From Conditional Specifications to Interaction Charts

A Journey from Formal to Visual Means to Model Behaviour

E.Astesiano and G. Reggio

DISI - Università di Genova, Italy
reggio, astes@disi.unige.it

Abstract. In this paper, addressing the classical problem of modelling the behaviour of a system, we present a paradigmatic journey from purely formal and textual techniques to derived visual notations, with a further attention first to code generation and finally to the incorporation into a standard notation such as the UML.
We show how starting from Casl positive conditional specifications with initial semantics of labelled transition systems, we can devise a new visual paradigm, the interaction charts, which are diagrams able to express both reactive and proactive/autonomous behaviour.
Then, we introduce the executable interaction charts, which are interaction charts with a special semantics, by which we try to ease the passage to code generation.
Finally, we present the interaction machines, which are essentially executable interaction charts in a notation that can be easily incorporated, as an extension, into the UML.
Keywords: design of visual notations, formal notations, behaviour modelling/specification, CASL, UML, interaction charts

1 Introduction

In a remarkable paper [10], celebrating and assessing a decade of TAPSOFT in 1995, Ehrig and Mahr, after admitting some disproportion between the original claims of formal methods and their real impact on software practices, were however insisting on the need of rooting engineering practices on "the contributions from theoretical and conceptual work". That call was taken up and expanded by the authors first in a talk at the last TAPSOFT (Lille,1997) [2, 3] and later on in some papers advocating the use of "well-founded methods" more than "formal methods" (see [5] for a general presentation). Well-founded methods are precisely rooted on theoretical and conceptual models, but presented in a way that is friendly for the user and more concerned with the practical engineering needs. In this paper, addressing the classical problem of modelling the behaviour of a system, we present in a sense a paradigmatic journey from purely formal and textual techniques to derived visual notations, with a further attention first to code generation and finally to the incorporation into a standard practical notation such as the UML [13]. A bit more precisely, we show how starting from a formal specification technique, namely, positive conditional specifications with initial semantics of labelled transition systems, expressed using the Casl specification language [6, 12], we can devise a new visual paradigm, which can also be adopted for an extension of UML. The paradigm is centered on the interaction charts, which are diagrams able to express also a proactive/autonomous behaviour, in opposition to the only reactive behaviour of the state charts and of the UML state machines.

The first main new contribution of this paper is the introduction of the executable interaction charts, by which we try to tackle the problem of the treatment of the nondeterministic choice among various alternatives when moving from abstract formal notations to more practical notations that need a kind of operational/executable semantics in order to ease the passage to code. Whereas there are no needs to restrict the alternatives in the first case, namely, it is possible specify a system that may nonderministically choose among a set of activities of any kind (internal, inputting, outputting, a mixture of inputting and outputting), in the latter either the sets of activities among which to choose are restricted (for example, only inputting and at most one internal, as in UML and Ada programming language) or some mechanism is introduced to be able to discover which alternatives are feasible in a certain situation (e.g., event queues/pools of UML). Executable interaction charts follow the second choice, by proposing the use of abstract buffers. The result is an abstract and executable visual notation to specify/model interactive behaviour, a kind of behaviour commonly found in "client" components, proactive agents and so on.

The second contribution of the paper is the introduction of the interaction machines, which are essentially executable interaction charts in a notation that can be easily incorporated, as an extension, in the UML notation.

We start in Sect. 2 by briefly summarizing the use of conditional specifications for modelling the behaviour of systems, then we introduce in Sect. 3 the interaction charts, showing how they have been derived from the corresponding conditional specifications. In Sect. 4 we present the executable interaction charts, and finally in Sect. 5 we show how they can be used to extend UML with a new kind of diagrams, the interaction machines.

2 Free Positive Conditional Specifications for Modelling Behaviour

Here, we use the word *system* to denote a dynamic entity of whatever kind, and so evolving along the time, without any assumption about other aspects; thus a system may be a communicating/nondeterministic/sequential/... process, a reactive/parallel/concurrent /distributed/... system, but also an object-oriented system (a community of interacting objects), and an agent or an agent system.

For modelling the behaviour of systems we adopt the well-known and accepted technique based on labelled transition systems (see [11, 16, 4]), which is today standard, widely used, and proven adequate in many cases, and there is a huge literature. For example, labelled transition systems are the basic formal models that we have used for giving the semantics to Ada [1] and to UML [18, 19].

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

The behaviour of a system S may be represented by an lts $(STAT, LAB, \rightarrow)$ and an initial state $s_0 \in STAT$; then the states in $STAT$ reachable from s_0 represent the intermediate (interesting) situations of the life of S and the transition relation $\rightarrow$ the possibilities of S of passing from a situation to another one. It is important to note that here a transition $s \xrightarrow{l} s'$ has the following meaning: S in the state s has the *capability* of passing into the state s' by performing a transition whose interaction with the external (to S) world is represented by the label l. Thus the label l contains information on the conditions on the external world for the capability to become effective, and information on the transformation of such world induced by the execution of the action, i.e., it describes the interaction of S with the external world during such transition.

Labelled transition systems may be used also to model *structured systems* (i.e., systems built by putting together several subsystems, simple or in turn structured). The lts modelling a structured system is defined by composing the lts's describing its composing subsystems; the states of this lts are sets of states of the subsystems, and its transitions consist of the simultaneous execution of sets of transitions of the subsystems (at most one for each subsystem), see [4, 7].

Labelled transition systems may be specified by means of algebraic specifications having the form shown below. In this paper we present the algebraic specifications using the language CASL [6, 12]. CASL has been designed by CoFI[1], the international *Common Framework Initiative for algebraic specification and development.* It is based on a critical selection of features that have already been explored in various contexts, including subsorts, partial functions, first-order logic, and structured and architectural specifications.

A CASL specification may include the declarations of sorts, operations and predicates (together with their arity), and axioms that are first-order formulae with strong and existential equality and a *2-valued logics.* In CASL large and complex specifications are easily built out of simpler ones by means of (a small number of) specification building primitives, among them *union* (keyword '**and**') and *extension* can be used to structure specifications. Extensions, introduced by the keyword '**then**', may specify new symbols, possibly constrained by some axioms, or merely require further properties of old ones.

spec LTS = DATA$_1$ **and** ... **and** DATA$_r$ **then**
  sorts *State*, *Label*, ...
  ops ...
  preds $\_\_ \xrightarrow{\_\_} \_\_ : State \times Label \times State$
    ...
  axioms ...

[1] `http://www.brics.dk/Projects/CoFI`

where DATA$_1$, ..., DATA$_r$ are the names of the specifications of the basic data used to define the states and the labels.

Any algebra M that is a model of LTS defines a labelled transition system, precisely

$$(State^M, Label^M, \_ \xrightarrow{\_} \_^M).$$

By choosing appropriately the set of axioms of the above specification, it is possible to specify particular classes of lts, and thus particular classes of systems by characterizing their behaviour. However, first-order logic is not expressive enough to specify all relevant classes of lts, i.e., to express all relevant properties on them (see, e.g., [9, 4]); for example, using first-order logic it is not possible to require liveness conditions. A convenient solution is to extend the first-order logic with temporal combinators, as proposed by LTL (Labelled Transition Logic) presented in [9, 4], and its CASL version CASL-LTL [17].

