A Graphic Notation for Formal Specifications of Dynamic Systems *

Gianna Reggio – Mauro Larosa

Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova, Italy
Viale Dodecaneso, 35 – Genova 16146, Italy
{ reggio } @ disi.unige.it
http://www.disi.unige.it

Abstract. Given an already fully developed formal specification method for reactive systems, we also develop an alternative graphic notation for its specifications to improve writing and understanding of such specifications and, hopefully, the acceptance of the method by industrial users.

1 Introduction

A good specification formalism, where "good" means technically well founded, expressive, adequate for some phase of the development process, ..., is not enough for having a formal specification method apt to be used for real applications by non-academic people (see e.g. [5] for a discussion on "formalism" and "method"). In our opinion, the formalism must be equipped with:

1. guidelines on how to proceed to use the formalism; for answering the question "How can I produce a specification ?"
2. friendly presentations of the formal specifications; for answering the question "Can I, and the other interested people (clients, implementors, ...) easily read and understand the formal specifications?"
3. supporting software tools.

Our group at the University of Genova in the last years has developed a logic (algebraic) formalism, LTL, for the specification of (concurrent) reactive systems at different levels of abstraction (see e.g. [4, 8]). In this paper by *reactive systems* we denote systems that are able to modify their own state along time interacting with the external (w.r.t. them) world; the concurrent ones are those whose activity is determined by the activity of other (possibly concurrent) reactive systems. Distributed/concurrent/parallel programs, embedded systems, information systems are examples of concurrent reactive systems.

We have also equipped such formalism with several "(also non-software) tools" to help answering the above questions. Among them:

* This work has been partially supported by the project 40%: "Progetto di una workstation multimediale ad architettura parallela".

– a textual language, METAL ([12]) for writing the specifications with a sufficiently friendly syntax (e.g. where no esoteric symbols are used, but only catchy keywords);
– techniques for structuring the specifications of complex systems;
– guidelines for producing the specifications;
– a way for writing the formal specifications with associated strictly corresponding informal specifications (just natural language text), see [2];
– a software tool for the rapid prototyping of the specified systems, see [1].

The above "tools" answer to point 1, partly to 3 (no tools to support verification) and to 2 by the associated informal natural language specifications.

The resulting specification method has been successfully applied to several case studies (e.g. [3]), also in projects in cooperation with some industry ([15, 16]). For a detailed comparison with other formal methods, see [4].

However, it is not enough to have a formal specification method to be really used and which can be competitive w.r.t. the commonly used and accepted informal methods (as [6, 7]). A relevant deficiency is the lack of a form of graphic notation. The role of graphic is also shown by the success of some formal methods equipped with them (e.g. Petri nets, SDL [10] and State Charts [9]).

For these reasons, we have tried to give a graphic counterpart to the formal specifications developed following the guidelines of our method and written using the specification language METAL.

Any sensible graphic notation has big advantages: readability, understandability, compactness (the corresponding textual counterpart is usually much longer); but it may have also some disadvantages: drawings larger than one page or resembling a maze are not better than text. To avoid the drawbacks of graphic we need ways to split the drawings in sensible pieces and to give them an appropriate lay-out. In our case, the guidelines for writing textual specifications can be extended to help producing good drawings, and since the specifications are already structured, we have a natural way to split the drawings. For what concerns this point, we have used also some techniques inspired by other graphic methods:

– the drawing associated with a specification consists of several overlapping transparent layers (*views*) determined by the activities required by the original guidelines; and so they are extremely natural and support the application of the guidelines to the development of the specifications. Depending on the complexity we can put in one drawing one or more views. This technique has been inspired by OOA-OOD [6, 7].
– each part of a specification is characterized by an icon, and such icon together the part name (*place holder*) can be used to refer to it inside a drawing, while its full drawing is given apart (as it is done in SDL).
– for the parts that usually are too large, we have developed special ways to split them into page-long pieces, see [17].

Notice that our proposal differs from other approaches having the same aims and present in the literature, which have been designed by putting together an existing informal graphic method with a formal notation (see e.g. [14]); here the graphic part is not given by an existing method but it has been derived from the formal specifications.

In our opinion this attempt seems to be promising. It has worked also for us, strikingly making easier to produce good specifications and reducing their size.

We still have a fundamental task to do: develop software tools for interactively producing the drawings (which cannot be sensibly made by hand), browsing and converting them into the corresponding textual versions (needed e.g. to use the already existing tools) and vice versa.

Our formalism, LTL, allows to write specifications adequate for the different phases of the development process, from the initial requirements till to the detailed design of the developed system. Here, for lack of room we consider only the specifications for what we call *design phases*, i.e. when the structure and the activity of the developed system are fully determined. Notice that "fully determined" must be intended at the abstraction level supported by the formalism, and not at the level of coding in some programming language.

The paper is organized as follows. The formalism LTL is shortly introduced in Sect. 2. Then, we present the guidelines for writing a specification of a system, briefly reporting the textual version and the corresponding graphic one. Reactive systems may be classified as *simple* and *concurrent*, where the latter are those whose activity is determined by the activity of more than one components, which are in turn reactive systems; as the guidelines are different for simple and concurrent systems we consider the first and the latter respectively in Sect. 4 and in Sect. 3. A complete presentation of the method is in [17]. Finally, in Sect. 5 we try to draw some conclusions and point to what should be the next tasks to get a method that may have chances to be really used.

In the paper, we will use, as running toy example, the specification of a very simple pocket calculator, described as follows; for more interesting examples see, e.g., [16, 15].

The calculator consists of a keyboard *for inputting digits and commands, a* display *for showing the pressed keys and the results, an* echo *that echoes on the display the keys pressed on the keyboard and a* computing unit*, which reads the digits and the commands from the keyboard, performs the appropriate calculations and writes the results on the display. A user interacts with the calculator by pressing the keys on the keyboard and by reading the display.*

2 The Formalism: Labelled Transition Logic (LTL)

To model reactive systems LTL uses labelled transition systems (see [11]).

A *labelled transition system* (shortly *lts*) is a triple $(S, L, \rightarrow)$, where S and L are two sets, the *states* and the *labels* of the system, and $\rightarrow\ \subseteq S \times L \times S$ is the *transition relation*. A triple $(s, l, s') \in \rightarrow$ is said a *transition* and is usually written $s \xrightarrow{l} s'$.

A reactive system R is thus modelled by an lts $LTS = (S, L, \rightarrow)$ and an initial state $s_0 \in S$; the states reachable from s_0 represent the intermediate (interesting) situations of the life of R and the arcs between them the possibilities of R of passing from a state to another one. It is important to note that here an arc (a transition) $s \xrightarrow{l} s'$ has the following meaning: R in the state s has the *capability* of passing into the state s' by performing a transition, where label l represents the interaction with the external (to R) world during such move; thus l contains information on the conditions on the external world for the capability to become effective, and on the transformation of such world induced by the execution of the action; so transitions correspond to *action capabilities*.

Concurrent systems, i.e. having cooperating components that are in turn other reactive systems, can be modelled through particular lts's obtained by composing other lts's describing such components.

An lts can be represented by a first-order structure L on a signature with at least two sorts, *state* and *label*, whose elements correspond to the states and the labels of the system respectively, and a predicate $\rightarrow$: *state label state* representing the transition relation. The triple $(\mathrm{L}_{state}, \mathrm{L}_{label}, \rightarrow^{\mathrm{L}})$, whose components are respectively the interpretation of *state*, *label* and of $\rightarrow$, is the corresponding lts.

