()* the standard Cantor console font (monaco) .
The following will help you understand these results:

col. label	*description*
n	number of nodes in the graph
e	number of edges in the graph
m	max. nbr. of children at any node
x	cardinality of the fixed point set
t1	run-time for naive fixpoint algorithm
t2	run-time for naive finite diff. algorithm
t3	run-time for hand-crafted finite diff. algorithm
t4	run-time for 1st level mechanical finite diff.- algorithm
t5	run-time for 2nd level mechanical finite diff.- algorithm
t6	run-time for mechanical finite diff.- algorithm, with dead code elimination

From the displayed results it is clear that the best version of the algorithm is the one corresponding to column t3, i.e. the hand-crafted finite differencing version, which made use of an identity, which an automated program transformation system could not derive from the naive fixed point definition of the problem. The algorithm derived from a mechanical application of finite differencing with dead-code elimination displays comparable results (in column t6), within a constant multiplicative factor.

Exercises

- what happens if one adds an arrow 7->1, represented by the pair [7,1] in the dependence graph. Hint: Compare the pre-requisites of 7 with those of 1,2, 4 ,6 or any other node.

- the pre-requisite analysis was carried out for a single source situation: find the pre-requisites for a single chapter, find the reacheability set for a single source node. Restate this analysis and the algorithms for multiple sources
- could the algorithms be made more efficient, if instead of computing the whole reachability or pre-requisites set of a given source, the problem was to test whether a specific node belongs to that set?
- let #X be the cardinality of the prerequisites set. Show that the optimized versions of the algorithm have an asymptotic time complexity O(#X), thus may only be improved by constant factors - complexity wise, a marginal improvement. Verify this by testing with numerous graph configurations.

Example 4: pre-requisites - revisited-2

Here we will look into the issue of presenting correctly, the prerequisites, in an order compatible with that of the given dependence graph. This compatible order could be defined as follows:
let S be the given set, let D be the dependence graph, let T be the sorted collection.
If a, b are elements of S and a is a pre-requisite of b according to D, then a should precede b in T.
In example 4, S is {2,1,4,6}, and D is dependence, the graph represented by

```
{    [1,2], [1,4], [1,6],
     [2,3], [2,5], [2,7],
     [4,5], [4,7],
     [6,7], [6,8], [6,9]
};
```

Since 2 is not a prerequisite to 4 or 6, nor 6 a prerequisite to 2 or 4 , nor 4 a prerequisite to 2 or 6 then 2, 4, 6 may be put into T in any order relative to one another. However 1 is a prerequisite to 2, 4 and 6. Therefore 1 should be placed ahead of 2,4,6. Therefore [1,2,4,6] or [1] followed by any permutaion of {2,4,6} is an acceptable solution.

Any such acceptable ordering of S is called a topological order of S (relative to the order specified by D).

partial orders, transitive closures

A relation R is a partial order relation if:
-it is antisymmetric: a R b & b R a imply a = b
-it is transitive: a R b & b R c imply a R c
The dependence graph of example 4 is closely related to a partial order relation but is not a partial order. Indeed, one has 1->2 and 2->3 in the graph, but not 1->3 : transitivity is violated. Adding all the missing edges, to meet the transitivity requirement is called 'computing the transitive closure'. This is easily done, e.g. by means of any of the single source graph reachability algorithms we have seen, for instance the naive one:

```
X := {s};
f := func(d); return d%+{g{x}: x in d}; end; $ f(D) is D+g[D]
T := f(X);
while X /= T do
     X := T;
     T := f(T);
end;
X := X less s;   $ remove the source from X
$ at this point X is the fixed-point i.e.
$ the set of all non-trival nodes
$ in g reachable from the single source s
```

One could augment g by adding all the edges from the source s to the reachable nodes:
g := g + {[s,u]: u in X};
And then doing this for every possible source node in g, i.e. in domain(g):

```
for s in domain(g) do
     X := {s};
     f := func(d); return d%+{g{x}: x in d}; end;$ f(D) is D+g[D]
     T := f(X);
```

```
    while X /= T do
          X := T;
          T := f(T);
    end;
    $ at this point X is the fixed-point i.e.
    $the set of all non-trival nodes
    $ in g reachable from the single source s
    X := X less s;     $ remove the source
    $ augment g with edges linking s to each reachable node
    g := g + {[s,u]: u in X};
end;
$ at this point g contains its transitive closure
```

A map g representing a binary relation is a pre-partial order if when it is augmented by its transitive closure it is a partial order.

topological sort

We need, for computing the topological sort, to be sure that our dependence graph D is a pre-partial order. However, we will see that we don't need to compute the transitive closure.

If D is really a pre-partial order, all we need is to find the elements of S which have no pre-requisites (in S), according to D, put them into an ordered collection T, and remove them from S. And we do this until S becomes empty.

```
$ given graph D, input set S
T := []; $ an empty ordered collection
g := {[y,x]: [x,y] in D}; $ g is the inverse graph of D
$ a node has no predecessor in D iff it has no successor in g
no_predecessors := {x: x in S | #(g{x}*S) = 0 };
while #S /= 0 and #no_predecessors /= 0 do
    T := T+[x: x in no_predecessors ];
    S := S - no_predecessors ;
    no_predecessors := {x: x in S | #(g{x}*S) = 0 };
end;
$ at this point S should be empty
$ and T is the sorted collection
```

This is of course a much simpler algorithm, which lends itself to numerous improvements.

Exercises

- we exposed in a previous section 4 algorithms for the pre-requisite problem. Adapt them all to the computation of the transitive closure. What can be said about their efficiency?
- compare this presentation of a transitive closure algorithm with that of Warshall algorithm in your prefered text book.
- carry out, on the exposed topological sort algorithm, the finite differencing analysis. Propose an efficient algorithm, test and compare the results.
- what is the fixed-point computed by the topsort algorithm?
- given a dependence graph D as above, how can we test if it is a pre-partial order or not? Hint 1: produce an algorithm derived from that of the topological sort Hint 2: prove that D is in pre-partial order iff D has no cycle, i.e. there is no node in D which is contained in its own pre-requisites set.
- define 'sorting' (i.e. re-arranging a collection of strings or numbers in ascending order) as a fixed-point problem. Hint: use the merge function exposed in example 2.