If, instead, we want to specify a particular system with a given behaviour, that is a particular lts, we can use *positive conditional specifications* with free (initial) semantics. Such specifications in CASL have the form shown below. The CASL **free** construct defines free specifications, which are specifications having initial semantics. Such semantics avoids the need for explicit negation; indeed, in the models of free specifications, it is required that values of terms are distinct except when their equality follows from the specified axioms, and positive atoms built by predicates hold only when their truth follows from the specified axioms.

spec FCONDLTS = DATA$_1$ **and** ... **and** DATA$_r$ **then**
free { **sorts** $State, Label, \ldots$
ops ...
preds $\_ \xrightarrow{\_} \_ : State \times Label \times State$
...
axioms $PosCond$ **} end**

where DATA$_1$, ..., DATA$_r$ are the names of the free conditional specifications of the data used to define the states and the labels, and *PosCond* is a set of positive conditional formulae, which have the form $\wedge_{i=1,\ldots,n}\ \alpha_i \ \Rightarrow\ \beta$, where each α_i is a positive atom, i.e., either $pr(t_1, \ldots, t_m)$ or $t_1 = e = t_2$ (existential equation), and β is either $pr(t_1, \ldots, t_m)$ or $t_1 = t_2$ (strong equation).

The initial model I of FCONDLTS, unique up to isomorphism, defines the lts

$$(State^I, Label^I, \_ \xrightarrow{\_} \_^I).$$

Any element of $State^I$ and of $Label^I$ is the interpretation of a ground term, and we have that $I \models s \xrightarrow{l} s'$ iff $s \xrightarrow{l} s'$ follows from *PosCond*. Thus, any ground term *stat* of sort *State* represents a system, the one having as initial state the interpretation of *stat* in *I*.

We can specify algebraically also the structured systems, again by free conditional specifications, built by extending the union of the specifications of their subsystems. The transition predicates of the subsystems will appear in the premises of the axioms of these specifications, whereas the transition predicate of the structured system will appear in the consequences. For lack of room we do not further detail this topic, see, [4, 7].

3 Interaction Charts

In this section we introduce the *interaction charts* as the visual counterparts of the free conditional specifications of lts introduced in Sect. 2, and thus a visual notation to present lts, and so the behaviour of systems. A first version of interaction charts was presented in [20], recently refined in [7]; then a Java oriented version named *behaviour graph* was proposed as part of the notation JTN [8].

We restrict the considered class of free conditional specifications of lts's to be able to associate with them an interaction chart, by fixing the structure of the states and of the labels, and the form of the conditional axioms defining the transition predicate.

Here we consider two cases, which will result in two slightly different variants of interaction charts; which variant to use depends on the applications and on the specifier style.

Generator Variant, the states and the labels are defined by means of total generator operations.

Record Variant, the states have a record structure and the labels are defined by means of total generator operations.

3.1 Interaction Charts: Generator Variant

In this case we consider free conditional specifications of lts, written again in CASL, having the following form.

spec NAME = DATA$_1$ **and** ... **and** DATA$_r$ **then** **free** {
    sorts *State*, *Label*
    ops $sg_1 : \ldots \to State$ %% state generators
        ...
        $sg_n : \ldots \to State$
        $lg_1 : \ldots \to Label$ %% label generators
        ...
        $lg_m : \ldots \to Label$
    preds $\_\_ \xrightarrow{\_\_} \_\_ : State \times Label \times State$
    axioms *GPosCond* } **end**

where NAME is an identifier, DATA$_1$, ..., DATA$_r$ are the names of the free conditional specifications (given elsewhere) of the datatypes used to define the states and labels, and each element of *GPosCond* has the form

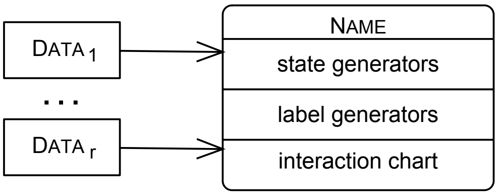
$$(*)\ cond \Rightarrow sg(t_1, \ldots, t_k) \xrightarrow{lg(t''_1,\ldots,t''_p)} sg'(t'_1, \ldots, t'_h)$$

where sg and sg' are state generators, lg is a label generator, $t_1,\ldots,t_k$, $t''_1,\ldots$, t''_p, t'_1, ..., t'_h are terms possibly with variables and *cond* is a conjunction of positive atoms, where $t_1,\ldots,t_k$, $t''_1,\ldots$, t''_p, t'_1, ..., t'_h and their subterms may appear while the transition predicate $\_\_ \xrightarrow{\_\_} \_\_$ cannot. Recall that the state and label generators are total operations[2].

[2] In CASL total operations are declared by $\ldots : \ldots \to \ldots$, whereas the partial operation by $\ldots : \ldots \to ? \ldots$.

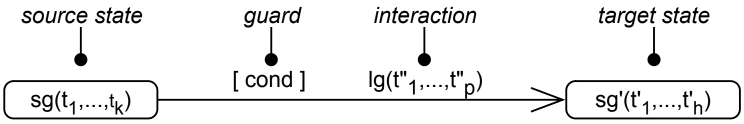
Note that in the initial model of this specification the states/labels represented by different generators or by the same generator applied to different arguments are different.

The visual notation for presenting the above *system specification* is[3]



In the above picture, a label generator $lg : \ldots \to Label$ is written $lg(\ldots)$, and similarly a state generator $sg : \ldots \to State$ is written $sg(\ldots)$, since in both cases the result type may be omitted because it is implicit. The *interaction chart* is the visual presentation of the set *GPosCond* of the conditional axioms of the specification defining the transition predicate.

A conditional axiom having form (*) is visually represented as



The visual presentations of all the axioms in *GPosCond* may then be put together building an oriented graph, as originally proposed in [20], by collecting together all rounded boxes related to states built by the same generator, and by writing only once repeated generator instantiations. The guards will be omitted when they are equivalent to true.

Example We give, in Fig. 1, the specification of a simple process (component) operating a calculation over up to 100 negative integers and refusing any positive number. IntPlus is the specification of integers extended with an operation *op*. To help understand the strong correspondence between the interaction chart and the conditional axioms of the corresponding specification, we report them below.

$run(100) \xrightarrow{null} stop$

$0 > N \;\Rightarrow\; run(CNT) \xrightarrow{receiveOk(N)} processing(N, CNT)$

$run(CNT) \xrightarrow{receiveKo(N)} ko$

$0 \leq N \;\Rightarrow\; run(CNT) \xrightarrow{receiveOk(N)} refusing(N, CNT)$

$refusing(N, CNT) \xrightarrow{refused(N)} run(CNT)$

$processing(N, CNT) \xrightarrow{result(op(N))} run(CNT + 1)$

Summarizing, an interaction chart is a labelled graph where

– nodes represent the relevant types/classes of situations in the life of the modelled system, during which some (usually implicit) invariant condition holds,

[3] Also the free conditional specifications of datatypes may be presented visually, see [20, 7].

