Casl-Chart: a Combination of Statecharts and of the Algebraic Specification Language Casl*

G. Reggio and L. Repetto

DISI - Università di Genova, Italy
reggio@disi.unige.it
http://www.disi.unige.it/person/ReggioG/

1 Introduction

In this paper we present CASL-CHART a formal visual specification language for reactive systems obtained by *combining* an already existing language for reactive systems, precisely the statecharts as supported by STATEMATE ([6, 7]), with an already existing language for the specification of data structures, precisely the algebraic specification language CASL ([12, 17]).

We think that is valuable to try to combining an existing specification language, intended for some particular applications following some particular paradigm, with another one aimed at the specification of the data, indeed usually the former is rather poor for what concerns the data part, and that prevents to use it effectively and productively for non-toy applications.

LOTOS, [9], a well established and used specification language for concurrent systems, has been developed when trying to use the CCS ("Calculus of Communicating Systems" [10]) for realistic applications, precisely the specification of protocols. At that point, it became clear the need to extend CCS with the possibility of specifying non trivial data; this extension was done by combining CCS with the already existing algebraic specification language ACT ONE [5].

There are many reasons because the designers of a specification language neglect the data part. In general their main efforts concern the peculiar constructs of the language, while the data part is filled in some way. Sometimes, they even think that the data are not relevant, since they have not tried to use their language for a non trivial application. In other cases, they just think that data and their transformations can be specified indirectly by using the specific constructs of the language This is usually true from a computational point of view, but not if we consider the use of the language in practice; look, for example, to the specification of queues and stacks realized by particular CCS processes in [10].

On the other hand, the algebraic specification languages, which are extremely apt to specify data structures, cannot be straightly used to specify reactive, concurrent, parallel,... systems; thus in the literature of the last years we can find many attempts to extend such languages in some way to cope with such applications (see [1] for an extended survey).

* Work supported by CoFI (ESPRIT Working Group 29432).

The above problem has been considered in the European "Common Framework Initiative" (CoFI) for the algebraic specification of software and systems, partially supported by the EU ESPRIT program [12][1], which brings together research institutions from all over Europe. A result of CoFI is the development of a specification language called CASL ("Common Algebraic Specification Language" [17]) that intends to set a standard unifying the various approaches to algebraic specification and specification of abstract data types. Indeed within CoFI, the "Reactive Systems Task Group" has the "... aim and scope of proposing and develop extensions of the common framework to deal with reactive, concurrent and parallel systems ...".

CASL-CHART is an attempt to work out a possible extension of CASL for the specification of reactive systems.

We have chosen the statecharts, as the language for the specification of the reactive systems, more precisely the statechart variant supported by STATEMATE [7], because it is visual, formal, well established, widely spread and used in industry. Moreover, statecharts, also if with a very different semantics, have been incorporated in the UML [18], the OMG recent standard notation for object oriented systems.

CASL-CHART is a combination of two formal specification languages. For us "combination" means that the original features of the two languages with their original semantics will be present in the combination; thus all data used in a CASL-CHART specification of a reactive system will be specified by using CASL, while the behaviour of such system will be specified by a statechart.

As a result we hae that whenever someone uses CASL-CHART, she/he does not specify algebraically the reactive aspects, but instead she/he uses the statechart machinery (events, steps, ...), and she/he specifies the data with CASL using its particular logic (many-sorted partial first-order logic). Thus, a CASL/ STATEMATEuser may take advantage of its previous know-how on using such languages, when she/he uses CASL-CHART.

The semantics of the combined languages will be given by "combining" the original semantics of the two languages. Indeed, here we are not interested to give the semantics of CASL-CHART by using CASL, and so in some sense to translate the statechart into CASL (also if that it is possible, see for example in [14] a semantics of the UML statecharts given by using CASL).

Here, we present CASL-CHART on a toy example, a pocket calculator, that it is small but it is sufficient to illustrate the novelties of CASL-CHART w.r.t. the classical statecharts supported by STATEMATE. A full presentation of CASL-CHART (precise visual syntax, reference manual, static and dynamic semantics) can be found in [15]. Unfortunately, we have not here the space to present algebraic specification and statecharts; the reader may refer, e.g., to [2], for the former, and to [8], for the latter.