Example 5: a simple real-time system: a digital watch-revisited

We have already identified the three main collections in Statecharts:
- the set of blobs, representing the individual states of the system being described, as well as groupings of such states

- the set of subsystems, representing the breakdown of a blob into concurrent subsystems
- the transitions (labelled arrows)

What is essential is the representation of state groupings, nestings and decompositions. We will therefore concentrate on the maps describing these relations:

- `blobs` represents the blobs nesting association
- `subsystems` represents the blobs decomposition into susbsystems, i.e. partitions
- `transitions` is the map representing the associations drawn by the arrows

Initially these maps are empty:

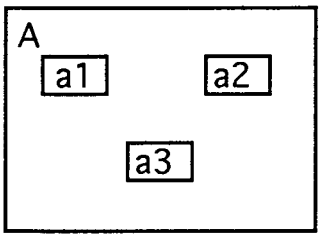
```
blobs := {};
subsystems := {};
transitions := {};
```

More collections will be needed. For instance there is a collection which plays an important pragmatic role, but not a conceptual one: the collection of identifications for all the objects in the Statechart. We assume that each object is assigned a unique atom, and that each atom is associated to a text string providing a name for that object. As a matter of fact we will separate state names from event names. E.g. whenever a new blob A is introduced we will need the following instructions (or equivalent) :

```
A := newat;              $ create an atom
state_name(A) := 'A';    $ associate a name
```

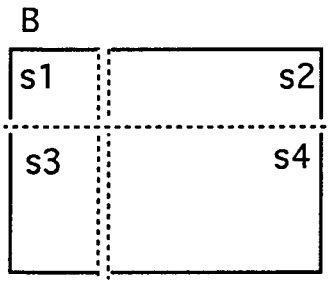
and analogously for each new event, an atom will be associated with an event_name.

The set `blobs` reproduces the blob's hierarchy and the set `subsystems` reproduces the subsystem hierarchy as in the following diagram examples:



blob A contains blobs a1,a2,a3

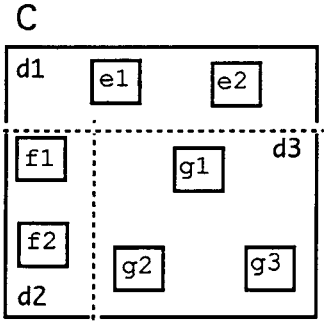
```
[a1,a2,a3] := [newat,newat,newat];
state_name(a1) := 'a1';
state_name(a2) := 'a2';
state_name(a3) := 'a3';
$ reproduce the membership ai in A
blobs{A} := {a1,a2,a3};
```



blob B is partitionned into subsystems s1,s2,s3,s4

```
[s1,s2,s3,s4] :=
 [newat,newat,newat,newat];
state_name(s1) := 's1';
state_name(s2) := 's2';
state_name(s3) := 's3';
$ reproduce the membership si in B
subsystems{B} := {s1,s2,s3,s4};
```

The following diagram mixes both kind of hierarchies:




```
C := newat;
state_name(C) := 'C';
[d1,d2,d3] := [newat,newat,newat];
subsystems{C} := {d1,d2,d3};
[f1,f2,e1,e2] :=
        [newat,newat,newat,newat];
[g1,g2,g3] := [newat,newat,newat];
blobs{d1} := {e1,e2};
blobs{d2} := {f1,f2};
blobs{d3} := {g1,g2,g3};
state_name(d1) := 'd1';
....
state_name(g3) := 'g3';
```

Observe that the set of all individual states is exactly

```
states := domain(blobs)+range(blobs) + domain(subsystems) +
```

```
range(subsystems);
```
Another - should be equivalent - definition is:
```
states := domain(state_name);
```

These examples illustrate the straightforward use of maps to represent blobs and subsystems hierarchy.

the different types of transitions

We will consider the following different kind of transitions:

1-the ordinary case : an arrow labelled by an event name between an origin state and any number of target states (N.B. in fig. 3 there is one instance of an arrow pointing to more than one target state : in blob *stopwatch* from state *zero* to states *reg* and *on*, labelled by event *b*). A transition means : if the origin state is active, then it is de-activated, and the target states become active. An ordinary transition from state *a* to state *b* labelled by *e* is represented by a term [[*a,e*],*b*] i.e., if there is no other transition from state *a*, labelled by *e*, [*a,e*] is uniquely associated to the target state *b*:
```
transitions([a,e]) := b; $ assume no other transition from [a,e]
```
An ordinary transtition from *a* to a set of states {*b1,b2*, ...} labelled by event *e* is similarly represented by:
```
transitions{[a,e]} := {b1,b2, ...};
```

2-the conditional transition : the transition label has a part surrounded by [] which specifies a boolean condition on specific state activation : the condition *in(s)* holds whenever the state *s* is active; this is implemented by the predicate *in_state(s)* where *in_state* is a boolean valued map. This means transition may take place only if the condition holds. A conditional transition from state *a* to state *b* labelled by *e* under condition *in_state(s)* is represented by a term [[*a,e*],[*b, 'in_state(s)'*]] i.e. if there is no other transition from [a,e] then:
```
transition([a,e]) := [b, 'in_state(s)'];
```
The condition may be complex, e.g., *in_state(s) and not in_state(u)*, and other atomic predicates than *in_state* may be considered. The string will be analyze-d(see section 6.3 of the user manual), the resulting abstract syntax tree will be evaluated, and depending upon the value the transition will yield an activation change or not.

Another collection has been introduced : the map *in_state* : states->boolean. Actually this map maintains the activation status. Thus, the 'current state' -which comprises the activated blob and all the derived states in nested blobs and subsystems is defined as:
```
curState := {x: x in states | in_state(x)};
```
When a transition from *a* to *b* takes place, it is because *in_state(a)* , i.e. *a* ∈ curState, then the de-activation of *a* takes place and is followed by an activation of *b*:
```
in_state(a) := false;  $ de-activation of a
in_state(b) := true;   $ activation of b
```
A transition represents an explicit activation request. It should be followed by the activation of all the derived states, all the states which are implicitly activated.

3-the transition with broadcasting of an event : the transition label is of the form *trigger_event / broadcast_event*. This transition behaves like an ordinary transition. As the target state becomes active, the event *broadcast_event* is sent to all the blobs and subsystems, and therefore could trigger other transitions. Broadcasting could generate chain-reactions, since event propagation follows immediately the target events activation. In fig. 3 there is only one example of a broadcast, the transition labelled *bt_rm/clh* (battery removed / clear history) between the blobs *alive* and *dead*. There is however no way of knowing the effect of the event *clh*, since it is not described in fig. 3.