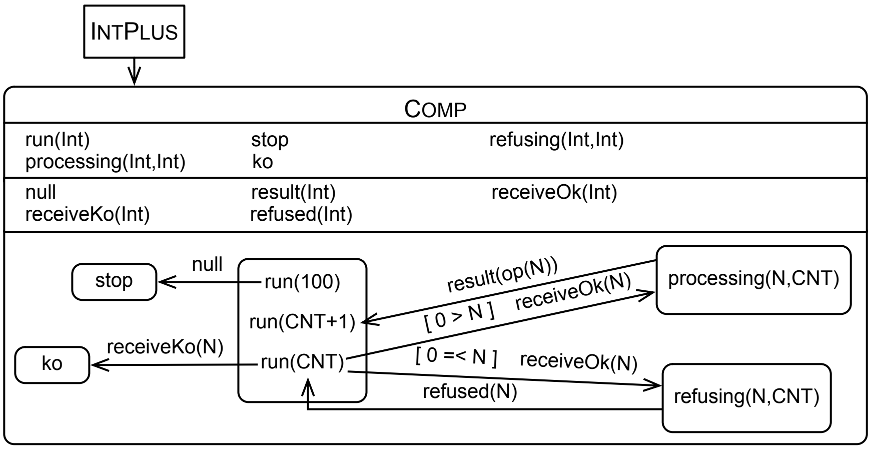


Fig. 1. Specification of a simple process with an interaction chart (generator variant)

– arcs represent the capabilities of the system of passing from a situation of one kind into another one of the same or of another kind and their labels describe the interaction of the system with the outside world during such move.

Thus, the interaction charts are a visual notation that follows the state-transition paradigm allowing to visually depict all the capabilities of interactions with the external environment of the modelled system, where transitions correspond to interaction capabilities, and interaction is intended as a description of the interchange between the modelled system and the external environment.

Notice that this is quite different from other visual notations, such as statecharts, where only the reactions to events coming from outside or from inside are visually depicted by the transitions. In some sense a statechart gives a picture of the reactive aspects of the behaviour of a system, whereas an interaction chart gives a picture of the interactive aspects of that behaviour.

3.2 Interaction Charts: Record Variant

In this case we consider free conditional specifications of lts, written again using CASL, having the following form.

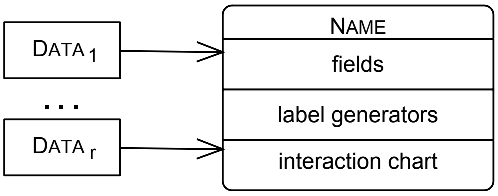
spec NAME = DATA$_1$ **and** ... **and** DATA$_r$ **and** STRING **then**
free { **sorts** $State, Label$
ops $<\_, \dots, \_>: s_1 \times \dots \times s_n \times String \to State$ %% record generator
$lg_1 : \dots \to Label$ %% label generators
...
$lg_m : \dots \to Label$
preds $\_ \xrightarrow{\_} \_ : State \times Label \times State$
vars $F_1 : s_1; \quad \dots\ F_n : s_n;$
axioms $RPosCond$ } **end**

where NAME is an identifier, STRING is a specification of strings of characters, DATA$_1$, ..., DATA$_r$ are the names of the free conditional specifications (given elsewhere) of the datatypes used to define the states and the labels, and each element of *RPosCond* has the form

(**) $cond \Rightarrow < F_1, \ldots, F_n, \text{“}ident_1\text{”} > \xrightarrow{lg(t_1,\ldots,t_m)} < t'_1, \ldots, t'_n, \text{“}ident_2\text{”} >$
where $ident_1$ and $ident_2$ are two identifiers (and so “$ident_1$” and “$ident_2$” are ground terms of sort *String*), F_1, …, F_n are variables of sorts s_1, …, s_n respectively (always the same for all the axioms), t_1, …, t_m, t'_1, …, t'_n are terms possibly with variables and *cond* is a conjunction of positive atoms, where $F_1, \ldots, F_n$, t_1, …, t_m, $t'_1, \ldots,$ t'_n and their subterms may appear and the transition predicate cannot. Again, recall that the record and the label generators are total operations.

Note that in the initial model of this specification the states are records with n fields and the labels represented by different generators or by the same generator applied to different arguments are different.

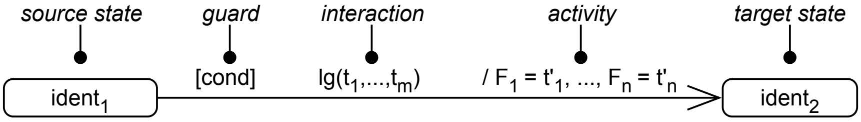
The visual notation for presenting the above system specification is



In the above picture, a label generator $lg : \ldots \rightarrow Label$ is written $lg(\ldots)$, as for the other variant. The fields are determined by the variables used to denote the state record components and are written $F_1 : s_1 \ldots F_n : s_n$.

The *interaction chart* is again a visual presentation of the set *RPosCond* of the conditional axioms defining the transition predicate.

A conditional axiom having form (**) is visually represented as



The visual presentations of all the conditional axioms may then be put together building an oriented graph by joining together all rounded boxes decorated by the same identifier.

The null field updates of the form F = F will be omitted, as well as the guard when they are equivalent to true.

Example We give, in Fig. 2, the specification of the same simple process used as example in Sect. 3.1. Again, to help understand the relationship of the interaction chart with the corresponding conditional axioms we report them below.

$CNT = 100 \;\Rightarrow\; < CNT, N, \text{“}run\text{”} > \xrightarrow{null} < CNT, N, \text{“}stop\text{”} >$

$0 > X \;\Rightarrow\; < CNT, N, \text{“}run\text{”} > \xrightarrow{receiveOk(X)} < CNT, X, \text{“}processing\text{”} >$

$< CNT, N, \text{“}run\text{”} > \xrightarrow{receiveError(X)} < CNT, N, \text{“}ko\text{”} >$

$$0 \leq X \;\Rightarrow\; < CNT, N, \text{“}run\text{”} > \xrightarrow{receiveOk(X)} < CNT, X, \text{“}refusing\text{”} >$$
$$< CNT, N, \text{“}refusing\text{”} > \xrightarrow{refused(N)} < CNT, N, \text{“}run\text{”} >$$
$$< CNT, N, \text{“}processing\text{”} > \xrightarrow{result(op(N))} < CNT + 1, N, \text{“}run\text{”} >$$

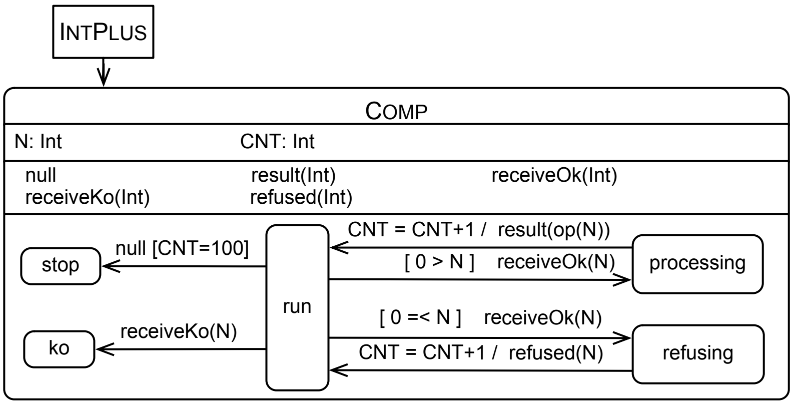


Fig. 2. Specification of a simple process with an interaction chart (record variant)

3.3 Specification of structured systems

We can visually present also the free conditional specifications of structured systems, by requiring they have the precise form described below.

