BULETINUL Universităţii Petrol – Gaze din Ploieşti	Vol. LVIII No. 2/2007	49 - 68	Seria Ştiinţe Economice

Cycle: UML FOR MANAGERS (VI)
State and Activity Diagrams

Liviu Dumitraşcu, Gabriel Irinel Marcu

Petroleum-Gas University of Ploieşti, Bd. Bucureşti 39, Ploieşti
e-mail: ldumitrascu@upg-ploiesti.ro, gimarcu@upg-ploiesti.ro

Abstract

The state diagrams (states-transitions) allows for the representation of they inner behavior of an object and the activity diagram allows for the specification of the a priory sequential transforming.
The present article presents the key concepts and the advanced concepts for the two types of diagrams (behavioral-dynamic) - of state and of activity.

Key words*: state diagrams, finite state automatic machine, state, initial state and final state, transition, event, message, condition, effect: action/activity, durable activity, effect, entry, exit, super-states, sub-states, self transition, internal transition, history, send, composition, concurrent regions*

State Diagrams

Modeling by Means of Automatic Machines

State (state-transition) diagrams of the unified UML language describe the inner behaviors of an object by means of a finite state automatic machine (state machine).

A finite state automatic machine mentions the frequency of the states that an object can reach during its life span as a reply to the events associated to it alongside with the reactions corresponding to these states.

This local approach of an object which describes the way in which it reacts to events taking into account its current state and the way it passes to another state is graphically represented as a state diagram.

A state diagram is, basically, a projection of the events which may appear in a finite state automatic machine (state machine).

Figure 1 presents a first example of a state diagram which displays its initial and its final state- three states and four events.

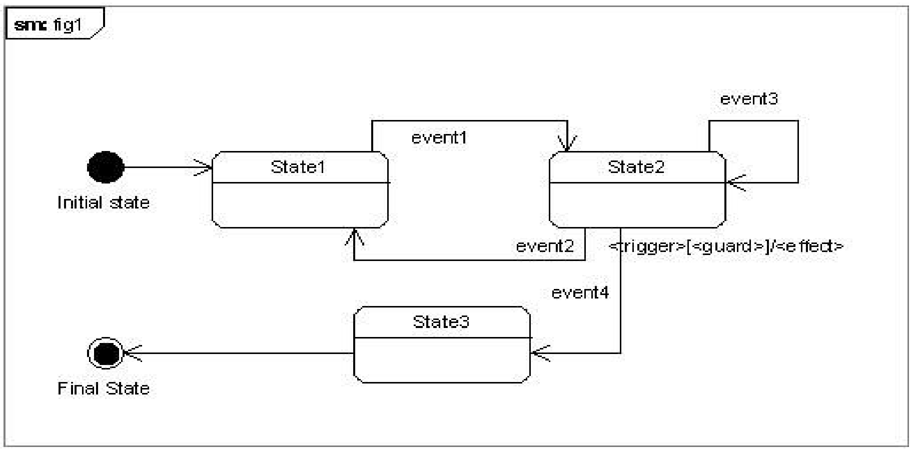
Fortunately, not all objects of a class require a state diagram representation. Usually, only the complex behavior of the objects of a class is represented by means of a state machine.

In order to find these objects with a dynamic and complex behavior, it is useful for us to ask ourselves the following questions:

- o Can the object of that class react differently to the apparition of the same event? In this case, each type of reaction characterizes a different state of the object.

- Does the class the object pertain to have to organize certain operations in a precise order? In this case, the sequential states allow for the mandatory chronological stating of the activation of objects.

Figure 1



State

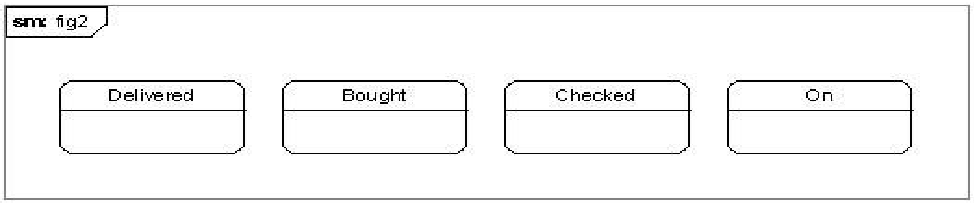
A state represents a situation during the life of an object when the object can accomplish a certain condition, perform an activity or wait for an event.