4-the transition to a H* point inside a blob : this symbol represents the last active state in the blob before it was last de-activated. If that blobs comprises subsystems, this 'most recently active' state contains the 'most recently active' state of each of the subsystems partition, and so on, until all the nested individual states which were 'most recently active' sub-parts of the given blob are

detailed. The record of this 'most recently active' state and of its 'most recently active' sub-parts is kept in a history structure. When a transition having as target a H* point is taking place, that 'most recently active' state is re-activated along with its 'most recently active' sub-parts.

E.g. there is a transition triggered by event *a* from *chime* to H*-*stopwatch*. That 'most recently active' state may be one of the following five: *zero*, {*reg, on*}, {*reg, off*}, {*lap, on*}, {*lap, off*}.

We introduce here another map `history` states -> 2 states to record, before the de-activation of a state its 'most recently active' state and embedded states. For instance that map always satisfies:

```
history(stopwatch) in {zero, {reg, on}, {reg, off}, {lap, on},
                       {lap, off}};
```

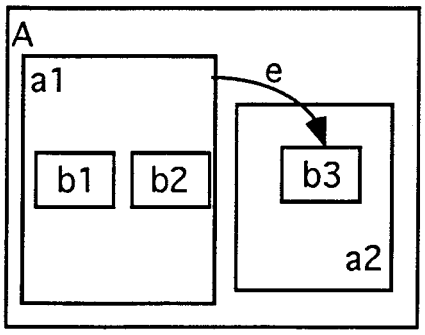
5-the transition to a default state : When a blob is getting activated by a transition -the arrow points at the border of the blob rectangle- no specific state inside that blob is the target of the transition, then the state which gets that activation is the *default state*. The default state is indicated by a non-labelled arrow originating at a dot, and pointing to the default state. For instance, the default state of the *watch* is *dead*, the default state of the *power* is *ok*, the default state of *light* is *off*, etc.. Unlike transitions to a H* point inside a blob, a transition to a default state has no specific denotation in our map `transitions`; it is while calculating where and how an activation propagates that a default state activation is uncovered.

The map default : states -> states records these association between a blob and its default state:

```
default(watch) := dead;
default(power) := ok;
default(light) := off;
```

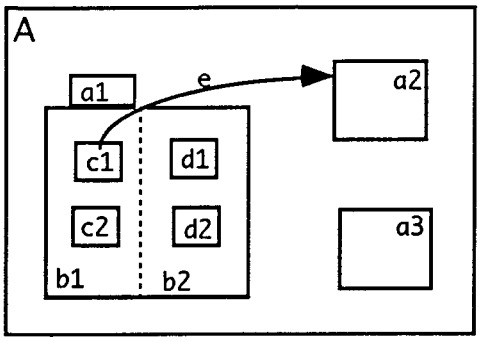
propagation of activation and de-activation

We have already been alluding to activation or de-activation propagation. We will try to be exhaustive:
- whenever a transition gets a target state *b* activated, then implicitly, all the blobs containing b are activated, and this propagates to all the blobs containing a blob including b, and so on. We call this *state membership propagation up the blobs.*
- whenever a transition gets a target state *b* activated, then implicitly, *b*'s default state gets activated, and this propagates to the default state of *b*'s default state, and so on. We call this *default state activation propagation.*
- whenever a transition gets a target state *b* activated, if *b* is actually a blob partitioned into subsystems, then implicitly, each of the subsystems default state gets activated, and so on down the default states and subsystems. We call this *the full default state activation propagation.*
- whenever a transition from state *a* gets a target state *b* activated, then state *a* gets de-activated, and all the states and subsystems of *a* which are not contained directly or indirectly in *b* should be de-activated too. We present here two different situations:



a transition represented by [[a1,e],b3] :

all the states contained in *a1* and not contained in *b3* should be de-activated. We call this *deactivation propagation across states*



a transition represented by [[c1,e],a2] :

it is insufficient to de-activate all the states in *c1* not contained in *a2*. This transition leaves completely the partitionned blob *a1*, whence all the subsystems of *a1* should be de-activated. Since they are disjoint sets of states, all the states contained in *a1* should be de-activated
We call this *deactivation propagation out of subsystems*

An actual transition may require a combination of these two kinds of de-activation processes. This will be the case of a transition from a deeply nested state, whithin a subsystem, to one of the top states in the blobs hierarchy. Fig. 3 exhibits all kinds of combinations of these de-activation schemes: e.g. the transition represented by the arrow from *displays* to *beep* labelled *t_hits_tm [in(enab)]* .

propagation and closures

All the above propagation processes are instances of the 'pre-requisites' problem or of its inverse, the reacheability problem. In each case the main computational tool is the efficient function for computing all the nodes reachable in a graph g from a given source s:

```
$------------------------------reach_
reach_ := func(g,s);
local X,T,dN,N0;

    X := {s};
    $     T = X%+{g{x}: x in X} i.e.  T = X + g[X]
    T := g{s} with s;
    dN := T-X;
    $ X /= T iff T-X /= {} iff g[X] - X /= {} iff #dN /= 0
    while #dN /= 0 do
          $ N0 = g[T] - T
          N0 := {y: u in dN,y in g{u} | y notin T};
          $     achieve T = X + g[X]
          X := T; $ X+dN;
          T := T + N0;
          $     assert T = X + g[X]
          dN := N0;
          $     assert dN = g[X] - X = T-X
    end;
    $ at this point X = X + g[X], T = X
    $ the set of all non-trival nodes
    $ in g reachable from the single source s
    return X less s;
end; $ end reach_
```

The main design problem is to define each time what is exactly the graph g, and how it is related to the already identified collections.
Let us consider all the above listed cases:
-*state membership propagation up the blobs.*: we need to find, given a target state *b*, which blobs *x* satisfy *b* in blobs{*x*} or *b* in subsystems{*x*}. We have to find which are the pre-requisites or predecessors of *b* in the graph blobs ∪ subsystems. The solution consist in introducing the inverse of graph blobs ∪ subsystems. Let it be called blobs_of:

```
blobs_of := {[y,x]: [x,y] in blobs + subsystems };
```

the propagation set 'up the blobs' from a given state *b* is then exactly:

```
reach_(blobs_of ,b);
```

-*default state activation propagation.* has seemingly a very similar solution: given a state *b* the set of all the directly or indirectly default states to activate would be

```
reach_(default,b);
```