– A transition of a structured system is made by the simultaneous execution of a group of transitions of its subsystems (obviously at most one for each subsystem), whose interactions form a set of *cooperations.*

– A cooperation is a set of complementary interactions, in the sense that the interactions being part of a cooperation can only be executed together; for example, sending and receiving a message along a channel, destroying a subsystem and being destroyed, sending a broadcast message and any number of reception of such message.

– There is a criterium for selecting which sets of cooperations will correspond to transitions; for example, interleaving (each transition corresponds to a unique cooperation), free parallel (each transition corresponds to a set of cooperations), and maximal parallelism (each transition corresponds to a maximal group of cooperations).

Here we do not have the room to present the details of the visual presentations of the cooperations and of the criteria; see [20, 7].

3.4 Interaction Charts: Additional Constructs

To effectively use the simple forms of interaction charts presented in Sect. 3.1 and 3.2, they have to be enriched with constructs allowing to easily present quite complex and large charts. Here we present some of them, those that we have found useful in the years; notice that some of them have been inspired by similar constructs of the UML state machines, whose introduction has been motivated by the needs of some of the proposers of the UML notation. Their semantics can be easily defined by transforming a chart using these features into a simpler one having the form defined in Sect. 3.1 or 3.2.

Syntactic facilities To help improve the layout of complex interaction charts.

- a state may be anonymous, i.e., the generator/the identifier is not written.
- a state may be depicted several times in a chart;
  - a generator chart may contain several rounded boxes decorated by patterns built by the same generator; such chart is equivalent to another one, where all those boxes are coalesced into a unique one including inside the decorations of all those boxes;
  - a record chart may contain several rounded boxes decorated by the same identifier; such chart is equivalent to another one, where all those boxes are coalesced into a unique one decorated by that identifier.
- a large system presentation may be split into several partial ones, where each one has the name compartment, and some of the other compartments (e.g., a representation containing the name, label and state generators, and another one containing the name and the interaction chart).

Initial/final states A node (at most one) of an interaction chart marked by ●⟶ is *initial* (only the states of the associated lts corresponding to that node may be used to determine the initial state of the specified system). The final states (any number), each one represented by ◉, explicitly show the end of the activity of the specified system; obviously no transition may leave a final state. A final state can be replaced by another one decorated by a zero-ary generator/identifier different from all those used in the chart.

For the record variant, the values of the fields in the initial state may be defined by decorating the arrow marking the initial state with $F_1 = t_1; \dots F_n = t_n$.

Local transitions (null interaction) A system may perform internal activity without any interaction with the external world, in this case we have transitions decorated by a null interaction. The null interaction is characterized by the fact that it takes part in a unique cooperation consisting just of itself. We assume that there is a unique predefined zero-ary label generator to represent a null interaction: null[4]. Moreover, null may be dropped from the transitions, to better depict the absence of interaction with the external environment.

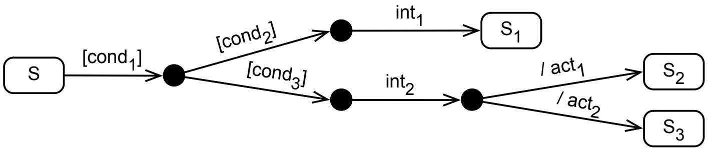
Factorizing transitions into segments It is useful to visually present a transition by joining many *transition segments* (that are not transitions) by a special symbol, the *junction*[5], visually presented by ●.

[4] Similar to the τ label of Milner's CCS.

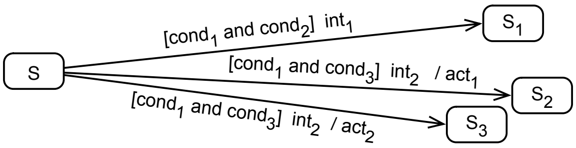
[5] We do not call a junction a pseudo-state as in UML [15], to stress that it is just a presentation mechanism without any special semantics in term of lts.

Technically, a *transition segment* is an arc either between two junctions or a junction and a state or a state and a junction annotated with a partial transition decoration (e.g., just a guard, an interaction, an activity, a guard and an interaction, ...). The meaning of junctions and segments is simply given by some replacement rules: a junction may be eliminated by connecting any incoming arc with any outgoing arc and annotating the resulting arc with the combination of the two decorations (clearly, not all combination of segments are correct, e.g., a guard cannot follow an activity).

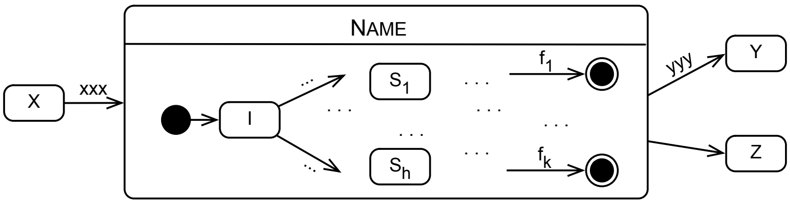
The factorization of transitions improves the readability of the charts, by splitting complex transitions into pieces, by avoiding to depict many times the same part of decoration, and also by making more clear which are the differences and the commonalities among some transitions.
For example the following fragment of interaction chart



stands for

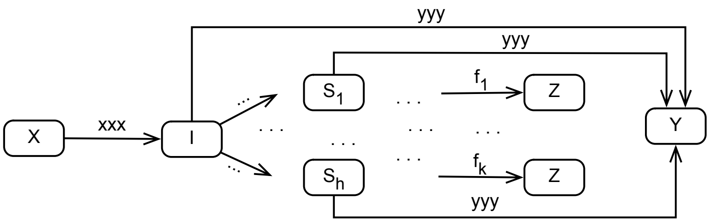


Composite (sequential) states A composite state is represented by a rounded box with a compartment containing the name and another one containing an interaction chart with a unique initial state and any number of final states. A composite state may be the target or source state of a transition, and the source of a unique special undecorated transition. Here we have a schematic generic composite state.



A composite state can be replaced by
– dropping the state icon,
– making the initial state the target state of any incoming transition,
– adding to any internal state (neither initial nor final) any outgoing transition,
– replacing the final states by the state target of the undecorated outgoing transition.

The above schematic composite state stands for the following fragment of interaction chart.

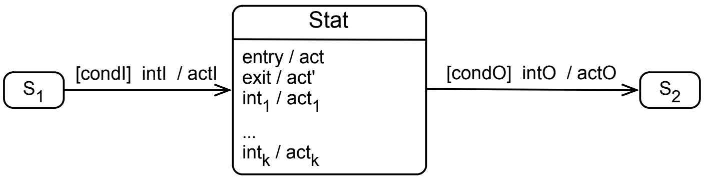


Because there is not a standard general well accepted way to define parallel composite states, and because the existing definitions are quite complicate always with subtle problematic points, we decided to avoid them in the interaction chart notation. Furthermore, this is not a big restriction; indeed, if we need to specify a system explicitly exhibiting a parallel behaviour it is always possible to see it as a structured system made by some subsystems cooperating among them in a parallel way.