Obviously we can have lts's whose states are built by states of other lts's (for modelling concurrent systems); in such a case we use first-order structures with different sorts corresponding to states and labels and with different predicates corresponding to transition relations. In a formal model for reactive systems we need to consider data too (e.g. the data used by a process); to handle these cases, we consider structures possibly having also sorts that just correspond to data and neither to states nor labels of lts's.

The first-order structures corresponding to lts's are called *LT-structures* and are formally defined as follows.

- An *LT-signature* $LT\Sigma$ is a pair (Σ, DS), where:
  * $\Sigma = (S, OP, PR)$ is a (first-order) signature,
  * $DS \subseteq S$ (the elements in DS are the *dynamic sorts*, i.e. the sorts corresponding to states of lts's),
  * for all $ds \in DS$ there exist a sort $l\_ds \in S - DS$ (labels of the transitions of the dynamic elements of sort ds) and a predicate $\_ \xrightarrow{\_} \_ : ds\ l\_ds\ ds \in PR$ (transition relation of the dynamic elements of sort ds).
- An *LT-structure* on $LT\Sigma$ (shortly $LT\Sigma$-structure) is a Σ-structure.

In this paper, for some of the operation and predicate symbols, we use a mixfix notation; for instance, $\_ \xrightarrow{\_} \_ : ds\ l\_ds\ ds$ means that we shall write $t \xrightarrow{t'} t''$ instead of $\longrightarrow(t, t', t'')$; i.e. terms of appropriate sorts replace underscores.

An *LT-specification* is a pair $\mathrm{SP} = (LT\Sigma, AX)$, where $LT\Sigma = (\Sigma, DS)$ is an LT-signature and AX a set of positive conditional formulae on Σ (i.e. of the form $\wedge_{i=1,\dots,n}\ \alpha_i \ \supset\ \alpha_{n+1}$, where for $i = 1, \dots, n+1$, α_i is an *atom*, i.e. a formula having form either $t = t'$ or $Pr(t_1, \dots, t_n)$; t, t', t_1, $\dots$, t_n are terms and $Pr \in PR$).

Notice, that since transitions are described by predicate symbols the axioms of an LT-specification allow us also to express properties on the activity of the systems.

SP determines one (up to isomorphism) LT-structure, precisely $\mathrm{I}_{\mathrm{SP}} = \mathrm{T}_\Sigma/\equiv_{AX}$, where T_Σ is the Σ-structure of the ground terms and $\equiv_{AX}$ is the congruence on T_Σ generated by AX, which can be characterized by a Birkhoff-like deductive system.

I_{SP} is called the initial model of SP, since it is initial in $\{\mathrm{L} \mid \mathrm{L} \models F, F \in AX\}$, i.e. in the class of the models of SP.

It is important to note that in I_{SP}: – each element is the interpretation of a ground term, i.e. it can be represented by a ground term; – two terms are identified iff their equality is consequence of the axioms (can be proved by the deductive system); – a predicate applied to some elements is true iff the truth of the corresponding ground atom is consequence of the axioms (can be proved by the deductive system). As a consequence, since the transitions correspond to the truth of the transition predicates, we have that the transitions in I_{SP} are just those whose corresponding atoms are consequences of the axioms (can be proved by the deductive system).

3 Specifying Simple Systems

In our method specifying a simple system means to abstractly determine an $LT\Sigma$-structure, where $LT\Sigma$ has one dynamic sort (i.e. an lts plus some data structures), by an LT-specification. This means to determine which are the states, the labels, the transitions of the lts, possibly some auxiliary data structures, needed to define such states/labels, and to abstractly specify them. Thus, we decompose the activity of specifying a simple system in four phases, presented in the next subsections; each phase produces a different view of the graphic specification; by overlaying all the views we get the complete graphic presentation.

In this section, as a current example of simple system we consider the computing unit component of the calculator.

The computing unit executes the various commands as follows. When the computing unit is on, it has always stored a partial current result and the command to apply next. The computing unit may read and reset the keyboard anytime a command is pressed; then it proceeds as follows: – by parsing the keyboard content it gets the second operand and the next command to apply; – then it evaluates the old stored command on the partial result and on the new list of digits, obtaining a new partial result; – afterwards it sends such result to the display while simultaneously it also reactivates the echo, returning then to wait for the pressing of a command. If the second operand is the empty list, then the evaluation returns the first operand, i.e. the old partial result; if both the operands are not empty and the command is the equal, then the evaluation returns the second operand, otherwise it applies the command to the two numbers represented by the two lists.

3.1 Basic data structures

In this phase we determine and specify the data structures used by the system. For example, the basic data structures of the computing unit are naturals, lists of digits and lists of keys.

A specification of a data structure is just an LT-specification without dynamic sorts (an algebraic specification of an abstract data type, see [18]); thus sorts, operations and predicates of the signature correspond respectively to the types of values, the operations and the boolean tests of the data structure, and the axioms to their properties. The initial semantics will determine the structure where all values are represented by a ground term and all and only the logical consequences of the axioms hold.

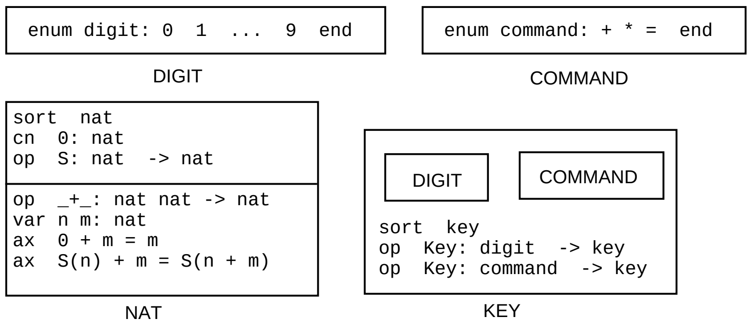
We assume that each data structure has a *main sort*, named as the structure itself.

The icon for a data structure is a rectangle. A graphic specification of a data structure is simply a rectangle including the specification text with below the name of the specification.

Whenever a specification `SP` uses another specification `SP'` (thus `SP` will have also the sorts, the operations, the predicates and the axioms of `SP'`), a rectangle enclosing the used specification name `SP'` is included in the icon of `SP`.

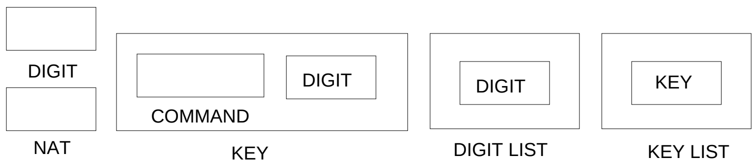
The specification text is split by horizontal lines putting together operations/ predicates with their axioms.

Below we report part of the data structures of the computing unit.



`enum srt: Op1 ... Opn end` is a METAL shortcut for the specification of a data structure corresponding to an enumeration type, having main sort **`srt`**, constant operations `Op1`, ..., `Opn` and no identifications on them.

Another view of the basic data structures is the so called *hierarchical schema* representing the mutual relationships among them; it consists of the icons (rectangles) of the various structures and of their names, while the text parts are dropped. Furthermore, this view may be made more compact, by putting the drawing of a structure, used only by another structure, inside the latter. Below we give the whole hierarchical schema for the basic data structures of the computing unit.