An object goes through a succession of states during its existence. A state has a finite variable duration, mainly determined by the received events.

States are represented by means of squares with rounded corners and they represent an abstract version of values and connections of an object.

Figure 2 represents the UML notation of the states: a square with rounded corners which can optionally contain the name of the state. Conventionally, the name of the state is doubly defined, centered and it starts in capital letter.

Figure 2



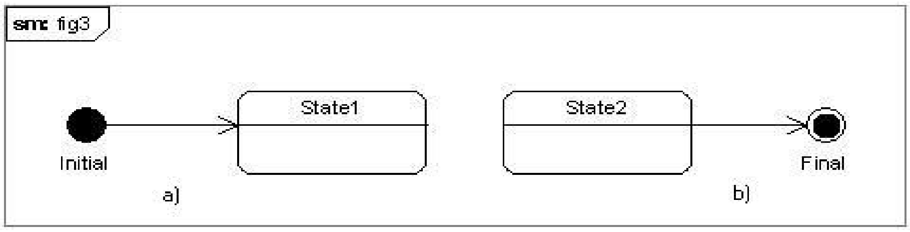
Initial and Final State

Apart from the succession of the states "name@, corresponding to the life span of a given object, state diagrams also contain two pseudo-states. The initial state of the corresponding state diagram (figure 3) and final state of the state diagram corresponding to the destruction of the instance (figure 3).

When we define states, we ignore the attributes which do not affect the behavior of the object and we regroup within the same state all combinations of values and connections that represent the same answers to events.

Connections can also have a state inasmuch as we can consider them objects (practically, states are associated to objects!).

Figure 3



Transition

A transition represents the instantaneous passing from one state to another. Example: taking a phone call (passing on from the state calling to the state connected).

An event (see next paragraph) can generate the transition of various objects. Conceptually speaking, these transitions are concrete.

As a general rule, a transition contains: a triggering event, a guarding condition, an effect and a target state.

The UML symbol of a transition is a line that connects the basic state to the target state, with the arrow pointing to the target state. The line can be made of various segments.

Figure 4

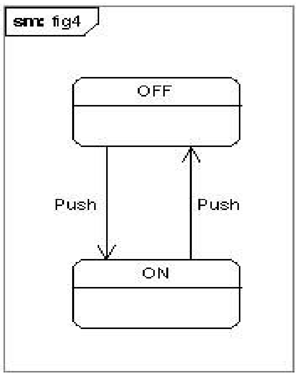


Figure 4 presents state diagram- transition for a desk lamp. The diagram starts with the initial state then it continues with an automatic transition till the desk lamp goes to the lit mode. At the first pressing of the switcher, the desk lamp goes on off mode and then on the on mode and so on.

Event

In UML, an event is the specification of an important event which can be localized in time and space. In the context of a state machine, an event is a stimulus which can trigger a transition towards a different state.

UML proposes four types of events:

- receiving a message sent by another object or by an actor. The sending of the message is usually a-synchronically represented.
- a call event of the receiving object. The call event is generally synchronically represented.

- the time event which is modeled by means of the key word after followed by an expression which represents a given time span considered from the entry into the current state.
- a change in the satisfaction of a condition (change event) which can be modeled by means of the key word then, followed by a boolean expression. The change event is produced when the condition becomes true.

Remark. An event can contain parameters that materialize the information or data flux between objects.

Message

A message is a unidirectional data transmission between two objects: the sender and the receiver.

The communication model can be synchronical or a-synchronical. The receiving of a message is an event which has to be dealt with by the receiver.

Condition

A condition (or guarding condition) is a boolean expression which must be true in order for the transition to be accomplished.