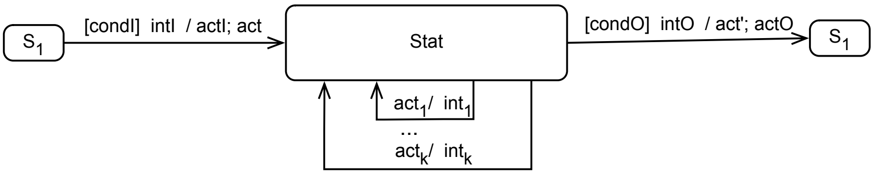
Subcharts Complex interaction charts may be modularly decomposed by defining and using subcharts. A *subchart* is an interaction chart with an initial state. Subcharts are declared in additional compartments of the system specifications that contain the name of the subchart (written in italic) and the interaction chart defining it. To include a subchart in the enclosing one it is sufficient to depict a rounded box with inside the name of the subchart, always written in italic. It stands for a composite state including the definition of that subchart.

Entry/exit actions and internal transition (only for the record variant) Entry/ exit actions and internal transition are associated to the states of an interaction chart. An *entry action* associated with a state is executed, as last thing, whenever a transition having that state as target is executed. An *exit action* associated with a state is executed, as first thing, whenever a transition having that state as source is executed. An *internal transition* presents an interaction capability that does not change the state.

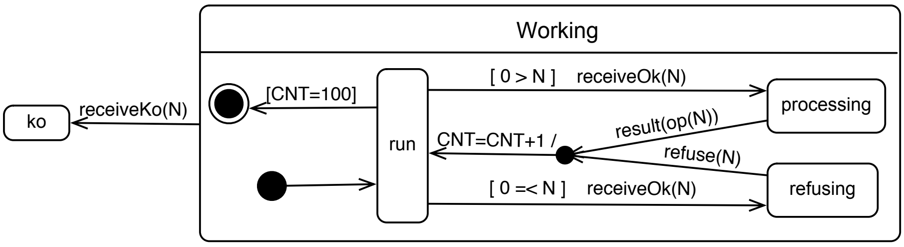
The following picture shows a generic state, named Stat, with one entry action, one exit action and k internal transitions, plus a generic incoming and a generic outgoing transition.



It stands for



Example We consider a variant of the simple process used as examples already in Sect. 3.1 and 3.2. In this case the counter is incremented also when a number is refused and the process may break down in any state, not only in the initial one. For simplicity, here we only give the new interaction chart using several of the additional constructs, among them initial, final and composite states, compound transitions and null interaction.



4 Executable Interaction Charts

The interaction charts presented in Sect. 3 specify the (simple) systems considered in isolation in an abstract formal way, by defining an lts determined by the corresponding free conditional specification. The concept of cooperation (finite sets of complementary interactions) together with a criteria to select among the possible groups of cooperations allow to specify the structured systems (i.e., systems built by several subsystems, simple or in turn structured) in an abstract formal way, again by determining a conditional specification defining an appropriate lts; however, for lack of room we cannot present the details and the corresponding visual notation here (see [20, 4, 7]).

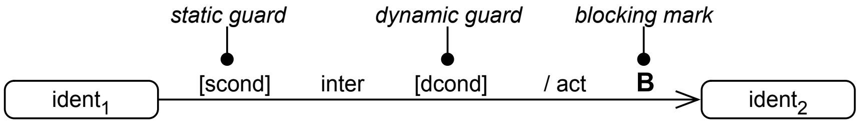
Thus, the interaction charts are a rigorous/well-founded (i.e., based on a formal foundation) notation that is extremely flexible and powerful, since a very large class of systems may be specified/modelled using it, including almost any relevant case, as shown by the many applications made in the years, from Ada to OO systems. However, due to their extreme abstraction and generality interaction charts have less nice aspects, which may prevent their use in the current software development practice. Indeed,

– the lts modelling a simple system determined by an interaction chart may have infinite transitions leaving a state;

– the sets of transitions of the subsystems generating a transition of a structured system can be determined only by considering all the subsystems together and all their possible transitions.

As a consequence, it is very hard to develop software tools to support the use of interaction charts, such as a code generator. Thus, here we propose a less general version of interaction charts, which we call *executable*, characterized by the fact that they a have an executable (operational) semantics, similar to those of Petri nets and Harel's statecharts.

The executable interaction charts are based on the record variant (see Sect. 3.2) and must have an initial state. Syntactically only the form of the transitions is changed. A transition of an executable interaction chart has the form



where scond and dcond are conjunctions of positive atoms, inter is a term built by a label generator, $F_1, \ldots, F_n$ are the field names, act $= F_1{=}t_1; \ldots;$ $F_n{=}t_n$, $\mathit{FreeVars}(\mathsf{scond}) \subseteq \{F_1, \ldots, F_n\}$, $\mathit{FreeVars}(\mathsf{dcond}) \subseteq \{F_1, \ldots, F_n\} \cup$ $\mathit{FreeVars}(\mathsf{inter})$, $\mathit{FreeVars}(\mathsf{act}) \subseteq$ $\{F_1, \ldots, F_n\} \cup \mathit{FreeVars}(\mathsf{inter})$. **B**, whenever present, denotes that the transition is blocking. As before, he null field updates of the form F = F will be omitted, as well as the guards when they are equivalent to true.

The operational semantics of an executable interaction chart is described below. The system goes on performing a basic-execution-step after another, where a *basic-execution-step* is defined in Fig. 3, where we write T.scond, T.inter, T.dcond, ... to denote the various parts of a transition T. Recall that at any time exactly one state is active (at the beginning the initial state is active).

```
(1) Let ET be the set of the transitions starting from the active state whose static
    guard holds;   if ET = ∅ then stop;
(2) let ETLIST be the list of the elements of ET in some order;
(3) if ETLIST is empty then go to (2);
    T = first(ETLIST); ETLIST = dropFirst(ETLIST);
(4) "attempt to execute T.inter";
    %%it can either fail or be successful returning a list of values VL instantiating the
    %%free variables of T.inter
    if it fails
    then
      if T is blocking then go to (4) else go to (3);
    else
      if T.dcond[VL/FreeVars(T.inter)] does not hold
      then
        go to (3)
      else
        execute T.act[VL/FreeVars(T.inter)];
        T.target becomes active and T.source, if different from T.target, becomes
        inactive; stop
```

Fig. 3. The basic-execution-step

In Fig. 3 we have a generic schema since step (4) "attempt to execute T.inter" must be defined case by case, i.e., the effect/meaning of performing an interaction must be defined, and it is not possible to simply say "there are other subsystems which have chosen to perform the transitions needed to build a cooperation". At this point we have two choices:

- to fix the interactions (e.g., reading and writing a buffer, sending and receiving messages along a channel, sending and receiving messages in a broadcasting way, ...), and thus the executable interaction charts are a unique notation;
- to propose a general schema for defining the meaning of executing a given set of interactions, and thus the executable interaction charts are a family of notations differing for the used interactions.