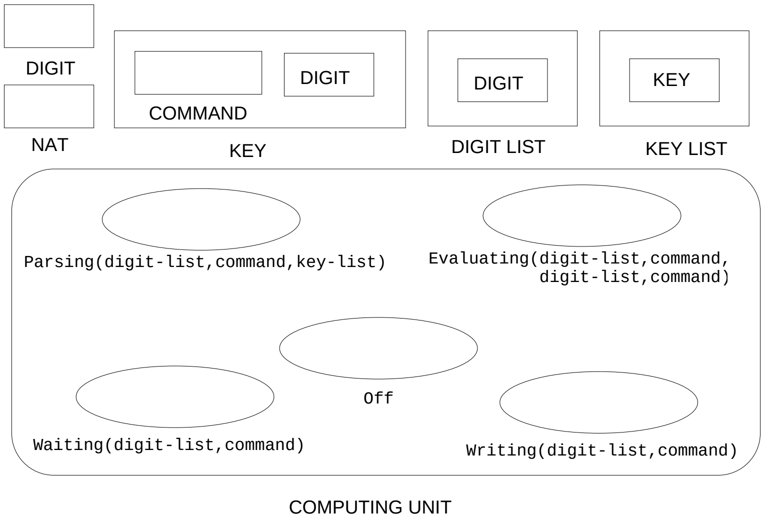
3.2 States

In this phase we determine and specify the (kinds of the) intermediate relevant situations in the life of the system, i.e. the states of the lts modelling it. Each state kind should be conceptually characterized by a natural language sentence corresponding either to an adjective or to a verb in the ing form.

The states are then specified as a data structure, possibly using the basic data structures, with a main sort corresponding to the states themselves, named as the system, and an operation for each state kind. Then the main sort is made dynamic. In this case the specification has no axioms, since we do not want to identify the terms representing the various states.

Each state kind is represented by an oval with below the corresponding operation applied to the sorts of its arguments (we do not need to report the result sort, since it coincides with the system name). The representations of the state kinds are enclosed in the system icon (a rectangle with rounded angles) and the hierarchical view of the basic data structures is reported over the system icon, to have at hand a synthetic view of such structures.

Below we give the specification of the states of the computing unit.

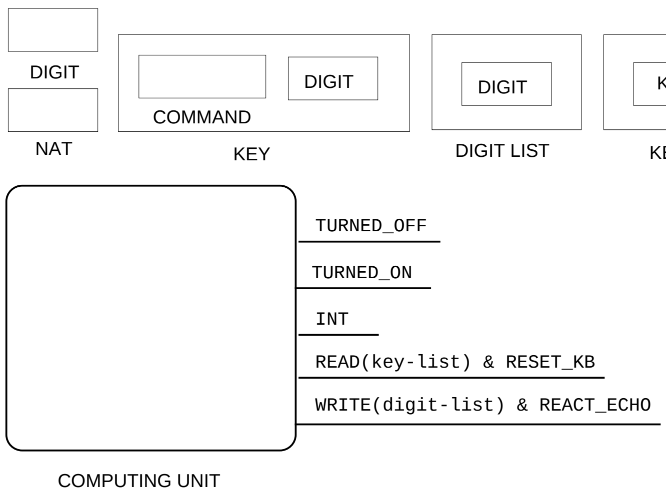


For example, the states of kind `Parsing` are characterized by three values of sort respectively `digit-list`, `command` and `key-list` specified by the data structures with the same names.

3.3 Interactions

In this phase we determine and specify the (kinds of the) interactions of the reactive system with the external world, i.e. the labels of the lts modelling it. Each interaction kind should be conceptually characterized by a natural language sentence corresponding to a verb having either active form ("to verb complements") or passive form ("to be verb-ed complements"). The interactions are then specified as a data structure using the basic data structures, with the sort of labels as main sort, and having an operation for each interaction kind; these operations may have several arguments describing the information exchanged between the system and the external world during each interaction of such kind (informally described by the complements).

The icon for the interaction kinds is a line attached to the system icon, to emphasize the idea that interactions are the ways for connecting the system to the external world. Each line is decorated with the corresponding operation, as for the states; moreover also in this case we report the hierarchical schema of the basic data structures.



3.4 Activity

In this phase we determine and specify the activity of the system by giving its action capabilities, i.e. the transitions of the lts modelling it. Formally, they are represented by the arrow predicate.

The action capabilities/transitions starting from the states of a given kind are defined by a set of conditional axioms of the form:

(*1) $$cond(t_1, \ldots, t_n, t''_1, \ldots, t''_k, t'_1, \ldots, t'_m) \supset$$
$$S\_Kind(t_1, \ldots, t_n) \xrightarrow{I\_Kind(t''_1, \ldots, t''_k)} S\_Kind'(t'_1, \ldots, t'_m)$$

S_Kind, S_Kind' are state kind operations; I_Kind is an interaction kind operation, t_1, …, t_n, lab, t'_1, …, t'_m are terms of the appropriate sort; and $cond(t_1, \ldots, t_n, t''_1, \ldots, t''_k, t'_1, \ldots, t'_m)$ is a conjunction of atoms in which the subterms and the variables of t_1, …, t_n, t''_1,…, t''_k, t'_1, …, t'_m may appear.

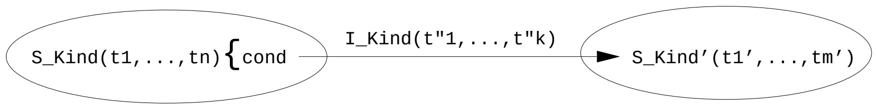
We have to give such axioms by recalling that they hold for all possible instantiations of their variables; thus one axiom determines a possibly infinite set of transitions, all those obtained by the variable instantiations of the arrow part for which the condition part holds.

We proceed by first determining the initial states and the transitions starting from such states; then we repeat such procedure for all state kinds reachable from that one, untill all state kinds have been considered.

Each axiom is accompanied by a natural language statement obtained by the natural language explanations of the state and interaction kinds given previously.

It may happen that to determine the action capabilities we have also to consider relevant "static (sequential) aspects" of the system. For example, for the computing unit, we have to define the parsing of key strings and the application of a command to two lists of digits. To accommodate them in the specification, we introduce some "auxiliary" operations/predicates defined by conditional axioms, as those of the data structures, possibly using other auxiliary operations.

The icon for a transition (an action capability) is very naturally an arrow; but decorated by terms with variables, instead of by values as in the various methods based on automata and finite state machines. Furthermore, the states must be represented by using their icon (an oval). Thus, a conditional axiom of the form as **(*1)**, see above, will be represented by



Now, we are ready to represent the whole activity of the system. The chosen icon is a (connected) labelled direct graph. The simplest idea, to get such representation, is to connect the arrows representing the axioms by gluing together states decorated in the same way; unfortunately the result is usually a non-connected graph. So, we group together all nodes referring to states of the same kind, and get what we call a *macro-state*. We represent a macro-state by the same icon of states, the oval, and put the decorations of all particular cases inside such oval.

In Fig. 1 we give the complete graph of the computing unit activity.

To get the full graphic representation of the activity we still need to consider the declaration of the used variables and the auxiliary operations. A variable `v` of sort `str`, where `str` is the main sort of a data structure `STR`, is graphically represented by attaching `v:` to the icon of `STR` in the hierarchical schema of the basic data structures of the system (which is reported also over this view of the system).