A condition can contain the attributes of the object as well as the parameters of the triggering event. Several associated transitions must have various guar conditions.

Effect: Action/Activity

A transition can specify an optional behavior by means of an object when the transition is triggered. This behavior which can be an action or a sequence of an action is called "effect" in UML 2.

A basic action can represent the actualization of an attribute, an operation act, the creation or destruction of another object as well as the sending of a signal to another object.

The activities associated to transitions are considered atomic and consequently, they cannot be interrupted. These are typical sequences of an action.

A state can contain a durable activity-do activity. This durable activity has a specific time span, can be interrupted and is associated to the states.

Figure 5

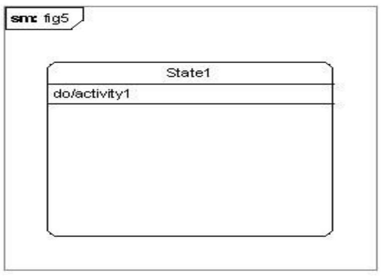
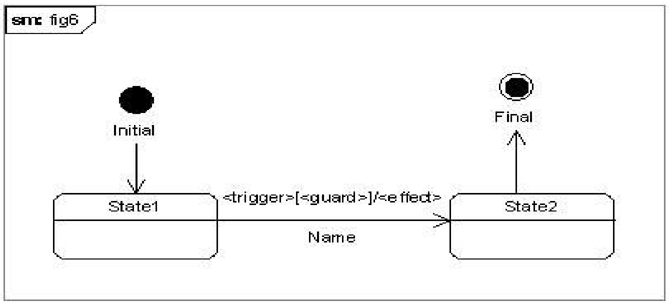


Figure 5 represents a state which contains a durable activity.

Principles to Accomplish State Diagrams

Figure 6 presents a more complete notation of a state diagram which, apart from the initial state of the object (creation) and the final state of the object (destruction) also contains a state with a durable activity and a condition with effect.

Figure 6



In order to efficiently build a state diagram, respect the principles described in [1] and presented bellow:

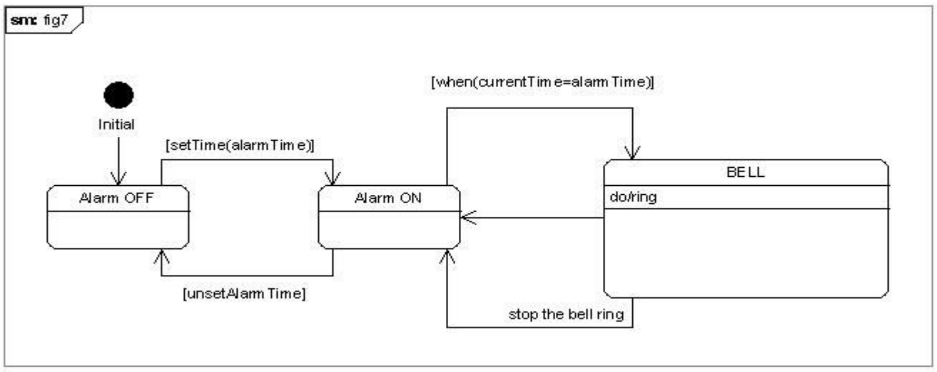
1. First represent the state sequence which describes the nominal behavior of an instance with its associated transitions.
2. Progressively add transitions which correspond to the alternative or error behaviors.
3. Add the effects corresponding to the transitions and to the activities within the states.
4. Structure the diagram in sub-states when it becomes too complex and use advanced notations: entry, exit etc.
5. Distinguish between inner events- “when” and temporal events “after” on the one hand and the ones dealing with receiving messages on the other.

Remark. An event (as transition) is conventionally instantaneous or atomic. Consequently, it is not correct to test its duration. The sole dynamic concepts in UML which benefit from the notion of duration are: state and durative activity.

6. Correctly use automatic transitions - continue or finite activity. A durable activity within a state can be:
   - o continuous (it does not stop until an event able to get the object out of the initial phase is produced);
   - o finite (can be interrupted by means of an event but it stops after a given period of time or when a condition is accomplished).