However this is true only if there are no partitioned states and subsystems in this set. If *c* is a partitioned state in the collection of direct or indirect default states of blob *b*, *c* has no default_state, only its subsystems may have default states. The computation of the propagation set continues by adding all the default states of the blobs in

```
subsystems{c}
```

and the indirect ones derived from them. The difficulty here is that the default map disconnects simple blobs from partitioned ones. For instance, in the example fig. 3, the state *alive* has no default state, thus the `reach_` function cannot find default states corresponding to a partitionned state.
The solution is to redefine the map `default` by adding associations between all the partionned state and the default state in each subsystem. In the example of fig. 3 this would imply adding

```
default{active} := {ok, off, c_disab, disab};
```

and an arbitrary state to select from {*displays, beep*} - since this was omitted from the chart.
Formally the change is defined by,

```
for x in domain(subsystems) do
    default{x} := {default(y): y in subsystems{x}};
end;
```

And if we want to take into account the omission of defining sometimes default state:

```
for x in domain(subsystems) do
    default{x} := {default(y)?arb(blobs{y}):
                                y in subsystems{x}};
end;
```

I.e., for a given subsystem y, if default(y) is undefined, then take an arbitrary element in the blob y, and let it play the role of the default.

With this modification, the *full default state activation propagation.* set for a given target state b is exactly `reach_(default,b)`.

-*deactivation propagation across states* : all the states which directly or indirectly are contained in the origin_state, and are not directly or indirectly contained in the target_state, need to be de-activated. It is easy to see that all the direct or indirect members of a given state x are in

```
reach_(blobs+subsystems,x);
```

Thus, for a transition from state a to state b, the states which need to be de-activated are those in:

```
reach_(blobs+subsystems,a) - reach_(blobs+subsystems,b);
```

An important subcase, is when the origin and the target state are the same. In that case, the solution is to de-activate all the member states, and let the *full default state activation propagation.* mechanism activate the default states.

-*deactivation propagation out of subsystems* : we have to be able to compare the partitioned blobs in which the origin and the target states belong. Let us assume we have been able to identify that the origin state *a* was in a partitioned blob *s*, and that the target state *b* is not a direct or indirect member of *s*, i.e. if `b notin reach_( blobs + subsystems,s)` then the deactivation concerns all the members of:

```
reach_(blobs+subsystems,s) - reach_(blobs+subsystems,b);
```

We have now to explain how the partitioned blob containing a given state is determined. This is a variant of the above determination of blobs_of.
For convenience, let us call a 'partitionned blob' a 'system'. We have to compute `syst_of`, the map which associates to a blob the system which cantains it. The set of systems is exactly domain(subsystems). To determine to which system a given state x belongs, one has to find if there is a blob y, in domain(subsystems), such that

```
x in reach_(blobs+subsystems,y);
```

Since subsystems may be nested too, there may be more than one such element. All we need, is to find, if there is a y in domain(subsystems), having a subsystem u such that: x in reach_(blobs, u). Since we are searching all the subsystems of y, this is stated formally:

```
x in %+{reach_(blobs, u) : u in subsystems{y}}
```

We have to add the possibility that x itself is a subsystem, i.e. one of these u in subsystems{y}. The full expression is therefore:

```
x in %+{reach_(blobs, u) with u : u in subsystems{y}}
```

We have defined a map:

```
syst_blob := {};
for y in domain(subsystems) do
    syst_blob{y} := {} %+ {reach_(blobs, u) with u:
                                    u in subsystems{y}};
end;
```

which associates to any system all its direct or indirect members, which are not part of another system. The map syst_of is therefore its inverse:

```
syst_of := {[y,x]: [x,y] in syst_blob };
```

Observe that during this investigation, we have been applying the reach_ function over and over again to well-defined graphs: blobs, blobs+subsystems, blobs_of , default. The only change was in the source node used. We could instead compute the transitive closures of these graphs, as was indicated in the pre-requisites example, using the func:

```
closure := func(g);
local X,s,T,PrevNew,New,changed;

for s in domain(g) do
    X := reach_(g,s);
    $ augment g with edges linking s to each reachable node
    g := g + {[s,u]: u in X};
end;       $ end for
$ at this point g contains its transitive closure
return g;
end; $ end closure
```

We will therefore introduce the following closures:

```
Blobs := closure (blobs);
Blobs_n_subs := closure (blobs+subsystems);
Blobs_of := closure(blobs_of);      $ blobs_of is the inverse of
                                    $ blobs+subsystems
Default := closure(default);        $ default is extended with
                                    $ subsystems defaults
```

These closures are computed once and for all, for a given statechart. Then the various propagation sets have very simple forms:

- -*state membership propagation up the blobs.*: for target state b:

```
Blobs_of{b}
```

- -*(full) default state activation propagation.* : for target state b:

```
Default {b}
```

- -*deactivation propagation across states* : for an origin state a and a target state b:

```
if a /= b then
    Blobs_n_subs {a} - Blobs_n_subs {b}
else      Blobs_n_subs {a}
end;
```

- -*deactivation propagation out of subsystems* : we introduce another map syst_of defined as follows:

```
syst_blob := {};
for y in domain(subsystems) do
    syst_blob{y} := {} %+ {Blobs{u} with u:
                                    u in subsystems{y}};
end;
syst_of := {[y,x]: [x,y] in syst_blob };
```

The detection test, to check if a transition from state a to state b is actually 'out of a subsystem' is

```
is_defined(syst_of(a)) and
    (syst_of(a) /= syst_of(b))
    and b notin Blobs_n_subs {syst_of(a)}
```