The auxiliary operations, if any, are represented as those of the data structures, but in this case we do not need to represent the used data structures, since they can be only the basic data structures. The full representation of the activity of the computing unit is in Appendix A.

4 Specifying Concurrent Systems

Concurrent systems are particular reactive systems having cooperating components (reactive systems) either active or passive. The passive components are

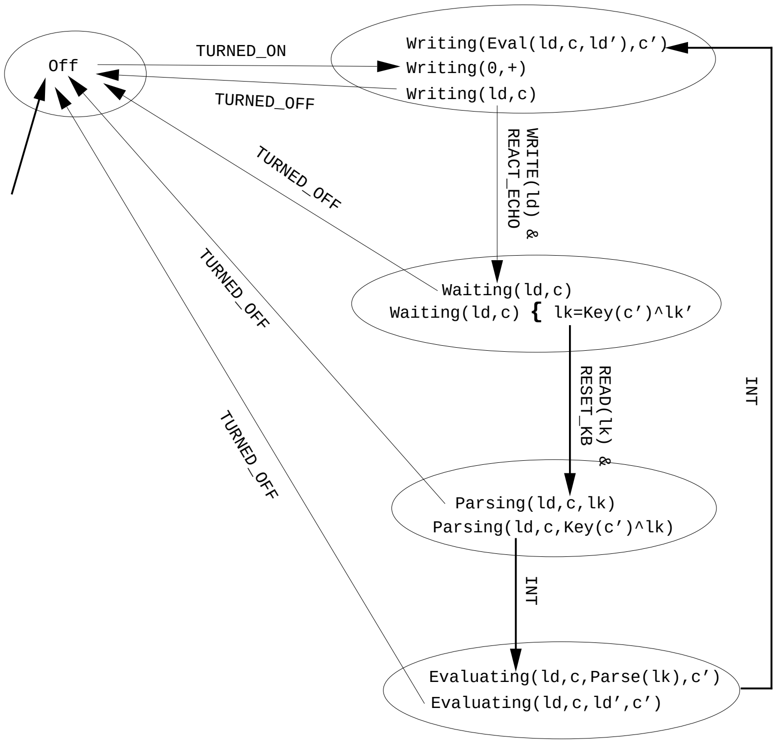


Fig. 1. The graph of the activity of the computing unit

those which can modify their state only because of the activity of some other (active) components. We do not need to describe the interactions and the activity of the passive components, since their states can be changed in any possible way; thus the passive ones are just data structures, while the active components are in turn reactive systems (simple or concurrent).

The calculator is a concurrent system having four components: two active, the computing unit and the echo, and two passive, the keyboard and the display.

The concurrent systems can be classified by considering whether the number of the component is *fixed* or not (all possible states of the system have/have not the same number of components). The guidelines slightly depend on this classification, and here, for lack of room, we consider only the case of systems with fixed number of components; see [17] for the treatment of the other case.

Also the activity of specifying a concurrent system with our method is decomposed in several phases producing the various views; here they are five, instead of the four for the simple systems, because now we also consider the components.

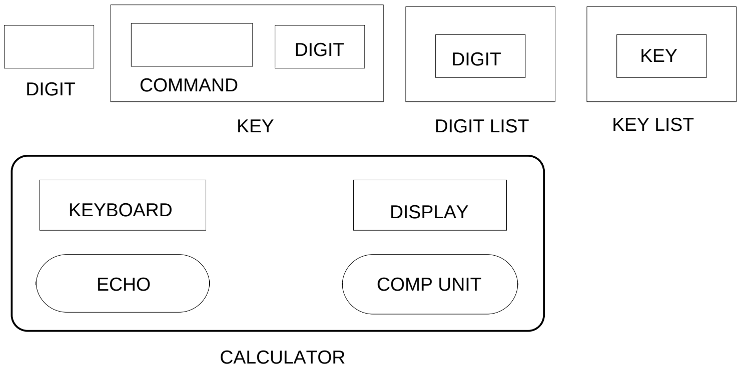
The phases concerning the basic data structures and the interactions are as for the simple systems.

4.1 Components

In this phase we determine and specify the components of the concurrent system. The active ones are reactive systems and we consider them as stand alone systems. The passive components are data structures with a main sort describing the component states. The fact that a system is concurrent and which are its components is graphically as follows.

The icons for active and passive components (reactive systems and data structures) are, as before, the rectangles with rounded angles and the rectangles. We graphically represent the fact that a system is concurrent and which are its components as follows. As usual the hierarchical schema of the basic data structures is reported over.

The structure of the calculator is presented as follows. The computing unit has been specified in the previous section, while the specifications of the other components of the calculator are reported in Appendix A.



4.2 States

Also in this case the states represent the intermediate relevant situations in the life of the system, but now they are just given by putting together the states of the components. For the case of the concurrent systems with fixed components, the states are just sequences of component states. Thus, there is no need of a graphic representation of the states, since they are fully determined by giving the components in the structure representation.

4.3 Activity

The action capabilities of a concurrent system in a state are intuitively described by saying how the components in such state cooperate: precisely, by saying which groups of action capabilities of the active components in such state depending on the states of the passive ones can be performed together, and how the passive ones are consequently modified.

Formally, the system transitions are defined by a set of conditional axioms of the form below, one for each cooperation kind, where we assume without loss of generality that all active components are on the left, and those which act are on the extreme left:

(*2) $a_1 \xrightarrow{lab_1} a'_1 \wedge \ldots \wedge a_k \xrightarrow{lab_k} a'_k \wedge cond(l, lab_1, \ldots, lab_k, p_1, \ldots, p_h) \supset$
$$a_1|\ldots|a_k|\ldots|a_n|p_1|\ldots|p_g|\ldots|p_h|\ldots|p_m|\ldots|p_r \xrightarrow{lab} a'_1|\ldots|a'_k|\ldots|a_n|p'_1|\ldots|p'_g|\ldots|p_h|\ldots|p'_m|\ldots|p_r,$$
where lab_1, ..., lab_k, lab are terms; a_1, ..., a_n, a'_1, ..., a'_k are variables of sort states of the active components; p_1, ..., p_r, are variables of sort states of the passive components; p'_j are terms of the same sorts.

a_1, ..., a_k, p_1, ..., p_m are the components cooperating as described by this axiom (p_1, ..., p_g are read and modified, p_{g+1}, ..., p_h are only read and p_{h+1}, ..., p_m are only modified); a_{k+1}, ..., a_n, p_{m+1}, ...p_r are the components which do not cooperate.

The above constraints on the form of the axioms ensure that only the interactions of the active component (the labels of the transitions) and the states of the passive ones are relevant for the composition (while the states of the active components cannot be considered), and so that interactions really represent the interfaces of a reactive system.