In this paper, we follow the second choice, and present it using, as example, the particular case of executable interaction charts, where subsystems communicate by sending and receiving asynchronous signals.

Syntactically the interactions are defined by a set of generators, as in Sect. 3. In this case we have two generators:
send(SysIdent,SignalName,ValueList) and rec(SysIdent,SignalName,ValueList).

For what concerns the semantics, we first define the cooperations among such interactions. In this case, send(si,n,vals) and rec(si',n',vals') form a cooperation whenever the arguments are identically, and these are all the possible cooperations.

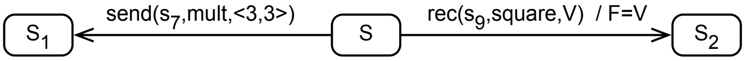
Then, we introduce some *abstract buffers* that will be accessed by the subsystems by reading or writing information about their possibilities/willingness to perform some interactions. Thus, the attempt to perform an interaction will correspond to access one of these buffers, and depending on its content it can result in the buffer communicating either the failure or the success (together with the values needed to instantiate the free variables).

In this case, for each subsystem there is a buffer containing the set of signals received by it and not yet consumed. To attempt executing send(si,n,vals) consists in adding to the buffer of si the sent signal <n,vals>, and thus it will never fail; whereas to attempt executing rec(si,n,vars) by a subsystem with identity si consists in seeing whether its own buffer contains a signal having form <n,vals>, if the answer is positive the attempt is successful, the values vals are returned and <n,vals> is deleted from the buffer, otherwise the attempt fails.

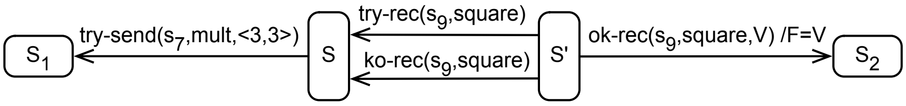
Formally, the semantics of an executable interaction chart is given by transforming it into an equivalent normal one (presented in Sect. 3.2). Precisely, a specification of a structured system where the subsystems are modelled by executable interaction charts is transformed into an equivalent specification where the subsystems are modelled by normal interaction charts.

Let SP be a specification of a structured system whose n subsystems are specified respectively by SP_1, ..., SP_n. The specification SPeq equivalent to SP is defined in the following way. The specifications SP_1, ..., SP_n using executable interaction charts are transformed in a standard way into specifications using normal interaction charts, say $\mathsf{SP'}_1$, ..., $\mathsf{SP'}_n$. The added buffers are defined by specifications of simple systems using normal interaction charts, say B_1, ..., B_k. $\mathsf{SP'}_1$, ..., $\mathsf{SP'}_n$ and B_1, ..., B_k are the specifications of the subsystems of SPeq, whereas all its non trivial cooperations are defined in a standard way, and have as participants one buffer and one original subsystem.

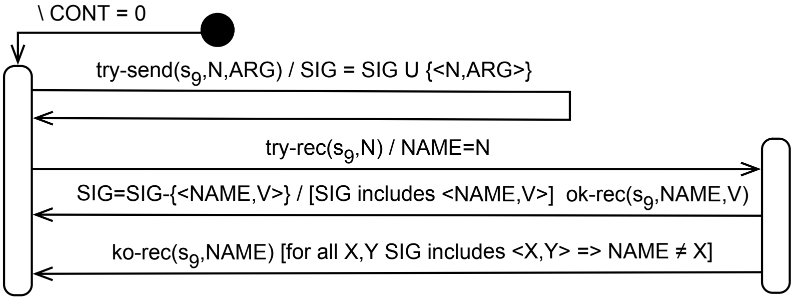
In the example, we show on a fragment of executable interaction chart using the rec/send interactions how to transform it into a normal one interacting with the buffers.



is transformed into



whereas the buffer for a subsystem identified by s_9 is modelled by the following interaction chart


\ CONT = 0
try-send(s9,N,ARG) / SIG = SIG U {<N,ARG>}
try-rec(s9,N) / NAME=N
SIG=SIG-{<NAME,V>} / [SIG includes <NAME,V>] ok-rec(s9,NAME,V)
ko-rec(s9,NAME) [for all X,Y SIG includes <X,Y> => NAME ≠ X]

The cooperations between the subsystems and the buffers are just pairs of identical interactions (e.g., <try-inter(...),try-inter(...)>).

Notice that there is not a unique way to define the buffers realizing the cooperations, not even in this simple case of asynchronous signals exchange (in our example, we could have organized the buffer as a list instead of as a set).

We can summarize the tasks for defining a variant of executable interaction charts as follows:

(a) fix which are the interactions used by the variant, by giving their generators and defining the types of their arguments,
(b) fix which are the cooperations among them,
(c) define the buffers supporting the above cooperations. The possible interactions of these buffers are try-inter(...), ok-inter(...), ko-inter(...), where inter is one of the interaction generators defined at (a). Define also their behaviour by means of a normal interaction chart.

The point (3) of the definition of the basic execution step given in Fig. 3, concerning the choice of one among the various transitions, may be made less casual by offering the possibility to control it by decorating the transitions leaving a state with priorities, just integer numbers. Then it is sufficient to replace line (3) of the definition of the basic-execution-step in Fig. 3 by

(3') let ETLIST be the list of the elements of ET ordered with respect to their priorities first those with the higher one (if several transitions have the same priority they are ordered in a casual way).

Then, a transition without priority stands for a transition with priority 0; and a transition decorated by else / act stands for a transition decorated by null / act i, where i is a number lower than the priorities of all the other transitions leaving the source state.

5 Extending UML with Interaction Machines

We propose to extend UML by adding a UML-like version of executable interaction charts that we call *interaction machines* .

UML 2.0 (but versions 1.. . . are quite similar) offers three main ways to model the behaviour:
– (behaviour and protocol) state machines, or state charts or state diagrams, showing the reactive behaviour of objects,
– sequence/communication/interaction[6] overview diagrams showing sets of sequences of events happening among a group of objects,
– and activity diagrams showing the control and dataflow aspect of the behaviour.
Thus, there is no way to present the interactive aspects of an object in isolation; such aspects may be shown only by scenarios where its interactions are performed with a selected set of partners. As a consequence, to depict the behaviour in isolation of a proactive object, i.e., one which does not simply react to events, but instead mainly triggers events to which other objects will react, we can use only a state machine, which will be not very informing and readable, since it will have very few transitions with heavy decorations, mainly with huge activity part[7]. For these reasons, incorporating into UML the interaction charts may be seen as a real extension adding more notational power.

5.1 Interaction Machines

We define the interaction machines by changing as less as possible the definition of behaviour state machines[8] of UML 2.0 [15]. Here for lack of room we just show

[6] Note that in the UML world [15] the term interaction has a meaning different from the one used in this paper, a UML interaction is a set of sequences of event occurrences among some objects.

[7] To overcome this problem UML 2.0 offers a very limited possibility to visually depict in a state machine some action either by enclosing it in a box, or, only for the send signal action, by a convex pentagon.