At this time we are not aware of other attempts to combine statecharts with an algebraic specification language, whereas there are many proposals for putting

[1] More information on CoFI at `http://www.brics.dk/Projects/CoFI/`.

together algebraic specification languages with other notations for coping with concurrency, parallelism, reactivity, and so on, see the extended survey in [1].

There are also proposals for combining statecharts with other specification languages; for example in the German EXPRESS project a specification language $\mu\mathcal{SZ}$ combining the statecharts with Z has been developed ([4, 19]); however in this case both Z and the statecharts languages have been extended/and or modified.

2 The Example: a Pocket Calculator

In this paper we use, as a running example, a pocket calculator with a keyboard, a display, and a printer. More precisely it is a small reactive system simulating a pocket calculator (think of, for example, a small application simulating a graphical calculator on the desktop of your PC).

This calculator may

- receive keys from a keyboard: either digits, used to express numbers, or commands (arithmetic operations and printer commands);
- echo the numbers on a display;
- compute the commands corresponding to operations and shows the results on the same display;
- execte the printer commands (print the display content and start a new line).

We concurrently structure the calculator systems as the parallel of four processes: three drivers, taking care of the interactions with the keyboard, the display and the printer respectively, and a computing unit, executing the operations. In Fig. 1 we show using a visual-informal notation how such components cooperate to realize the functionalities of the calculator.

In the following sections we show how we have specified the calculator system using CASL-CHART, showing in the meantime the various features of this visual/formal specification language.

3 The Calculator Specification: a CASL-CHART

A CASL-CHART specification of a reactive system consists of

- a specification of the data used by the system, presented using the algebraic specification language CASL,
- and of a statechart that uses such data.

3.1 The Data Part: a Casl Specification

The informal description of our design of the calculator, see Fig. 1, uses keys (that are digits and commands, that are in turn arithmetic operations, commands for the printer and the equal), numbers (i.e., sequences of digits), and characters. We have specified these data are specified by means of the following CASL specification, which shows many features of that language.