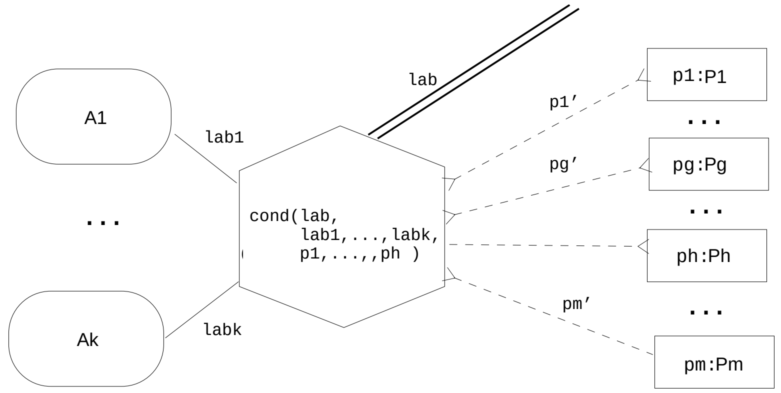
Also in this case it is possible to use auxiliary operations to specify the static/ sequential aspects.

The activity of a concurrent systems cannot be represented by a labelled graph. In the case of the simple system using the macro-states we are able to keep the graph small enough; while if we proceed in this way for a concurrent system we usually get too many nodes. Also for the small example of the calculator the activity graph is too big to be presented in a sensible way. Furthermore, the specification method is conceptually based on the idea of describing how the components cooperate and not on explicitly giving the action capabilities of the whole system. For these reasons we have tried to represent graphically the cooperation, instead of the system action capabilities.

The icon for cooperation is the *hexagon*. The icons for the components have been defined before; those of the passive components are decorated with a variable denoting their generic state.

A line decorated by a term of label sort connecting an active component icon to the hexagon represents that such component cooperates with such interaction. A dashed line from a passive component icon to the hexagon represents that a passive component is involved in the cooperation. Such lines have an arrow from the passive component when the component is read and an arrow towards the hexagon when the component is modified, in such cases the term representing the new component state is put over the line. The interaction with the external world (w.r.t. the concurrent system) is represented by a double line leaving the hexagon decorated with the term representing such interaction.

Thus, a generic axiom describing a cooperation kind of the form **(*2)**, see above, is graphically represented by:



The cooperation of the calculator is represented in Fig. 2, where the double lines labelled by `INT`, corresponding to calculator internal activity, have been dropped.

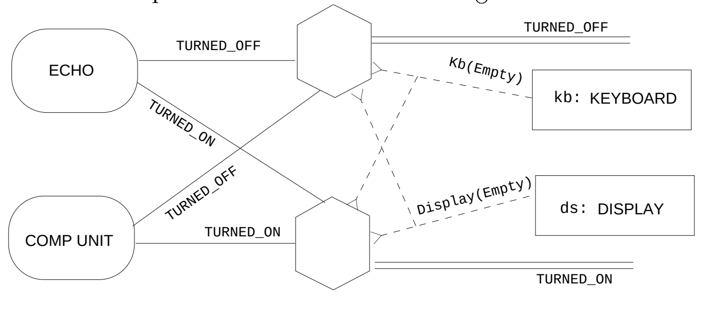
How to decompose the cooperation diagram Unfortunately, in realistic applications the cooperation drawing may be too large or too complex (too many intersecting lines). We now present a modular way to decompose it.

First we put together the cooperation kinds sharing common features (e.g. those about logically related features, those involving the same components, those having the same interaction with the external world, ...). Each of such parts is called *macro-cooperation kind*.

In the case of the calculator a choice of macro-cooperation kinds may be: those resulting in internal activity, those related to the turning on and off and those related to a user working with the calculator.

The elements of each macro-cooperation kind will be put together in one separate drawing; while in the drawing of the activity of the system they will be replaced by an icon. The icon for macro-cooperation kind is a double hexagon with inside its name. If the cooperation kinds grouped in a macro are C1, ..., Cn, then there is a dashed line [line] connecting the double hexagon with the icons of the passive [active] components taking part in either C1 or ...or Cn [decorated by the set of the interaction kinds appearing in either C1 or ...or Cn], and a double line decorated by the set of the interaction kinds with the external world produced by either C1 or ...or Cn.

The two cooperation kinds about turning on-off of the calculator:



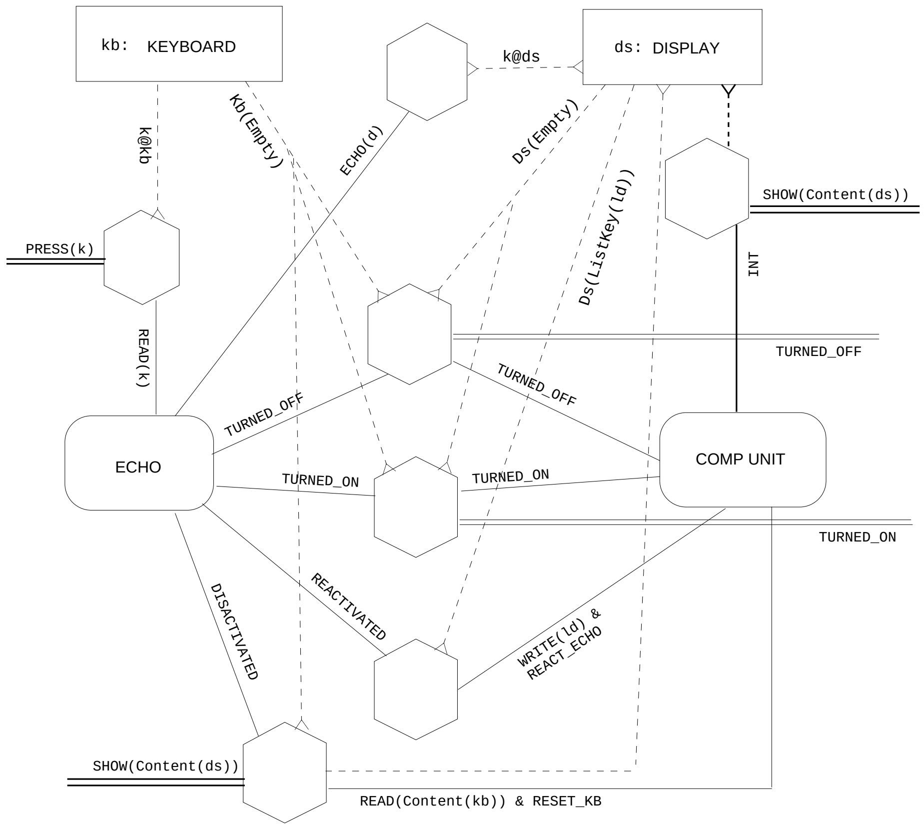
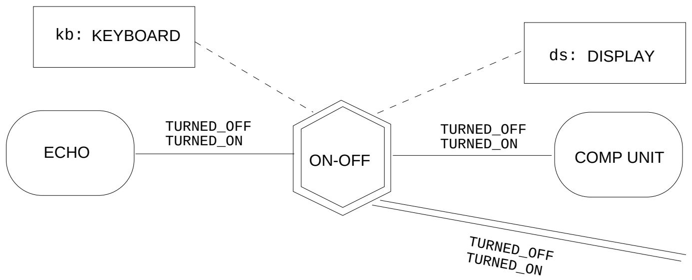


Fig. 2. The activity of the calculator

becomes a macro-cooperation kind graphically represented by



In Appendix A we give the calculator activity in a modular way by using two other macro-cooperation kinds.

5 Conclusions and future work

In this paper we have presented an attempt at making an already developed formal method for the specification of concurrent reactive systems better suited to be used in practice by giving a graphic presentation to its formal specifications.