If this expression is true, the propagation set is then

```
Blobs_n_subs {syst_of(a)} - Blobs_n_subs {b}
```

the include file of the statechart interpreter

```
$
$
$                statechart representation
$                ex: a digital watch according
$            to D. Harel, CACM 31, 5, 1988, pp 514-530
$
$_____________________________________________
—

$----------global data
$----------simul_init
$----------transitions: detailed prescription
$----------reach_
$----------closure
$----------naming
$----------broadcast
$----------perform
$----------simul


$_____________________________________global data
$ several maps to describe the system decomposition
subsystems := {};
blobs := {};
default := {};   $ default state in a blob
history := {};   $ record the last state the blob was in
transitions := {};      $ the event directed transition map: {[state,
[event, state, ...]], ..}

theVerb      := true;       $ the simulation mode:
                $ verbose (by default) or not verbose
theEventQ := [];       $ the broadcast event queue

$ auxiliary map for trace execution
$ give a name to each state suitable for a trace exec.
state_name := state_name?{};
$ give a name to each event suitable for a trace exec.
event_name := event_name?{};
names_ := {};   $ the union of event_name and state_name

$ other auxiliary maps are created at init time: simul_init:
$   states, blobs_of, in_state, Blobs_n_subs
$   Blobs, syst_blob, syst_of, Default


$_____________________________preprocessing:
$_____________________________simul_init

simul_init := func(theSystem, theFirstEvent opt verb);

    $ define the whole set of states
    states := domain(blobs)+range(blobs);
    blobs_of := {[y,x]: [x,y] in blobs};$ blobs inverse map
    in_state := {[x,false]: x in states};       $ all states are
                                $ inactive
    in_state(default(watch)) := true;       $ excepted 'dead'
    $ the current state is {x: in_state(x) = true}

    $ complete the default map by incorporating links between
    $ partitionned states and default states in subsystems
    for [x,y] in subsystems do
        $ find or define a default state
        default{x} :=
            default{x} with default(y)?arb(blobs{y});
    end;

    $ pre-compute maps corresponding to membership relations
    $ or their inverse

    $ the closure of the 'contains' relation
    $ for states and blobs
    Blobs_n_subs    :=   closure(blobs+subsystems);
    Blobs_of        :=   closure(blobs_of);
    Default         :=   closure(default);
    $ given a state, find which system it belongs to:
    Blobs           :=   closure(blobs);
    $ which blobs belong to which syst?
    syst_blob := {};
    for x in domain(subsystems) do
        syst_blob{x} := {} %+ {Blobs{y}: y in subsystems{x}};
    end;

    $ add to syst_blob the sussystems themselves
    syst_blob := syst_blob+subsystems;

    $ find out in which syst a blob is:
    $ given a state st, it belongs to syst syst_of{x}
    $ syst_o is syst_blob inverse map
    syst_of := {[x,y]: [y,x] in syst_blob};

    theEventQ := [];       $ the broadcast event queue
    names_ := state_name+event_name;       $ used by naming

    theVerb := verb?theVerb;
    $ start by launching the theSystem sending it theFirstEvent
    perform(theFirstEvent,theVerb); $ verbose!

end; $ end simul_init

$_____________________________________blobs &
subsystems

$       as blobs are introduced, arbitrary ids are
$       assigned to them: these ids are atoms
$       created by newat.

$       The layout of the system described by a statechart
$       is captured by two maps:
$           blobs, subsystems
$   for documentation and visibility another map is essential:
$           state_name

$       For instance if a new blob A is introduced:
$           A := newat;
$           state_name(A) := 'A';
$       If A is supposed to contain a1,a2,a3 as sub-blobs
$           [a1,a2,a3] := [newat,newat,newat];
$           state_name(a1) := 'a1';
$           state_name(a2) := 'a2';
$           state_name(a3) := 'a3';
$           blobs{A} := {a1,a2,a3};
$       this captures the membership ai in A
$
$       If however, A is decomposed into 'orthogonal'
$       or concurrent subsystems
$       capture A as Harel's 'cartesian product' a1 x a2 x a3
$           subsystems{A} := {a1,a2,a3};
$       Then a1,a2,a3 may be further decomposed into
$       blobs and subsystems

$_____________________________________transitions

$   the transition map is a set of items
$       [[orig_state, event],target_state]
$               (this is the most common case)
$       [[orig_state, event],[target_state, condition]]
$       [[orig_state, event],[target_state, action_event]]
$   in these item representations,
$   target state may be one of the following:
$       an atom             (this is the most common case)
$       'an exprn string defining target_state(s)'
$       ast_exprn_defining_target_state(s)
$   when the target_state is not an atom, the eval function
$   is invoked to compute, at simulation time the value
$   of the string or ast expression. This value should be
$   either a single atom or a set of atoms representing each
$   an individual state
$
$   a string is used to designate a target state as the
$           'history'
$   of a given blob, i.e. the last known state
$   in which that blob has been seen active
$       [state_a, event_a], 'history(blob_m)']
$   represent a transition from state_a, under event_a to the
$   state in which blob_m was left, the last time it was active
$
$   conditions are represented by
$       'an exprn string expressing a boolean'
$       ast_exprn_expressing a boolean
$   when a condition is encountered, it is evaluated
$   by the eval function, at simulation time. Only if the
$   boolean is true, the corresponding target state(s) is (are)
$   added to the actual transition set from the given
$   orig_state.


$ a priori transition is not a smap ....


$_____________________________main processing
functions:
$   compute the closure of the relation state in blob so&so
$   to compute this closure, there is no diff. betw blobs
$   and subsystems
```

```
$   compute the closure of blobs+subsystems:

$--------------------------------reach_
reach_ := func(g,s);
local X,T,dN,N0;

    X := {s};
    $    T = X%+{g{x}: x in X} i.e.  T = X + g[X]
    T := g{s} with s;
    dN := T-X;
    $ X /= T iff T-X /= {} iff g[X] - X /= {} iff #dN /= 0
    while #dN /= 0 do
        $ N0 = g[T] - T
        N0 := {y: u in dN,y in g{u} | y notin T};
        $    achieve T = X + g[X]
        X := T; $ X+dN;
        T := T + N0;
        $    assert T = X + g[X]
        dN := N0;
        $    assert dN = g[X] - X = T-X
    end;
    $ at this point X = X + g[X], T = X
    $ the set of all non-trival nodes
    $ in g reachable from the single source s
    return X less s;
end; $ end reach_


$--------------------------------closure
$   computes the transitive closure of
$   the graph g
closure := func(g);
local X,s,T,PrevNew,New,changed;

    for s in domain(g) do
        X := reach_(g,s);
    $ augment g with edges linking s to each reachable node
        g := g + {[s,u]: u in X};
    end;     $ end for
    $ at this point g contains its transitive closure
    return g;
end; $ end closure

$--------------------------------naming
$   in all the global data, atoms
$   are used to designate states, blobs, systems, events
$   invoking naming(g) allows a display of the same information
$   where atoms are replaced by strings
naming := func(g);
    local gn;
    if is_atom(g) then return names_(g);
    elseif is_string(g) or is_number(g)
        then return g;
    elseif is_tuple(g) then return [naming(x): x in g];
    elseif is_set(g) then return {naming(x): x in g};
    else
        printf '\ninvalid type\n';
        return g;
    end;
end; $ end naming


$--------------------------------broadcast
$ the interpreter:
$ for each event, computes all the possible transitions
$ from the current state

$   simple broadcasting actions consists in:
$   storing the events in a queue expecting that
$   at the end of each perform-ance, the interpreter will
$   empty the queue
broadcast := func(anEvent);
    theEventQ := theEventQ with anEvent;
    return;
end; $ end broadcast


$--------------------------------perform
$ the interpreter:
$ for each event, compute all the possible transitions
$ from the current state

$ perform is a very simple interpreter:
$   -it computes the current state
$   -it computes the transitions in all the sub-systems
$       of the current state to another ste
$   perform supports the conditional events: conditions
$   are expressed as strings or ast's, which are evaluated
$   on demand, 'dynamically'

perform := func(anEvent opt verb);
    local curState, allTrans,v_state,cond;
    local df, orig;
    curState := {x: x in states | in_state(x)};
    for x in domain(history) do
        $ record history if there is something to record
        df := Blobs_n_subs {x}*curState;
        if df /= {} then
            history(x) := df;
        end;
    end;
    if verb then
        printf '\nevent: ',event_name(anEvent), '\n';
        printf ' current_state :',
        { state_name(x): x in curState}: 5*[9] with '\n';
        printf '\n';
    end;

    for x in curState | in_state(x) do
        $ enumerate all the transitions from the current state
        allTrans := transitions{[x,anEvent]};
        $ sort out all the possible cases and
        $ compute accurately the set of transitions
        for u in allTrans do
            cond := om;
            if is_tuple(u) then
                [v_state,cond] := u;
            else
                v_state := u;
            end;
            if is_string(v_state) then
                $ compute the state from the expression
                v_state := eval(analyze(v_state)(1));
            elseif is_ast(u) then
                $ compute the state from the expression
                v_state := eval(v_state);
            end;

            if is_string(cond) then
            $ compute the condition from the expression
                cond := eval(analyze(cond)(1));
            elseif is_ast(cond) then
            $ compute the condition from the expression
                cond := eval(cond);
            elseif is_atom(cond) then
            $ cond is not a condition but an action event!
                broadcast(cond);
                cond := true;
            end;
            $ update the definition of the set of
            $ all simultaneaous transitions
            if is_om(cond) or cond then
                if is_set(v_state) then
                $ this is the case when history
                $ is getting 'rich'!
                  allTrans := (allTrans less u) +
                                v_state;
                elseif u /= v_state then
                  allTrans := (allTrans less u) with
                                v_state;
                end;
            elseif not cond then
                allTrans := allTrans less u;
            end;
        end; $ end for u in allTrans
        $ now the syntactically aspects of allTrans have been
        $ completely processed
        if #allTrans /= 0 then in_state(x) := false; end;
        for v_state in allTrans do
            in_state(v_state) := true;
            $ update all derived states
            $ is x->v_state is a transition
            $ out of a subsystem?
            if x = v_state then
                $ de-activate all the members of x
                $ the default states will be re-activated
                for v in Blobs_n_subs {x}*curState do
                  in_state(v) := false;
                end;
            else
                orig := x;
                $ compare syst_of(x) and syst_of(v_state):
                $ is this an out-going transition?
                $ an in transition satisfies:
```

```
        $ v_state in Blobs_n_subs {syst_of(x)}
        if is_defined(syst_of(x)) and
          (syst_of(x) /= syst_of(v_state))
          and v_state notin
              Blobs_n_subs {syst_of(x)} then
          $ de-activate all the sates in
          $ Blobs_n_subs {syst_of(x)}
          orig := syst_of(x);
          in_state(orig) := false;
        end;
        for v in Blobs_n_subs{orig}*curState -
          Blobs_n_subs {v_state}*curState do
          in_state(v) := false;
        end;
      end;
      if blobs_of{x} /= blobs_of{v_state} then
        $ propagate state membership up the blobs
        df := Blobs_of{v_state} with v_state;
        for v in df do
          in_state(v) := true;
        end;
      end;
      if verb then
        printf 'transition from : ',
          state_name(x),' to ',
          state_name(v_state),'\n';
      end;
      $ is there a default state?
      $ since default is augmented with the subsystem
            $ the reachability closure contains all that is
            $ needed!
            df := Default{v_state};
            for xf in df do
              in_state(xf) := true;
              if verb then
                printf '\tcascade to default state: ',
                    state_name(xf),'\n';
              end;
            end;
          end; $ for v_state in allTrans
        end; $ for x in curState
        $ process broadcasted events: chain reactions are allowed
        while #theEventQ /= 0 do
          take anEvent fromb theEventQ;
          perform(anEvent, verb);
        end;
      end; $ end perform


      $----------------------------------simul
      $   simple simulation: create random events
      $   and requests their interpretation
      simul := func(n);
        for i in [1..n] do
          anEvent := arb(events);
          perform(anEvent, theVerb);
        end;
      end; $ end simul
```