Figure 7 presents a complete state diagram for an alarm clock, diagram adapted after Pascal Roques.

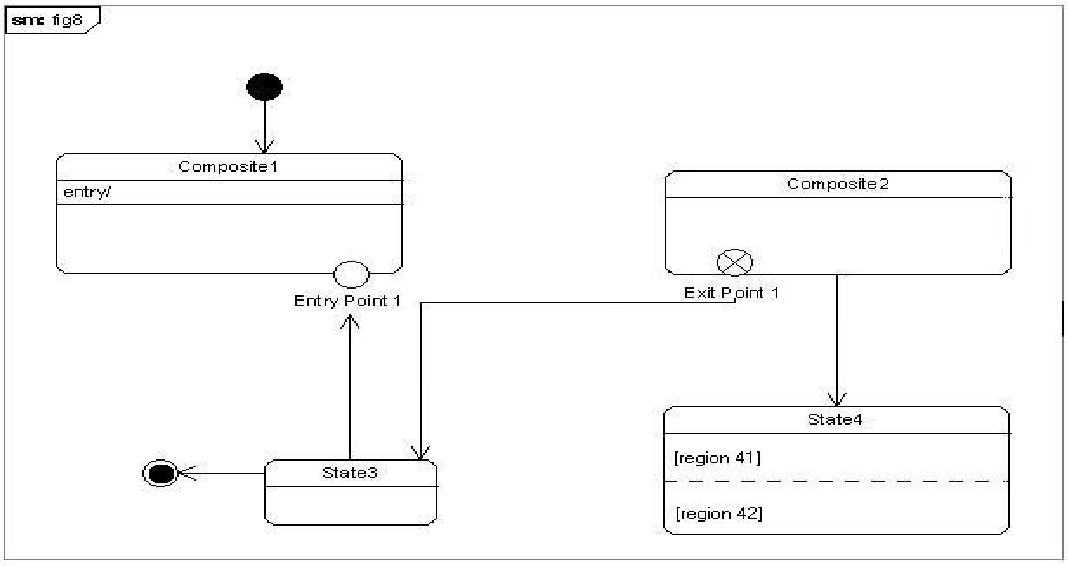
Figure 7



Remark. For this example it is enough to add a durable activity do/call to the state "bell ring" and an automatic transition when exiting this state.

7. Use the concept of composed states or super-states in order to factorize the numerous transitions triggered by the same event and which refer to the same state.
8. Make a distinction between self transition and internal transition. In the case of a self transition, the object leaves its initial sate in order to come back to it afterwards. Internal transition represents a couple (event/effect) which has no influence on the current state. Internal transition is similar to a transition which recycles a state, without a subsequent effect of the type entry, exit etc.
9. Use the pseudo-state "history" which allows for a super-state to remember the last sequential sub-state which was active before an exit transition.
10. Do not forget to represent in your state diagram the action of sending a message to another object when a transition is triggered. The syntax of such a particular action is the following: "/send.target.message".
11. Describe the inside of a complex composite state in a different diagram. In such a case, a particular notation (0-0) which signifies two bound state is represented at the bottom, on the right side of the unexpended composite state.
12. Figure 8 illustrates the shortened notation of the composite states.