[8] Note, that it is possible to define also protocol interaction machines, since the distinction between behaviour and protocol state machine is orthogonal with respect to depicting interactions or reactions.

how to define the basic form of the interaction machines (but there is no problem to incorporate the other more complex constructs of the state machines).

Recall that the abstract syntax of UML is given by means of an object-oriented description, a class diagrams, called *metamodel*, whose classes correspond to the abstract syntactic categories, presented inside [15]. At the metamodel level, the interaction machines may be added as a new subclass of the metaclass Behaviour, defined using the existing metaclasses whenever possible, see Fig. 4.

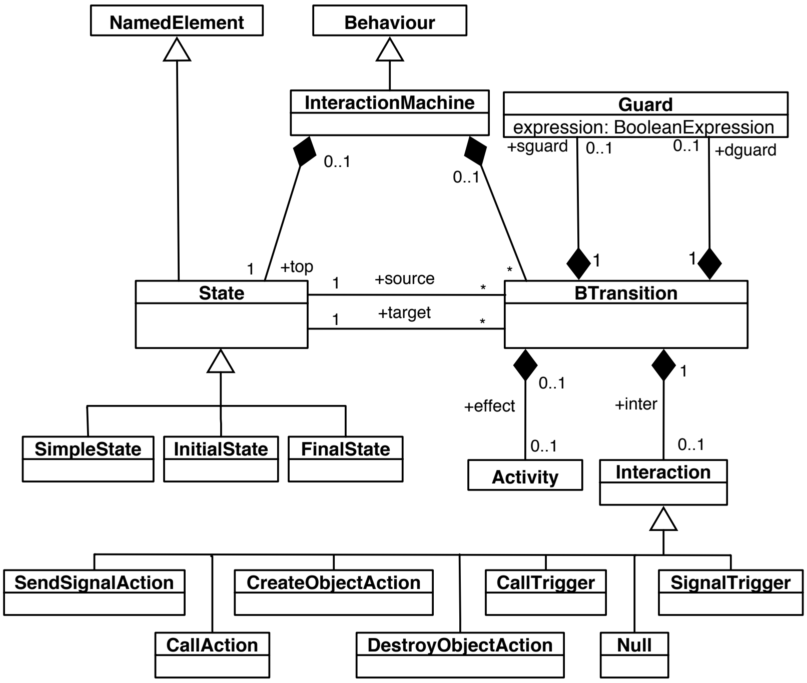
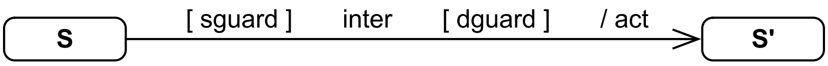


Fig. 4. The metamodel definition of interaction machines

Here we assume that the context of an interaction machines must be an active class.

Interaction machines are defined as the executable interaction charts, but the transition decorations are expressed using UML ingredients; precisely, the interactions are determined by the UML actions and events, and the guards and the activities are expressed by using the means offered by the UML. The generic form of the transitions of the interaction machines is



where

- sguard and dguard are boolean expressions (written using OCL[9]).
- inter is defined by the fragment of the UML metamodel in Fig. 4. It may be either a simple action requiring an interaction with some other object (here we only consider: operation call, signal sending, creation and destruction of objects), or an event that is the result of an interaction by some other object (call and signal trigger) or null that is no interaction with any other object, i.e., an activity purely internal to the context object.
- act, the *activity*, is an UML action.

In this case the "Constraints" defining the well-formed constructs are quite important and are as follows

- the evaluation of sguard and dguard cannot have any side effect, as already required for state machines [15];
- the evaluation of sguard, dguard and the execution of act must be possible without accessing anything outside the context object. For example, this means that, differently from UML state machines, a call of an operation of another object cannot appear in the activity part;
- the evaluation of any expression appearing in an interaction of the kind action must be possible without accessing anything outside the context object;
- only interactions of the form call and signal event may have formal parameters that may appear in dguard and act;
- all the attributes of the context active class are visible only inside the class itself and the interaction machine; thus no other object may access or modify them indirectly, except explicitly calling operations of the object (sending signals to it).

Notice that when we speak of objects, obviously we do not consider instances of UML datatypes.

The intuitive meaning of a transition of an interaction machine is that the object may perform (whenever possible) some interaction with some other object possibly followed by some *local* activity when some guard conditions are satisfied.

The precise meaning of an interaction machine is given by specializing the basic-execution-step of Fig. 3 to this particular case, that is essentially to define what means to attempt executing the particular interactions used here, as shown below.

CallAction execute the call action; here we consider only asynchronous calls, thus it cannot fail;

CallTrigger if there is matching call in the event pool, then take it instantiating the parameters, otherwise fail;

SendAction execute the send action; sending signal is always asynchronous, thus it cannot fail;

SignalTrigger if there is matching signal in the event pool, then take it instantiating the parameters, otherwise fail;

[9] The Object Constraint Language to specify constraints and other expressions appearing in UML models, see [14]

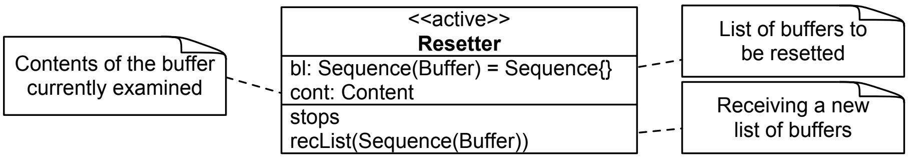
null do nothing, clearly it cannot fail
CreateObjectAction execute the create action; it cannot fail;
DestroyObjectAction execute the destroy action, it cannot fail.

In this case, the abstract buffer supporting the cooperations are just the event pools associated with any UML object [15]. Notice, that the basic-execution-step in this case is a generalization of the run-to-completion-step used to describe the semantics of the state machines in [15], Sect. 15.3.12.

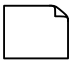
5.2 An Example: the distributed buffer resetter

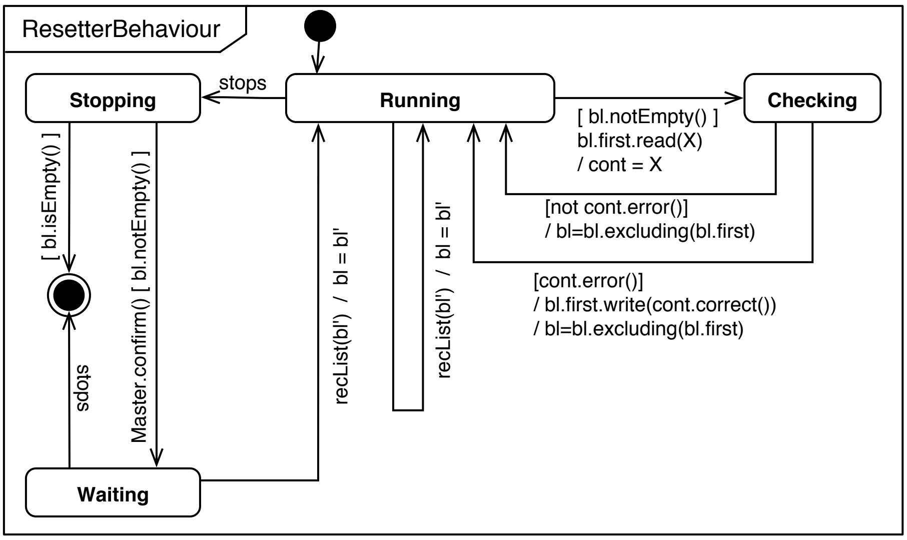
In this section we present a simple example of the use of the interaction machines to model a nonpurely-reactive active object, the *Distributed Buffer Resetter*. This example is quite paradigmatic of autonomous agents doing monitoring and maintenance over distributed systems or Internet. The resetter accesses some buffers one after another following some given ordering, and resets each buffer if its contents is "wrong". At each moment, the resetter can receive from some manager the list of the buffers to reset, or it can be stopped.