the include file of the digital watch specification

```
$
$
$              statechart representation
$              a digital watch according
$          to D. Harel, CACM 31, 5, 1988, pp 514-530
$

$___________ blobs and subsystems
$_______watch
$_______alive
$   power light chime_st c_enab alarm_st main
$   displays up_alarm update_ stopwatch
$_______disp_run
$   disp run
$___________events
$___________transitions
$___________initializations, activation

$_____________________________________________
—

linclude statechart.cntr

$   N.B. as blobs are introduced, arbitrary ids are
$       assigned to them: these ids are atoms
$       created by newat

$ several maps to describe the system decomposition
subsystems := {};
blobs := {};
default := {};   $ default state in a blob
history := {};   $ record the last state the blob was in
transitions := {};      $ the event directed transition map: {[state,
[event, state, ...]], ..}


$ auxiliary map for trace execution
$ give a name to each state suitable for a trace exec.
state_name := state_name?{};
event_name := event_name?{};
$_____________________________________________ $
$                                  $
$___________ blobs and subsystems ___________ $
$                                  $
$_____________________________________________ $

$_____________________________________watch
$ the watch is a blob made of dead and alive
watch := newat;
dead := newat;
alive := newat;
blobs{watch} := {dead,alive};
default(watch) := dead;
state_name(watch) := 'watch';
state_name(dead) := 'dead';
state_name(alive) := 'alive';

$ dead is not decomposed

$_____________________________________________alive
$ alive is partitioned in subsystems:
$   main, power, light, chime_st, alarm_st

[main, power, light, chime_st, alarm_st]
    := [newat,newat,newat,newat,newat];

state_name(main) := 'main';
state_name(power) := 'power';
state_name(light) := 'light';
state_name(chime_st) := 'chime_st';
state_name(alarm_st) := 'alarm_st';


subsystems{alive} := {main, power, light, chime_st, alarm_st};

$ let us first deal with the 'small' subsystems

$_____________power
[weak, ok]
    := [newat,newat];
blobs{power} := {weak, ok};
default(power) := ok;

state_name(weak) := 'weak';
state_name(ok) := 'ok';

$_____________light
[on, off]
    := [newat,newat];
blobs{light} := {on, off};
default(light) := off;

state_name(on) := 'on';
state_name(off) := 'off';

$_____________chime_st
[c_disab, c_enab]
    := [newat,newat];
blobs{chime_st} := {c_disab, c_enab};
default(chime_st) := c_disab;

state_name(c_disab) := 'c_disab';
state_name(c_enab) := 'c_enab';

$_____________c_enab
[c_beep, quiet]
```

```
    := [newat,newat];
blobs{c_enab} := {c_beep, quiet};
$   default(c_enab) := om;

state_name(c_beep) := 'c_beep';
state_name(quiet) := 'quiet';

$_____________alarm_st
[disab, enab]
    := [newat,newat];
blobs{alarm_st} := {disab, enab};
default(alarm_st) := disab;

state_name(enab) := 'enab';
state_name(disab) := 'disab';

$_____________main
[displays, beep]
    := [newat,newat];
blobs{main} := {displays, beep};
$   default(main) := om;
state_name(displays) := 'displays';
state_name(beep) := 'beep';

$_____________displays
[time_, date_, chime_, alarm_, up_alarm, update_, stopwatch]
    := [newat,newat,newat,newat,newat,newat,newat];
blobs{displays} := {time_, date_, chime_,
            alarm_, update_, stopwatch};
default(displays) := time_;
$ as a default: history  is the same as default state
history(displays) := time_;

state_name(time_) := 'time_';
state_name(date_) := 'date_';
state_name(chime_) := 'chime_';
state_name(alarm_) := 'alarm_';
state_name(up_alarm) := 'up_alarm';
state_name(update_) := 'update_';
state_name(stopwatch) := 'stopwatch';

$_____________up_alarm
[min_, t_min_, hour_]
    := [newat,newat,newat];
blobs{up_alarm} := {min_, t_min_, hour_};
$   default(up_alarm) := om;

state_name(min_) := 'min_';
state_name(t_min_) := 't_min_';
state_name(hour_) := 'hour_';


$_____________update_
$ the following states are 'shared': date, min_, t_min_, hour_
$ with other blobs
[ day_, sec_]
    := [ newat, newat];
blobs{update_} := {min_, sec_, t_min_,
            hour_, date_, day_};
$   default(update_) := om;

state_name(sec_) := 'sec_';
state_name(day_) := 'day_';

$_____________stopwatch
[disp_run, zero]
    := [newat,newat];
blobs{stopwatch} := {disp_run, zero};
default(stopwatch) := zero;
$ as a default: history  is the same as default state
history(stopwatch) := zero;

state_name(disp_run) := 'disp_run';
state_name(zero) := 'zero';

$_____________disp_run
$ disp and run are disp_run subsystems
[disp,run]
    := [newat,newat];
subsystems{disp_run} := {disp,run};

state_name(disp) := 'disp';
state_name(run) := 'run';

$_____________disp
[reg, lap]
    := [newat,newat];
blobs{disp} := {reg, lap};
$   default(disp) := om;

state_name(reg) := 'reg';
state_name(lap) := 'lap';

$_____________run
[on_r, off_r]
    := [newat,newat];
blobs{run} := {on_r, off_r};
$   default(run) := om;

state_name(on_r) := 'on_r';
state_name(off_r) := 'off_r';


$_____________events
$ consider events as an enumerated set

$   battery events: in-sert, dy-ing, rm (remove), wk (weakening)
[bt_in, bt_dy, bt_rm, bt_wk]
    := [newat,newat,newat,newat];

$   a,b,c,d: button events : the pressing event
$   b_up = de-pressing (releasing) button b
[a, b, b_up, c, d]
    := [newat,newat,newat,newat,newat];

$   two_min   2 min elapsed time since a button was pressed
$   t_hits_hr   internal time reaches chime-alarm time
$   beep_rt   beep return, i.e. return from beep state
$           (occurs at most 2 min after entering beep state)
$   beep_st   occurs 2 seconds after entering c_beep state
$   clh       clear history
[two_min, t_hits_hr, beep_rt, beep_st]
    := [newat,newat,newat,newat];

$ all the events:
events := {  bt_in, bt_dy, bt_rm, bt_wk,
            a, b, b_up, c, d,
            two_min, t_hits_hr, beep_rt, beep_st
            };

event_name := { [bt_in,'bt_in'], [bt_dy,'bt_dy'],
            [bt_rm,'bt_rm'], [bt_wk,'bt_wk'],
            [a,'a'],[b,'b'], [b_up,'b_up'],
            [c,'c'], [d,'d'],
            [two_min,'two_min'], [t_hits_hr,'t_hits_hr'],
            [beep_rt,'beep_rt'], [beep_st,'beep_st']
            };

$_____________transitions

$ a priori transition is not a smap ....

transitions([dead,bt_in]) := alive;
transitions([alive,bt_rm]) := [dead,clh];$ clh : clear history
transitions([weak,bt_dy]) := dead;
transitions([ok,bt_wk]) := weak;
transitions([off,b]) := on;
transitions([on,b_up]) := off;
transitions([c_disab,d]) := [quiet,'in_state(chime_)'];
transitions([c_enab,d]) := [c_disab,'in_state(chime_)'];
transitions([c_beep,beep_st]) := quiet;
transitions([quiet,t_hits_hr]) := c_beep;
transitions([disab,d]) := [enab,'in_state(alarm_)'];
transitions([enab,d]) := [disab,'in_state(alarm_)'];
transitions([displays,two_min]) :=
            [displays,'not in_state(stopwatch)'];
transitions([displays,t_hits_hr]) := [beep,'in_state(enab)'];
transitions([beep,beep_rt]) := 'history(displays)';
transitions([chime_,a]) := 'history(stopwatch)';
transitions([time_,a]) := alarm_;
transitions([time_,c]) := sec_;
transitions([time_,d]) := date_;

transitions{alarm_} := {[c,min_], [a,chime_]};

transitions([alarm_,c]) := min_;
transitions([alarm_,a]) := chime_;
transitions([up_alarm,b]) := alarm_;
transitions([date_,c]) := [day_,'in_state(update_)'];
transitions([date_,d]) := time_;
transitions([sec_,c]) := min_;
transitions([min_,c]) := t_min_;
transitions([t_min_,c]) := hour_;
```