We have shown how to produce drawings, instead of textual specifications, following our method; thus we have established also a correspondence between graphic and textual specifications, and so we are able to pass from one to another; e.g., see in [17] the corresponding textual specifications for the calculator case.

Some advantages of using the graphic version of the method are as follows.

We have "psychological advantages" when presenting the method; instead of scaring possible users by pages of formulae, we show "appealing" drawings.

The graphic notation supports and makes more explicit the underlying concepts of the method and enforces the application of the guidelines, e.g.:

* we are giving the capabilities of the system of passing from a state to another, when drawing arrows;
* we are giving the interactions of the system with the outside world, when drawing lines (electric wires) attached to the system icon;
* all action capabilities from a state are described together.

We get much more compact presentations of the specifications; instead of three pages of formulae we may have just one page long drawing. That helps also to grasp global aspects of the system with just a glimpse (e.g. whether a state is terminal for a system, whether there is a connection between two components), instead of inspecting a list of formulae.

Clearly, to get these advantages we should give an appropriate lay-out to the drawings, and so the guidelines have to be extended to avoid to convey wrong information by a bad laid-out drawing (see [13]). For example, see in Fig. 3 an alternative wrong lay-out for the activity graph of the computing unit, given before in Fig. 1; now the drawing has a nice symmetry, but now the fact that the computing unit proceeds in a cyclic way is not so easy to grasp.

We have the possibility to discuss/simulate/prototype by hand the behaviour of the specified system just by looking at the drawings and that can be very useful in the validation activity and when discussing the design of the system with clients.

As a short trivial example taken from the calculator, assume that we want to know whether it may be turned off in any moment (but this kind of analysis can be done for interesting point). We look at the calculator activity drawing. There is only one cooperation kind resulting in the interaction `TURNED_OFF`; it requires that both the computing unit and the echo are turned off and there is no condition on the passive components. The computing unit and the echo can be turned off except when already off. Thus the calculator can be turned off in any case except when already off. If we want that the calculator can be always turned off, we see that it is sufficient to add the turning off action capabilities to the computing unit and to the echo when they are off.

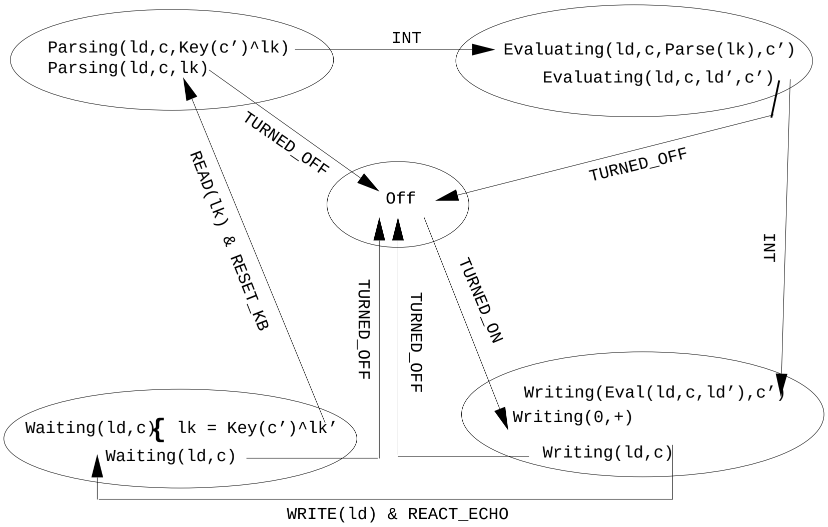


Fig. 3. A bad lay-out for the computing unit graph

In our opinion this attempt seems to be promising; but we still have a fundamental step to do: to develop some software tools for supporting its use. Precisely, we need:

- A graphic editor for producing directly on the screen the drawings.
- A converter from the graphic to the textual presentation and vice versa; to be able, e.g., to use existing tools developed for the textual specifications.
- A browser for examining the specifications able to switch from a view to another and to open the specification of a subpart when clicking its place holder icon inside the representation of some other one.

The above set of tools seem feasible; indeed there are already existing similar tools for other graphic formalisms; we just need some resources.

While the dynamic aspects of the specifications are greatly highlighted by the graphic version, the presented notation offers only a small improvement for the specifications of the data structures: we just make explicit the hierarchical structure and spatially organize the text, grouping related parts. Regarding this point, there are no relevant proposals in the literature; only recently some work on how to graphically represent the terms in the axioms has appeared, but nothing for the resulting data structure. This topic merits to be further studied.

The graphic specifications could be intuitively considered as another concrete syntax for our textual specification language; but here they are not presented as precisely as by means of a BNF grammar. Thus, another future task is to try to use techniques developed in the field of visual languages, to describe the drawings and the correct lay-outs in a more exact way, and to make the correspondence between textual and graphic versions more precise.

Acknowledgments. The present work is strictly connected with other researches carried out in Genova. The authors thank Egidio Astesiano for many fruitful discussions and the referees for helpful comments.

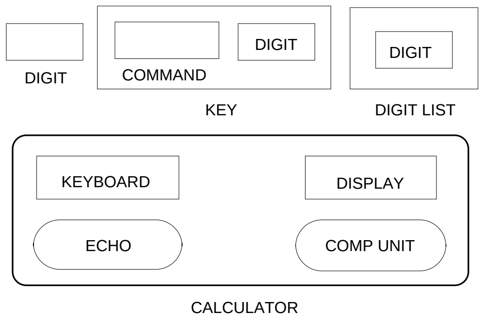
References

1. E. Astesiano, F. Morando, and G. Reggio. The SMoLCS Toolset. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *Proc. of TAPSOFT '95*, number 915 in L.N.C.S., pages 810–811. Springer Verlag, Berlin, 1995.
2. E. Astesiano and G. Reggio. Formally-Driven Friendly Specifications of Concurrent Systems: A Two-Rail Approach. Technical Report DISI–TR–94–20, DISI – Università di Genova, Italy, 1994. Presented at ICSE'17-Workshop on Formal Methods, Seattle April 1995.
3. E. Astesiano and G. Reggio. A Dynamic Specification of the RPC-Memory Problem. In M. Broy, S. Merz, and K. Spies, editors, *Formal System Specification: The RPC-Memory Specification Case Study*, number 1169 in L.N.C.S., pages 67–108. Springer Verlag, Berlin, 1996.
4. E. Astesiano and G. Reggio. Labelled Transition Logic: An Outline. Technical Report DISI–TR–96–20, DISI – Università di Genova, Italy, 1996.
5. E. Astesiano and G. Reggio. Formalism and Method. In M. Bidoit and M. Dauchet, editors, *Proc. TAPSOFT '97*, number 1214 in L.N.C.S., pages 93–114, Berlin, 1997. Springer Verlag.
6. P. Coad and E. Yourdon. *Object-Oriented Analysis*. Prentice-Hall, Englewood Cliffs, N.J., 1991.
7. P. Coad and E. Yourdon. *Object-Oriented Design*. Prentice-Hall, Englewood Cliffs, N.J., 1991.
8. G. Costa and G. Reggio. Specification of Abstract Dynamic Data Types: A Temporal Logic Approach. *T.C.S.*, 173(2):513–554, 1997.
9. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.
10. ITU. Z.100 ITU Specification and Description Language (SDL). Technical report, ITU, Geneva, 1993.
11. R. Milner. *A Calculus of Communicating Systems*. Number 92 in L.N.C.S. Springer Verlag, Berlin, 1980.
12. F. Parodi and G. Reggio. METAL: a Metalanguage for SMoLCS. Technical Report DISI–TR–94–13, DISI – Università di Genova, Italy, 1994.
13. M. Petre. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Communications of the ACM*, 38(6), 1995.
14. F. Polak, M. Whiston, and K. Mander. The SAZ Project: Integrating SSADM and Z. In J.C.P. Woodcock and P.G. Larsen, editors, *Proc. FME'93: Industrial-Strength Formal Methods*, number 670 in L.N.C.S. Springer Verlag, Berlin, 1993.
15. G. Reggio. A Graphic Specification of a High-Voltage Substation. Technical Report DISI-TR-96-22, DISI – Università di Genova, Italy, 1996.
16. G. Reggio. A Graphic Specification of a Hydroelectric Power Station. Technical Report DISI–TR–96–21, DISI – Università di Genova, Italy, 1996.