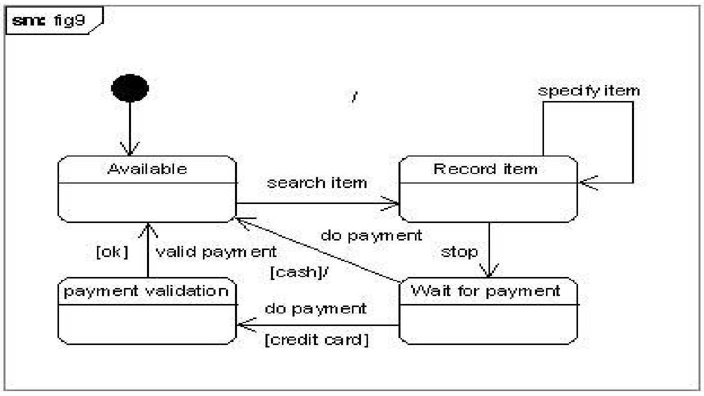
Figure 8



13. The concurrent regions allow for the expression of parallelism within an object (see figure 8).
14. Do not waste time on drawing the state diagrams which contain only tow states of the type "on/off".
15. Improve the class definition by means of the state diagrams as follows: the public operations correspond to the name of the messages issued by the actors; the private operations correspond to the name of the issued messages; the attributes correspond to the names of the remaining data, dealt with within the actions, activities or conditions.

16. The concept of operation does not make sense for a human actor. For a non-human actor the operation list represents its interface (for example, in the API sense) when it is used for the studied system.
17. Do not forcibly use all the subtleties of the state diagrams: think of those who read them!
18. Figure 9 models the simplified system for a cash machine (SCM) by means of the state diagram.

Figure 9



Activity Diagram

Basic Concepts

Activity

An activity represents a behavior which can be decomposed in several actions. Activity is a new UML concept. It is modeled by means of the activity diagram. With UML 2, activity diagrams have reached the most spectacular evolution.

Complete transformations are described by means of activities which offer and concise and clear graphics of a sequential transformation, alongside with the entire specific object oriented programming language (basic actions, curves and decisions, method demands, exception administration etc).

Activities offer a complete description of the transformations associated to behaviors in the sense of an UML interaction. They are currently used in order to label other programmes (transformations associated to messages from a sequence diagram, transitions from a state diagram etc) or to describe the way to implement an operation within a class.

Remark. Activity diagrams are relatively similar to state diagrams but their implementing is slightly different. Particularly, they offer a support for the description of concurrent mechanisms, yet this not being their only function.

An UML activity is represented by means of a square with rounded corners which contains the name of the activities placed in the right upper corner. Under this name we can mention the parameters involved within the activity. Alternatively, we can use the ways to parameter further on described in the paragraph.

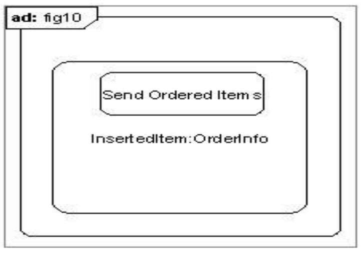
Action

An action represents the smaller performable and functional unit within an activity. Examples: VAT calculation, sending an order to a provider etc.

Actions are represented the same way as activities (a square with rounded corners). The name of the action is placed within the square.

Figure 10 presents two actions placed within an activity (the Denz establishing).

Figure 10



In order to describe the actions, UML does not specify any specific syntax. In order to grant a bigger importance to an activity diagram, we can use a pseudo-code or a syntax close to a programming language.

Remark. An activity starts by means of an initial activity mode, represented by means of a black dot and it ends by means of a final mode, represented by means of a black disk encircled with a ring.

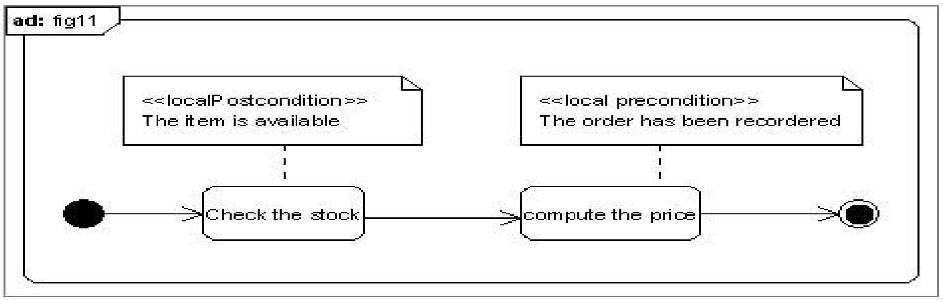
Activity Interfaces

In order to represent he flux transgressing an activity, actions are connected by means of activity interfaces.