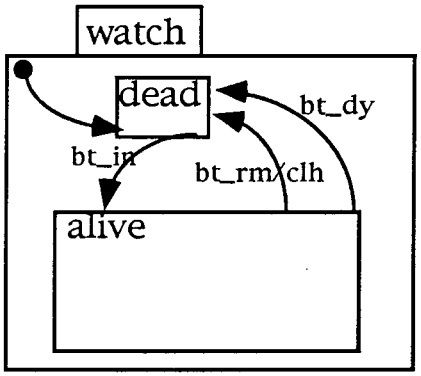
```
transitions{[hour_,c]} := {[date_,'in_state(update_)'],
                [alarm_,'in_state(up_alarm)']};

transitions([day_,c]) := time_;
transitions([on_r,b]) := off_r;
transitions([off_r,b]) := on_r;

transitions{[zero,b]} := {on_r, reg};

transitions{[reg,d]} := {[lap,'in_state(on_r)'],
                [zero,'in_state(off_r)']};

transitions([lap,d]) := reg;
transitions([stopwatch,a]) := time_;
transitions([update_,b]) := time_;
```

```
$_____initializations, activation

$ initialization:

$ start by launching the watch:

$   theFirstevent := bt_in;
$   theSystem := watch;
$   verb      := true;      $ be verbose

simul_init(watch,bt_in,true);

simul(10);
```