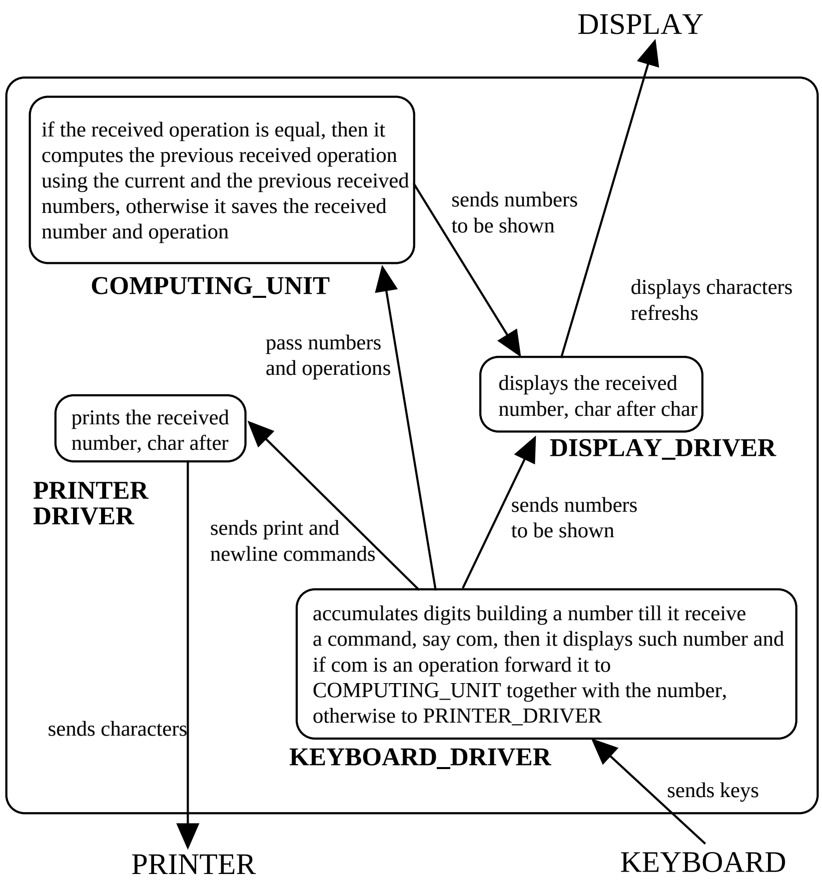


Fig. 1. Concurrent Structuring of the Calculator System

spec DATA =
  CHAR **then**
free {
  types
    $Digit ::= 0 \mid 1 \mid \ldots \mid 9;$
    $Operation ::= plus \mid mult \mid min \mid div;$
    $Command ::=$ **sort** $Operation \mid pr \mid nl \mid eq;$
    $Key ::=$ **sort** $Digit \mid$ **sort** $Command;$
    $Num ::= null \mid \_\_ . \_\_(Digit; Num);$ %% numbers as sequences of digits
  ops $first : Num \to ?\ Digit$
    $rest : Num \to ?\ Num$
    $\_\_\ to\_char : Digit \to Char$
  pred $is\_null : Num$
  vars $d{:}Digit;\ n{:}\ Num$
- $first(d\ n) = d$
- $rest(d\ n) = n$
- $0\ to\_char = '0'$

  ...
- $9\ to\_char = '9'$
- $is\_null(null)$

} **end**

DATA is the extension (CASL keyword **then**) of the specification CHAR, provided by the standard CASL libraries [16], with some data types, some operations, and a predicate. The CASL construct **types** allows one to provide for the given types, the constructors (either constant, as *null*, or with parameters, as $\_\_ . \_\_$, for *Num*), and possible subtypes (*Operation* is a subtype of *Command*, and *Key* is the disjoint union of *Digit* and *Command*).

CASL allows the users to freely define the syntax of the operations and predicates in a specification; e.g., $\_\_\ to\_char$ states that *to_char* is a postfix operation, and $\_\_ . \_\_$ that . is infix.

The underlying logic of CASL is many-sorted partial first-order logic. Thus in a CASL specification it is possible to declare total (as *to_char*) and partial operations (as *first*), using $\to$ and $\to?$ respectively, and predicates (as *is_null*).

"=" in axioms stands for *strong equality*: $t = t'$ holds iff both terms are defined and have the same value, or both are undefined[2]. It is possible to require the definedness of a term with the special atom *def t*.

The keyword **free** states that the specification DATA has an initial semantics; such semantics is characterized by the fact that an atom (either an equation or a predicate application) holds in such model iff it can be proved in the sound and complete deductive system for the CASL logic.

Thus, in the initial model of DATA we have that *is_null* does not hold on $0\ .\ null$, and that $first(null)$ is undefined because we cannot prove that $is\_null(0\ .\ null)$ and $def\ first(null)$.

[2] CASL provides also for existential equality $=_e$: $t =_e t'$ holds iff both terms are defined and have the same value.

CASL provides also a rich set of constructs for structuring specifications and for "architectural specifications" that are not used in this simple example.

3.2 The Whole System Behaviour: a Chart

We specify the behaviour of a reactive system with a statechart, denominated in CASL-CHART simply *chart*, whose form is very similar to that of the STATEMATE statecharts. In Fig. 2 we show the chart defining the whole calculator system.

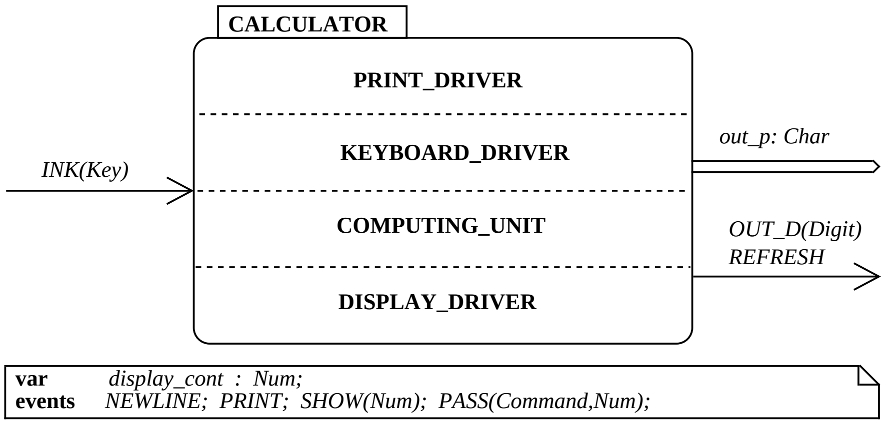


Fig. 2. Chart: **CALCULATOR**

is the icon for the chart, while **CALCULATOR** is the name of the chart, or better of its upper level state.