Using UML we model the buffer resetter using an active class with an associated interaction machine.



<<active>>
Resetter
bl: Sequence(Buffer) = Sequence{}
cont: Content
stops
recList(Sequence(Buffer))
Contents of the buffer currently examined
List of buffers to be resetted
Receiving a new list of buffers

The class has two attributes cont and bl, and two operations stops and recList. The text enclosed by  is an UML comment.



The resetter in the running state has three possible moves:

- when bl is not empty, it may access the first buffer of the list getting its content; if such content is an error (checked by the operation error) it resets such buffer by correcting its content;
- it may receive a new list of buffers to be reset by accepting a call of its operation recList;
- it may receive a request to stop (by a call of its operation stops); in such cases it passes in the state **stopping**. If bl is empty, it terminates by a transition into the final state, whose atomic interaction is the null one, otherwise it asks for a confirmation to its master by calling its operation confirm. Then, if it receives it as a new call of the stops operation, it terminates; if instead it receives a new list of buffers, it goes on to work again.

6 Conclusions

The main message of this paper is to witness the evolution from purely formal techniques to visual notations that are more friendly for the user, but still rooted in "theoretical and conceptual work", as advocated by Ehrig and Mahr [10] about ten years ago.

On the technical side, our journey was consisting of the following intermediate steps:

- a description of the behaviour of systems using CASL conditional specifications, with initial semantics, of labelled transition systems;
- their visual presentations by means of interaction charts;
- their specialization as executable interaction charts, i.e., interaction charts with a special operational semantics targeted at an easier passage to the code;
- finally a proposal of an extension of UML by introducing the interaction machines, which are essentially the UML version of the executable interaction charts.

The technical motivation of the introduction of the interaction machines in UML is their ability to represent both reactive and proactive behaviour of an object, where by proactive we mean autonomous behaviour of the kind required for example when modelling autonomous agents.

Note the difference with the approach taken in the UML [15], where interaction means a set of sequences of event occurrences among some objects. In other words in UML there is no provision to represent the interactions of an object in isolation.

The approach we have shown here allows to use visual notations where the formalities are completely hidden, though being amenable to a precise semantics; this is the essential meaning of the strategy that we call "well-founded methods" [5].

Note also that in the case that we have presented in this paper we have not given a semantics to an existing practical notation, but we have gone the opposite way: the interaction charts, with their ability to express autonomous behaviour, have been suggested by a purely formal specification technique (Casl specifications of labelled transition systems). This is one of the modalities appearing in what we have called “virtuous cycle” of the interaction between foundational and engineering work [5].

References

1. E. Astesiano, A. Giovini, F. Mazzanti, G. Reggio, and E. Zucca. The Ada Challenge for New Formal Semantic Techniques. In *Ada: Managing the Transition, Proc. of the Ada-Europe International Conference, Edimburgh, 1986*, pages 239–248. University Press, Cambridge, 1986.
2. E. Astesiano and G. Reggio. Formalism and Method. In M. Bidoit and M. Dauchet, editors, *Proc. TAPSOFT '97*, number 1214 in Lecture Notes in Computer Science, pages 93–114. Springer Verlag, Berlin, 1997.
3. E. Astesiano and G. Reggio. Formalism and Method. *T.C.S.*, 236(1,2):3–34, 2000.
4. E. Astesiano and G. Reggio. Labelled Transition Logic: An Outline. *Acta Informatica*, 37(11-12):831–879, 2001.
5. E. Astesiano, G. Reggio, and M. Cerioli. From Formal Techniques to Well-Founded Software Development Methods. In *Formal Methods at the Crossroads: From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002. Revised Papers.*, number 2757 in Lecture Notes in Computer Science, pages 132 – 150. Springer Verlag, Berlin, 2003.
6. M. Bidoit and P.D. Mosses. *CASL User Manual, Introduction to Using the Common Algebraic Specification Language.* Number 2900 in Lecture Notes in Computer Science. Springer-Verlag, 2004.
7. C. Choppy and G. Reggio. Towards a Formally Grounded Software Development Method. Technical Report DISI–TR–03–35, DISI, Università di Genova, Italy, 2003. Available at `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio03a.pdf`.
8. E. Coscia and G. Reggio. JTN: A Java-targeted Graphic Formal Notation for Reactive and Concurrent Systems. In Finance J.-P., editor, *Proc. FASE 99*, number 1577 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1999.
9. G. Costa and G. Reggio. Specification of Abstract Dynamic Data Types: A Temporal Logic Approach. *T.C.S.*, 173(2):513–554, 1997.
10. H. Ehrig and B. Mahr. A Decade of TAPSOFT: Aspects of Progress and Prospects in Theory and Practice of Software Development. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *Proc. of TAPSOFT '95*, number 915 in Lecture Notes in Computer Science, pages 3–24. Springer Verlag, Berlin, 1995.
11. R. Milner. *A Calculus of Communicating Systems.* Number 92 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1980.
12. P.D. Mosses, editor. *CASL Reference Manual, The Complete Documentation of the Common Algebraic Specification Language.* Number 2960 in Lecture Notes in Computer Science. Springer-Verlag, 2004.
13. OMG. *UML Specification 1.3*, 2000. Available at `http://www.omg.org/docs/formal/00-03-01.pdf`.
14. OMG. *UML 2.0 OCL Specification*, 2003.
15. OMG. *UML 2.0 Superstructure Specification*, 2003.

16. G. Plotkin. An Operational Semantics for CSP. In D. Bjorner, editor, *Proc. IFIP TC 2-Working conference: Formal description of programming concepts*, pages 199–223. North-Holland, Amsterdam, 1983.
17. G. Reggio, E. Astesiano, and C. Choppy. Casl-Ltl : A Casl Extension for Dynamic Reactive Systems Version 1.0– Summary. Technical Report DISI-TR-03-36, DISI – Università di Genova, Italy, 2003. Available at `ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAll03b.ps`.
18. G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In T. Maibaum, editor, *Proc. FASE 2000*, number 1783 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000.
19. G. Reggio, M. Cerioli, and E. Astesiano. Towards a Rigorous Semantics of UML Supporting its Multiview Approach. In H. Hussmann, editor, *Proc. FASE 2001*, number 2029 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2001.
20. G. Reggio and M. Larosa. A Graphic Notation for Formal Specifications of Dynamic Systems. In J. Fitzgerald and C.B. Jones, editors, *Proc. FME 97 - Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1997.