An interface specifies the way in which the control and the data flux transgresses from an action to another. An activity interface is represented by means of a line ending in an arrow pointing towards the next action.

Figure 11 (the same as the previous figures) represents an activity diagram requiring more activity interfaces. Additionally, by means of two notes attached to the action and which contain the key words "localPrecondition" and "localPostcondition" we also mentioned post and preconditions.

Figure 11



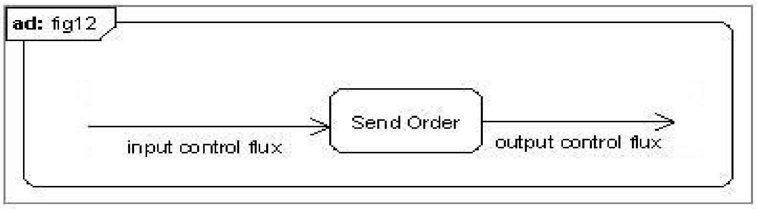
Control Flux

An activity diagram is basically an organizing scheme that illustrates the control flux (workflow) from one activity to another. It is an ideal instrument for the description of procedural logics.

A control flux explicitly represents a control that goes from one activity to the next. Practically, few moulders make the distinction between a generic activity interface and a control flux, because both share the same notation.

Control fluxes adjust actions (see figure 12).

Figure 12



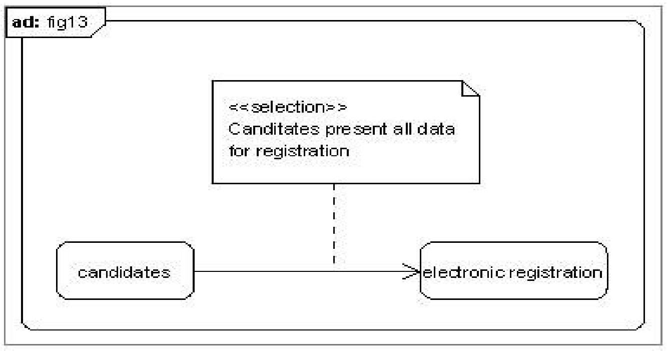
Object Flux

UML also proposes a specialised activity interface version for the description of data: the object fluxes.

The object fluxes allow for the modeling of the data sent to various receivers (multicasting), the selection and transformation of tokens. The object flux is represented the same way as the generic interface activity. UML 2 has unified the notation for the control fluxes and for the object fluxes.

Figure 13 represents an activity diagram within which objects are selected based on a selection behavior.

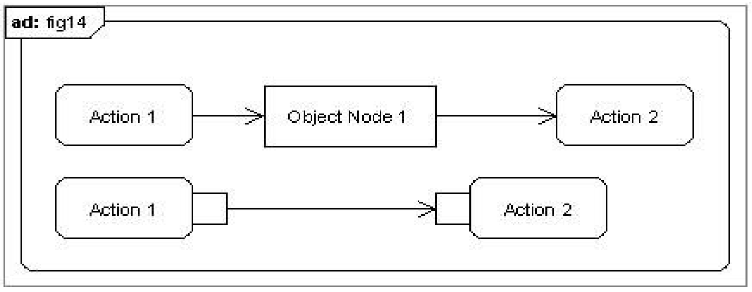
Figure 13



UML 2 proposes two equivalent notations for the object fluxes (see figure 14).

The second solution uses a bend with its origin and destination in a pin corresponding to an object flux. A pin is represented by means of a smaller square attached to the edge of an activity.

Figure 14



Tokens

Conceptually, UML models data transfers via an activity interface as a token. A token can represent real data, an object or the control focus. Incomes and outcomes of an action are represented as tokens.