A reactive system may interact with its external environment in a discrete and in a continuous way. In a chart the first kind of interaction is provided by the *events*. The events may be received from outside (input events) graphically presented in CASL-CHART by a simple incoming arrow attached to the chart icon, and sent outside (output events) graphically presented by an outgoing simple arrow. Moreover, in CASL-CHART events may have parameters, that are values of the types defined by the CASL specification of the data.

The chart **CALCULATOR** has an input event *INK* parameterized by an element of type *Key*, and two output events *OUT_D* and *REFRESH*, the latter without parameters.

The *variables* provide the continuous interactions in a chart. Similarly to the events, they are distinguished in input (which can be only read by the chart) and output (that can be only written by the chart) and are typed using the basic types specified by the CASL part. The variables are graphically presented in CASL-CHART by double arrows, incoming for input and outgoing for output ($\Leftarrow$, $\Rightarrow$).

The calculator has a unique output variable *out_p* of type *Char*.

The not box below the chart contains the declarations of the *local events* and of the *local variables* that will be used, instead, only by the chart itself.

A reactive system may be composed by several components operating in parallel; in CASL-CHART such components are named *hierarchical charts*, and the parallel decomposition of a chart is shown by splitting the chart icon in slots by means of dashed lines.

CALCULATOR is made of four components: **PRINTER_DRIVER**, **KEYBOARD_DRIVER**, **COMPUTING_UNIT** and **DISPLAY_DRIVER**.

In this case we present the hierarchical charts corresponding to the four components on different drawings, but we could also have put them inside the various slots of **CALCULATOR**.

The chart describes a reactive system that moves step after step. At each step depending on the received input events, on the local events generated in the previous step, and on the values of the variables, its various components will perform a step, clearly those that can do it, generating output and local events and modifying the values of the variables.

3.3 The System Components: Hierarchical Charts

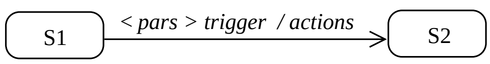
Display Driver The display driver is a very simple example of hierarchical chart. Indeed, its behaviour is very simple: it waits until it detects the occurrence of the local event $SHOW(n)$, then it records n in two local variables *to_display* and *display_cont*; thus it forwards the digits composing n to the display one after the other; finally it goes back to wait.

The hierarchical chart in Fig. 3 visually presents such behaviour.

A hierarchical chart is "hierarchically/sequentially" decomposed in several states, and in any moment only one of them is active. **DISPLAY_DRIVER** has three states: the initial state, represented by ●, which will be active at the beginning, WAITING_D, and DISPLAYING.

The local variable *to_display* is declared in the box below the chart icon, because its scope consists of just this hierarchical chart; while the scope of *display_cont* is the whole chart **CALCULATOR**.

The three transitions of this hierarchical chart have different forms, but the general form of a transition in CASL-CHART is



where

- S1 and S2 are two states of the chart, as in classical statecharts (source and target states).
- *trigger* is the transition trigger, and is a CASL formula (thus a formula of the many-sorted partial first-order logic) built using also the chart variables (input and local) and special atoms related to the statechart machinery,

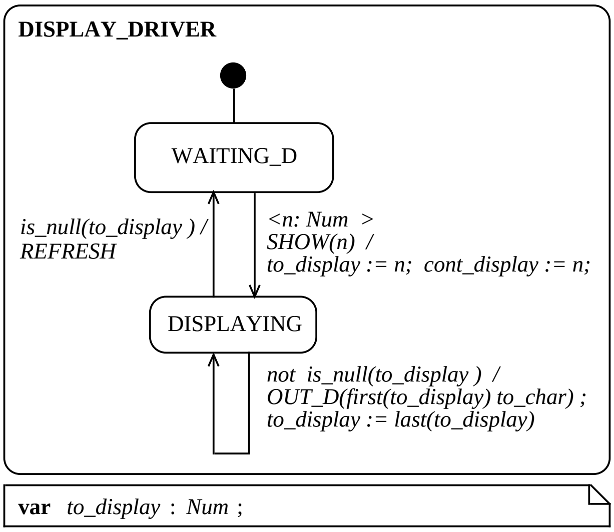


Fig. 3. Hierarchical Chart: **DISPLAY_DRIVER**

checking, for example, for the happening of events and for a state of whole chart to be active.
The CASL-CHART transition trigger is a combination of the trigger and of the condition part of the transitions of the classical statecharts.