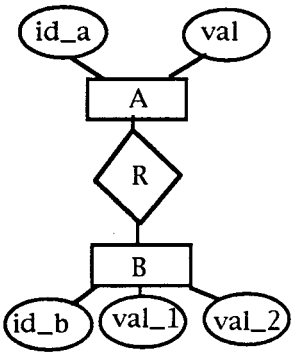
Exercises

- is the process of actually changing the internal time settings described in the fig. 3 chart? What about the alarm time settings?
- one should be able to 'zoom in' and 'zoom-out' in a statecharts. Zooming out of a specific blob means neglecting the internal structure of that blob.

For instance, zooming out of 'alive' the statechart in fig. 3 may be represented by the chart in the opposite column. Zooming in 'alive' in this chart will restitute only one level. Each Zoom operation modifies the reference data structure for the blobs, the subsystems, and the transitions. Define the zoom operations.



- the Entity Relationship Diagrams (ERD) have been described in an exercise concerning data models (see the exercises following **a data table-revisited-2**). These diagrams are used to define actual data organizations. By processing one of these diagrams one should be able to suggest (automatically) possible data organizations.

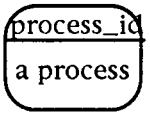


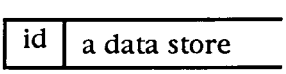
analysing and processing the diagram in the opposite column could yield a data organization using the following generic terms:

A: {..., [id_a, val], ...}
B: {..., [id_b, [val_1,val_2]], ...}
R: {..., [id_a,id_b], ...}

representing by maps the data model

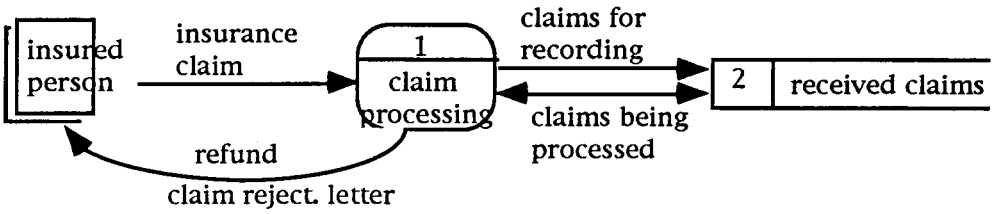
Define a set-oriented representation of ERDs (e.g. by analogy to our blobs, subsystems and transitions maps) which could be used for ERD diagram analysis.
- could one represent ERDs by statecharts? (Hint: forget about the interpretation of statecharts as finite state automata representation)
- the Data Flow Diagrams (DFD) are commonly used in the user requirement definition documents. DFDs are based upon the following iconic representations:





this represents an agent outside the scope of the model, usually an input source or the target of an outgoing information stream	this represents usually a computing or engineering process	This represents a file, a database, or a part of it

Directed arrows link these items. They are labelled with a definition of a data stream. The following is a typical (though extremely simple) DFD:



Define a representation of the DFD, and an interpreter. Usually a DFD presentation of a system is made of several diagrams providing more and more details on the processes. Unlike statecharts these diagrams are not nested: the details of a process with id *i* is given in a diagram labelled *i*, and involves new processes with distinc ids. The uniqueness of the identification is across all diagrams for a given system.

- In a realistic use of diagrams like DFD, labelled arrows and data stores are also represented by detailed diagrams, using a data representation like ERD. Draw a simple set of DFD and ERD diagrams for providing a better understanding of the insurance claim recovery process. Identify the verifications that the computer system processing these diagrams should make to insure their consistency.
- draw a statechart to describe the behaviour of an automated teller machine
- draw a (set of-) statechart(s) describing the workings of a meteorological captor. This automated meteo station is recording temperature every minute, atmospheric pressure every 3 minutes, wind speed every 30 seconds in a database, and unloading via a teletransmission network, every 24 hours, but also upon request all the accumulated data.
- modify the above meteo specification so that the meteo station displays on distinct digital displays the average temperature, atmospheric pressure and wind speed over the last 10 minutes as well as the last recorded values.
- transform the data for the digital watch into n-ary relations, for the relational calculus: in this form are they suitable for interpreting the statechart? provide a ERD representation for the digital watch data, suitable for recording this data in a standard (relational?) database.
- a text processing system is a simple real-time system, consisting of a keyboard, a pointing device (e.g. a mouse) a display window. Represent by a statechart the main functions: adding text from the keyboard, selecting text, copy-pasting, deleting, scrolling.