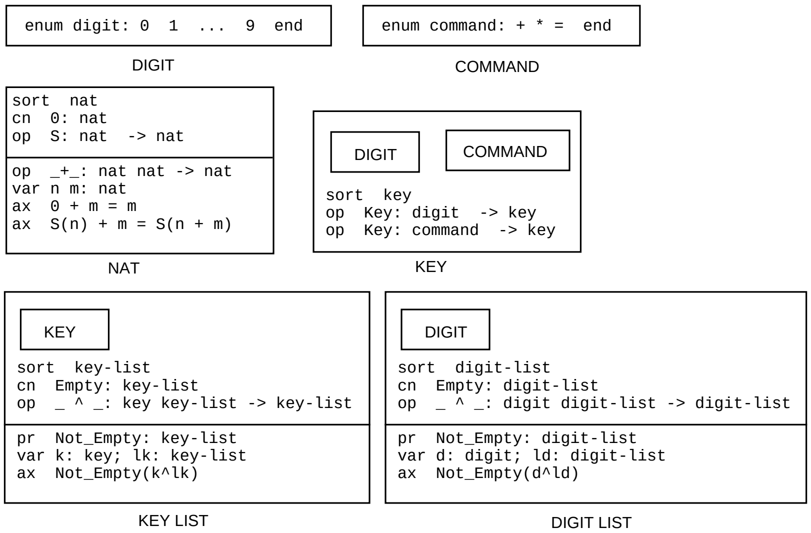
17. G. Reggio and M. Larosa. A Graphic Notation for Formal Specifications of Dynamic Systems. Technical Report DISI–TR–97–3, DISI – Università di Genova, Italy, 1997. Full Version.
18. M. Wirsing. Algebraic Specifications. In J. van Leeuwen, editor, *Handbook of Theoret. Comput. Sci.*, volume B, pages 675–788. Elsevier, 1990.

A The Complete Graphic Specification of the Calculator

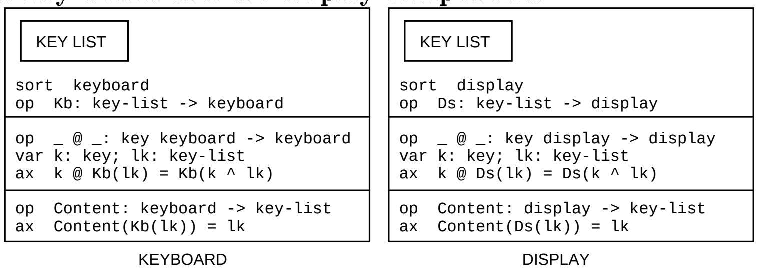
The Structure of the calculator



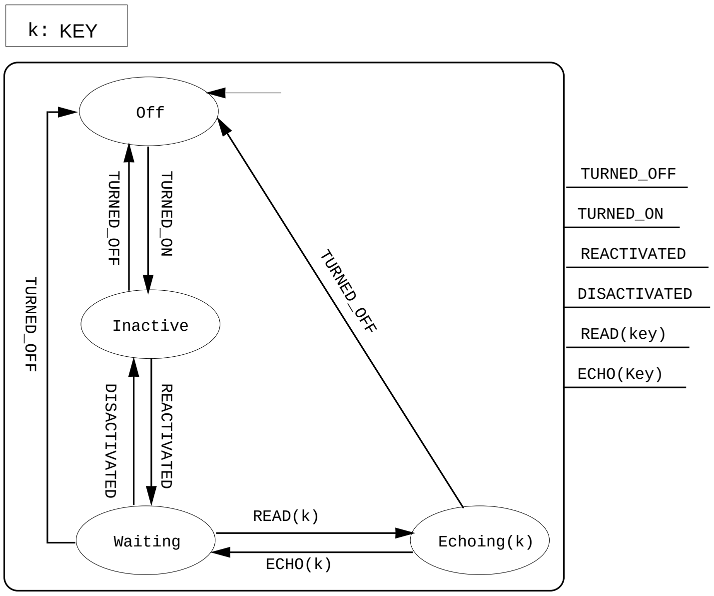
The basic data structures of the calculator



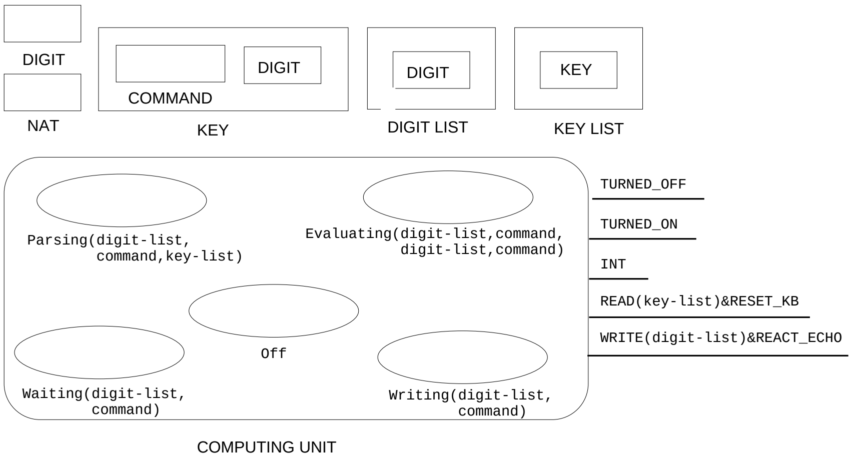
The key-board and the display components