- *actions* are the statements describing what to do when the transition is fired, such statements include, for example, assignments to the variables (local and output) and the generation of events; this part is similar to the corresponding one of the classical statecharts.
- *pars* are the transition parameters, this part, differently from the others, is not present in any form in the classical statecharts.[3] It consists of a list of variables typed using the types of the CASL specification of the data followed by a condition
  $< x_1 : t_1, \ldots, x_n : t_n \bullet cond >$,
  $x_1, \ldots, x_n$ are the transition parameters and may appear in *trigger* and in *actions*.
  For any instantiation of the transition parameters with $v_1, \ldots, v_n$, values of the appropriate types, s.t. *cond* holds there is a transition obtained by replacing $x_1, \ldots, x_n$ by $v_1, \ldots, v_n$.

There exists a default value for any part of the transition to be used if such part is lacking: the default for the for the trigger is the always true formula, that for the actions is the null statement, and that for the parameters is the empty set of parameters.

[3] UML statecharts have something of similar because also their events are parameterized, see [18].

At each step, a transition whose source state is active and whose trigger holds will be fired; if several transitions with the same source state may fire one of them will be nondeterministically chosen. After the firing the associated actions will be executed, and the target state will become active (clearly the source will be not active except when it coincides with the target).

The transition from WAITING_D to DISPLAYING means that for any value n of type *Num*, if the event $SHOW(n)$ has happened, then n will be assigned to the variables *to_display* and *display_cont*.

The transition from DISPLAYING to itself is not parameterized, it may be fired when the content of *to_display* is not null, and when it will fire the event

$$OUT\_D(first(to\_display)\ to\_char)$$

will be generated and the content of the variable *to_display* will be modified. Notice that *is_null*, *first*, *rest*, and __ *to_char* are predicates and operations specified in DATA.

The last transition of the hierarchical chart for the display driver is very simple; indeed it has just the trigger part consisting of checking whether the content of *to_display* is null.

Printer Driver The hierarchical chart specifying the behaviour of the printer driver is quite similar to that of the display driver, see Fig. 4. They differ only because the printer driver passes the character to be printed to the printer by using the output variable *out_p*, instead of generating an event, and gets the number to be printed by looking at the local variable *display_cont*, instead of receiving it as a parameter of the event firing the printing procedure.

Keyboard Driver The behaviour of the keyboard driver is more complex and we specify it with the hierarchical chart in Fig. 5.

The keyboard driver receives keys from the keyboard by means of the events $INK(k)$, then its behaviour depends on k. If k is a digit, checked by $k \in Digit$[4], then it is accumulated to build a number, otherwise the accumulated number is shown on the display. "*in* WAITING_D" checks that the display driver is in the state WAITING_D, and thus it is not showing another number. Afterwards, if k is the print or newline command, then it is passed to the printer driver by generating the corresponding events; otherwise, i.e., it is an operation, checked by $k \in Operation$ (*Operation* is a subtype of *Command*), it is passed to the computing unit together the accumulated number.

Computing Unit The computing unit is the last component of the **CALCULATOR** and is specified by the hierarchical chart in Fig. 6.

[4] In the specification DATA *Digit* is a subtype of *Key*, and CASL provides special predicates "$\in$" for checking if an element belongs to a subtype.

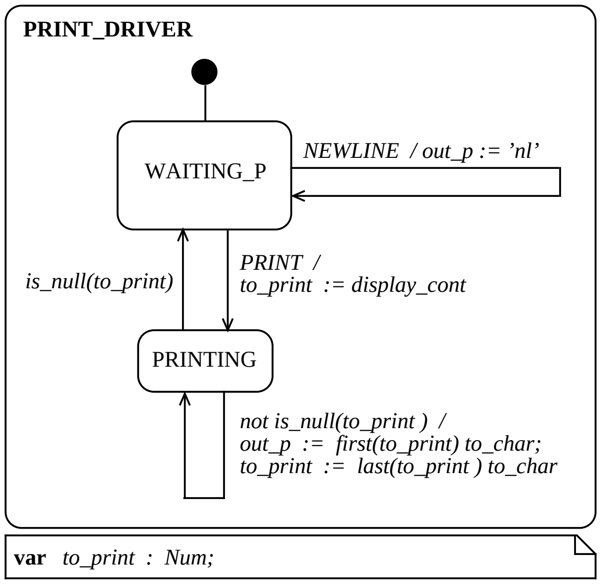


Fig. 4. Hierarchical Chart **Printer_Driver**

The essential role of this component is to compute the result of the application of the received operation to two numbers, represented by sequences of digits. We specify its behaviour rather simply by using a partial operation

$$apply : Operation \times Num \times Num \to ?\ Num,$$

specified by using CASL instead of using the statechart machinery. Technically, we extend the union of the specification DATA with that of the integers INTEGER by *apply* using also some auxiliary operations (*code* and *decode*).

spec APPLY =
  DATA **and** INTEGER **then**
  ops $apply : Operation \times Num \times Num \to ?\ Num$
    $code : Integer \to ?\ Num$
    $decode : Num \to Integer$
  vars $n, n' : Num$;
  axioms
    $apply(plus, n, n') = code(decode(n) + decode(n'))$
    $apply(mult, n, n') = code(decode(n) * decode(n'))$
    $apply(min, n, n') = code(decode(n) - decode(n'))$
    $apply(div, n, n') = code(decode(n)/decode(n'))$
    $\forall\ n : Num \bullet code(decode(n)) = n$
    $\forall\ i : Integer \bullet i > 0 \ \Rightarrow\ decode(code(i)) = i$
} **end**

This example shows how in CASL-CHART the data transformations may be specified algebraically, and thus in a quite abstract way. Moreover, they can be

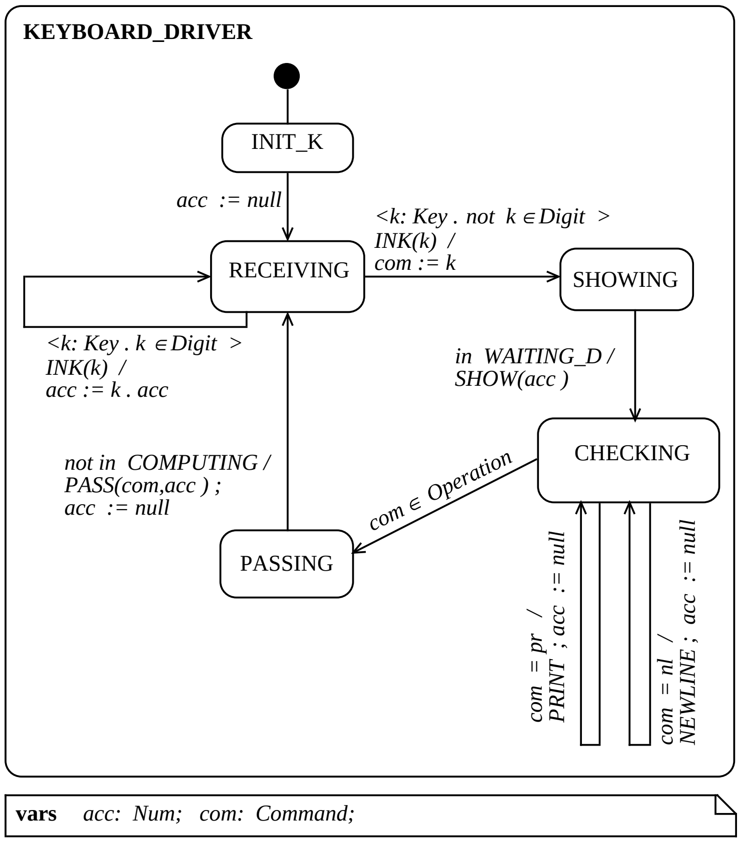


Fig. 5. Hierarchical Chart **Keyboard_Driver**

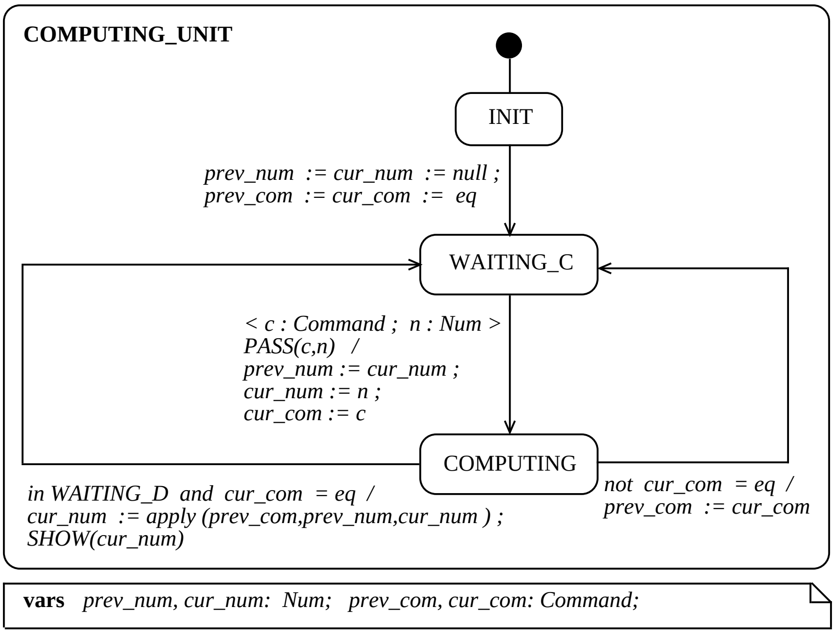


Fig. 6. Hierarchical Chart **Computing_Unit**

checked by using the various tools supporting the algebraic specifications (as theorem provers and rapid prototypers), see, e.g., [11].

3.4 Other Casl-Chart Constructs

To produce the specification of the pocket calculator, presented in the preceding sections, we have did not used all the Casl-Chart constructs. We want to recall that in Casl-Chart we have also:

- multilevel charts. In the examples shown in this paper, the states of the hierarchical charts are simple states, but they can be decomposed by associating with them a chart, that can be drawn either inside the state icon or on a separate sheet. The charts and hierarchical charts used in the decompositions may be accompanied by declarations of local variables, as we have seen before, and of local events; while the input/output variables and events are allowed only for the upper level chart, describing the whole system (as **CALCULATOR** in our example).
- The "real time" temporal combinators of Statemate statecharts may be used in the triggers; precisely: *before*, *since* and *at*, requiring that some condition holds before x time unit, since x time unit, at the time x.
- there are special events corresponding to enter/to exit a state.

4 Semantics

The complete static and dynamic semantics of CASL-CHART is in [15] and we cannot report it here due to lack of space.

The semantics of the CASL specification of the data results in a set of algebras (first-order structures) $\mathcal{D}$ defining the data used in that particular specification. Given $D \in \mathcal{D}$, we define a possible semantics of the CASL-CHART specification using D following the semantics of the STATEMATE statecharts of [7], to be more precise we give two semantics as in [7], a micro-step and a macro-step semantics.