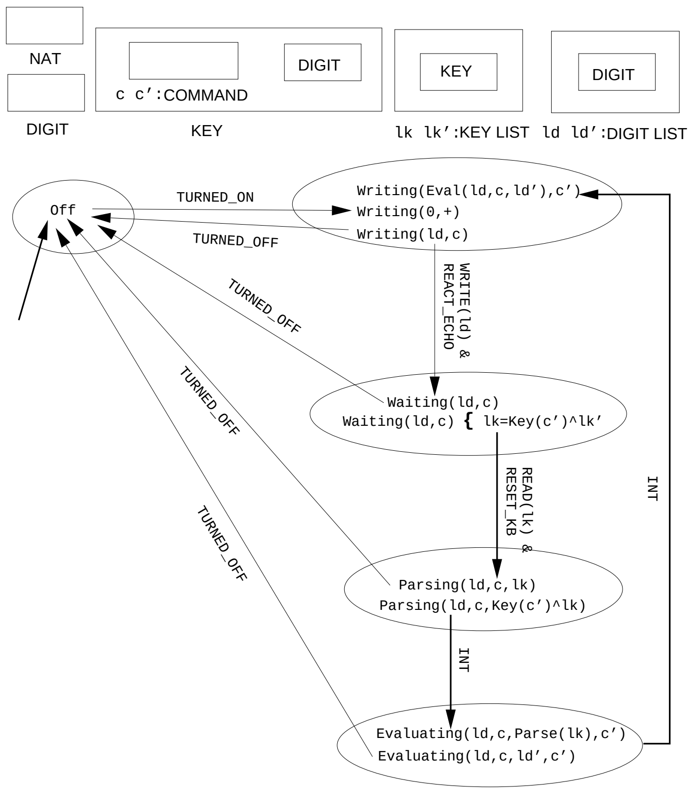
The echo



The computing unit states and interactions

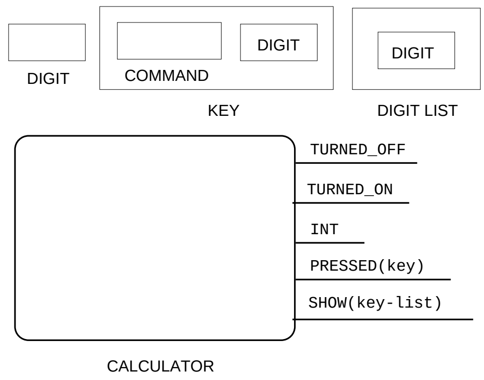


The computing unit activity

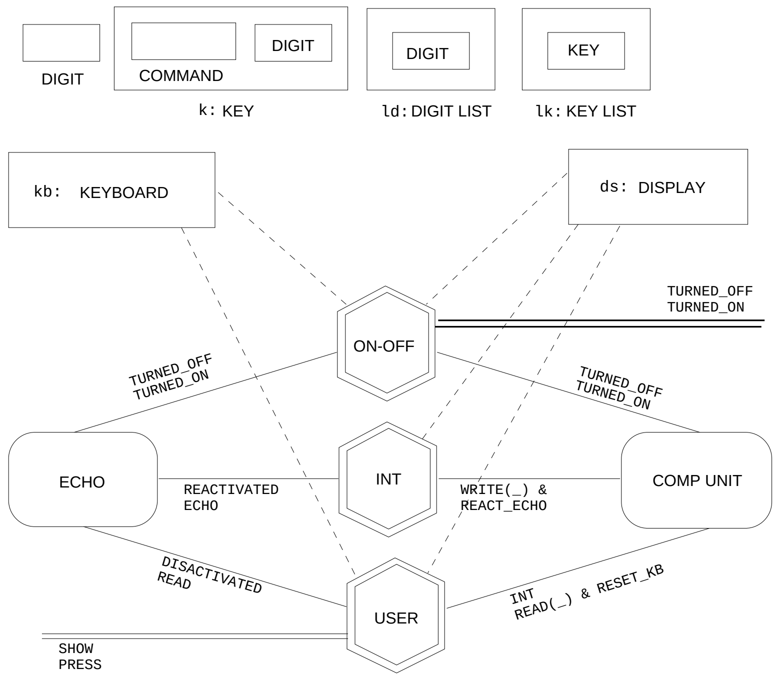



```
**  given a list of keys returns the list of the contained digits
op  Parse: key-list -> digit-list
ax  Parse(Empty) = Empty
ax  Parse(Key(d) ^  lk) = d ^  Parse(lk)
ax  Parse(Key(c) ^  lk) = Parse(lk)

**  given a number returns its representation as list of digits
op  Cod: nat -> digit-list
op  Cod': nat -> digit-list
ax  Cod'(0,ld) = 0 ^ ld
ax  . . .
ax  Cod'(9,ld) = 9 ^ ld
ax  if k < 10 then Cod'(m*10 + k) = Cod'(m,Cod'(k,ld))
ax  Cod(m) = Cod'(m,Empty)

op  Decod: digit-list -> nat
ax  Decod(0 ^ ld) = Decod(ld) * 10 + 0
...
ax  Decod(9 ^ ld) = Decod(ld) * 10 + 9

**  given two lists of digits, ld1 and ld2, and a command c returns the
**  list of digits corresponding to the evaluation of c on ld1 and ld2
op  Eval: digit-list command digit-list -> digit-list
ax  Eval(ld,c,Empty) = ld
ax  if Not_Empty(ld2) then  Eval(ld1,==,ld2) = ld2
ax  Eval(ld1,+,ld2) = Cod(Decod(ld1) + Decod(ld2))
ax  Eval(ld1,*,ld2) = Cod(Decod(ld1) * Decod(ld2))
ax  Eval(Empty,c,ld) = Eval(0 ^  Empty,c,ld)
```

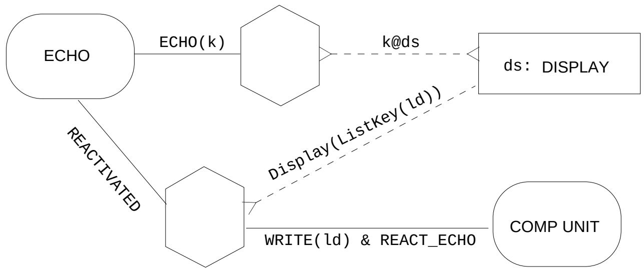
The interactions of the calculator



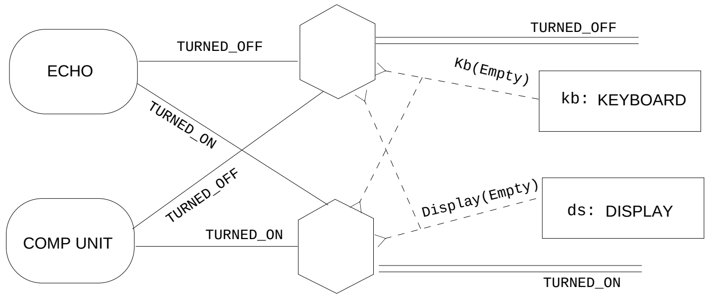
The calculator activity using macro-cooperation kinds



Macro-cooperation kind INT



Macro-cooperation kind ON-OFF



Macro-cooperation kind USER

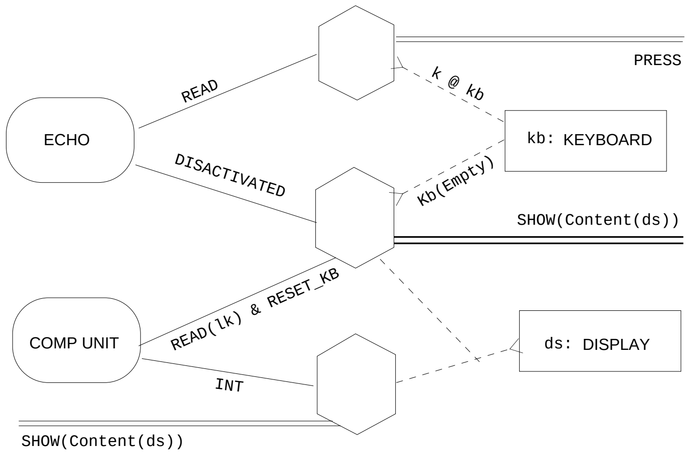


Table of Contents

This article was processed using the LaTeX macro package with LLNCS style