The only difference between our semantics and [7] is when evaluating conditions and expressions, in such point we follow the CASL semantics (see [13]) by considering the special terms (as the statechart variables and the transition parameters) as extra constant operations and the special atoms (as those checking whether an event has happened or that some state of the chart is active) as extra zero-ary predicates.

This combination of the semantics should allow also for a combination of existing support tools. For example, if we restrict the used CASL specifications to conditional specifications, then the STATEMATE toolset could be extended to a toolset for CASL-CHART by replacing the modules taking care of the evaluation of expressions and of conditions with, for example, a rewrite tool for CASL.

5 Conclusion and Future Works

We have presented, on an example, CASL-CHART, that is a combination of the algebraic specification language CASL and of the statecharts as supported by STATEMATE. See [15] for a complete presentation of the syntax and of the semantics of CASL-CHART.

The CASL-CHART specification language is truly a combination of CASL and of the statecharts, because both the ways to specify data structures and the statechart machinery have been preserved, at the syntactic level and also at the semantic level. We have thus obtained a CASL extension for the specification of reactive systems, that is a way to use CASL for relevant practical applications, and, in our opinion, also a better variant of statecharts.

The advantages of CASL-CHART w.r.t. STATEMATE statecharts are

CASL-CHART **specifications may be more abstract** because the used data and their transformations may be axiomatically specified (e.g., in our small example we have not detailed described as to perform the coding, decoding of numbers, but we have just asserted that such operations are one the inverse of the other, see the specification APPLY). In STATEMATE statecharts, instead such data elaborations have to be described in a very detailed way by using the statechart machinery.

CASL-CHART **specifications are more compact** because in CASL-CHART we have the possibilities

– of defining apart, in the CASL specification, the user defined data with an user defined syntax, and the operations and the predicates to operate on them;
– of parameterizing events, and thus transitions, over values.

The above features, allows the CASL-CHART users to write statecharts whose visual diagram is simpler and smaller than a corresponding STATEMATE one. We think that this aspect of CASL-CHART is rather valuable, because one of the problems with the visual languages is to avoid that the dimensions of the drawings become too large.

STATEMATE offers three visual notations for the specification of a reactive system during its development: statecharts, that we have considered in this paper, for describing reactive components, *activity charts* to describe the structure of the system in terms of logical components (statecharts and of other kinds), and *module charts* to describe the structure of the system at the implementation level (similar to the deployment diagrams of UML [18]). We plan to extend CASL-CHART to cope with activity and module charts. For what concerns module charts, we plan to investigate their relationships with the architectural specification of CASL (see [3]) developed with similar aims.

Statecharts are one of the many notations incorporated by UML ([18]), and so also if their semantics is partly different from that of the STATEMATE statecharts, we think that this work could offer a starting point for developing a combination of UML with the algebraic specification language CASL, allowing to offer an alternative to OCL, soundly founded on a formal notation.

References

1. E. Astesiano, M. Broy, and G. Reggio. Algebraic Specification of Concurrent Systems. In E. Astesiano, B. Krieg-Bruckner, and H.-J. Kreowski, editors, *IFIP WG 1.3 Book on Algebraic Foundations of System Specification*, pages 467 – 520. Springer Verlag, 1999.
2. E. Astesiano, B. Krieg-Bruckner, and H.-J. Kreowski, editors. *IFIP WG 1.3 Book on Algebraic Foundations of System Specification*. Springer Verlag, 1999.
3. M. Bidoit, D. Sannella, and A. Tarlecki. Architectural specifications in CASL. In *Proc. 7th Int. Conf. Algebraic Methodology and Software Technology (AMAST'98), Amazonia, Brazil*, Lecture Notes in Computer Science, pages 341–357. Springer Verlag, Berlin, 1999.
4. R. Bussow, R. Geisler, and M. Klar. Specifying Safety-Critical Embedded Systems with Statecharts and Z: A Case Study. In E. Astesiano, editor, *Proc. FASE'98*, number 1382 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1998.
5. H. Ehrig, W. Fey, and H. Hansen. ACT ONE: An Algebraic Specification Language with two Levels of Semantics. Technical Report 83-01, TUB, Berlin, 1983.
6. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *EEE Transactions on Software Engineering*, 16(4):396–406, 1990.
7. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

8. D. Harel and M. Politi. *Modeling Reactive Systems With Statecharts : The Statemate Approach.* McGraw Hill, 1998.
9. I.S.O. ISO 8807 Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. IS, International Organization for Standardization, 1989.
10. R. Milner. *A Calculus of Communicating Systems.* Number 92 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1980.
11. T. Mossakowski. CASL: From Semantics to Tools (tool). In *Proc. TACAS 2000*, Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000. To appear.
12. P.D. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In M. Bidoit and M. Dauchet, editors, *Proc. TAPSOFT '97*, number 1214 in Lecture Notes in Computer Science, pages 115–137, Berlin, 1997. Springer Verlag.
13. The CoFI Task Group on Semantics. CASL The Common Algebraic Specification Language: Semantics CoFI Note S-9. Technical report, 1999. `ftp://ftp.brics.dk/Projects/CoFI/Notes/S-9/`.
14. G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In *Proc. FASE 2000 - Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000. To appear.
15. G. Reggio and L. Repetto. CASL-CHART : Syntax and Semantics. Technical Report DISI-TR-00-1, DISI – Università di Genova, Italy, 2000. `ftp://ftp.disi.unige.it/person/ReggioG/ReggioRepetto00a.ps`.
16. M. Roggenbach and T. Mossakovski. Basic Data Types in CASL . CoFI Note L-12. Technical report, 1999. `http://www.brics.dk/Projects/CoFI/Notes/L-12/` .
17. The CoFI Task Group on Language Design. CASL The Common Algebraic Specification Language Summary. Version 1.0. Technical report, 1999. Available on `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`.
18. UML Revision Task Force. *OMG UML Specification*, 1999. Available at `http://uml.shl.com`.
19. M. Weber. Combining Statecharts and Z for the Desgin of Safety-Critical Control Systems. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, number 1051 in Lecture Notes in Computer Science, pages 307–326. Springer Verlag, Berlin, 1996.