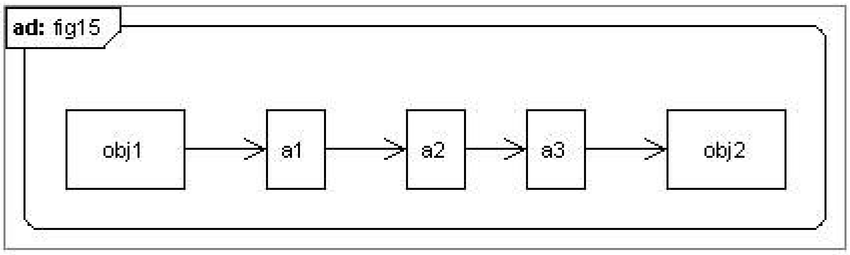
Activity Nodes

An activity can be modeled by means of various types of nodes: parametering nodes and control nodes.

Parametering Nodes

By means of parametering nodes we can represent incoming parameters for a given activity or the outcoming generated elements. The parametering mode is represented as a rectangle situated on the interface of this activity. This node can get a name or a description, mentioned inside the rectangle. Input parametering nodes sever action interfaces from the first action. Output parametering nodes sever action interfaces from the last action (see figure 15).

Figure 15

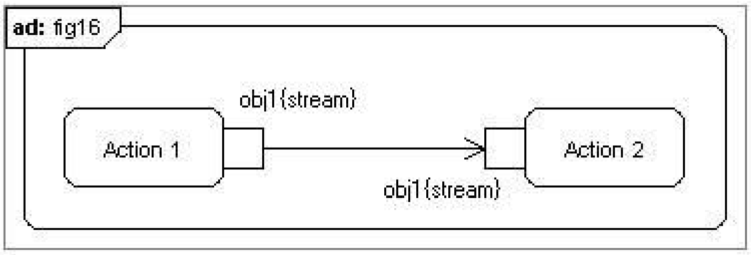


Object Nodes

Object nodes allow for the representation of the complex data that pass thorough an activity diagram. An object node represents an instance of a particular class under a given state. An object node is represented as a rectangle within which we mention the name of the node. The name of the node constitutes the represented type of data.

Remark. Object nodes can be eventually used in streaming. This means that they are constantly used and created (figure 16).

Figure 16



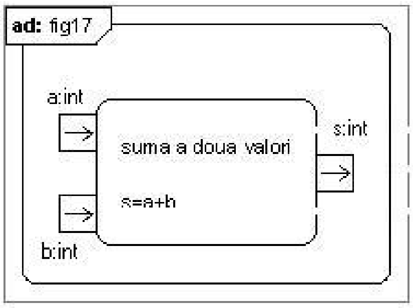
Pins

In UML 2, apart from the rectangular notation, object nodes can be modeled as pins, small squares applied on the border of the action.

If no activity interface enters or exits an action, one can indicate if a pin is an entry/exit pin, by placing a small arrow inside the rectangle representing the pin.

Figure 17 represents an action (added values) which presents entry and exit pins.

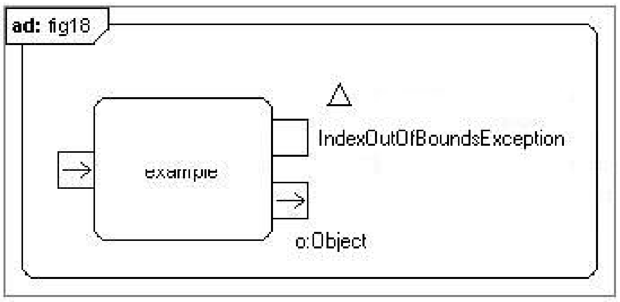
Figure 17



If the exit from an action is connected to an exception (error condition), this can be represented by means of an exception pin, drawing a small rectangle above it.

Figure 18 represents an exception pin for the situation when the method ElementAt pertaining to the class vector belonging to the standard Java library sends back the vector placed in the position sent by index, as an argument, but it can fail if this index is negative or it is bigger than the vector.

Figure 18



Control Nodes

Control nodes allow for the representation of decisions, of concurrent actions or of a synchronizing action.

Initial Nodes

An initial node constitutes the initial point of an activity and it can be represented as a dot.

Decisional Nodes and Nodes of Confluence

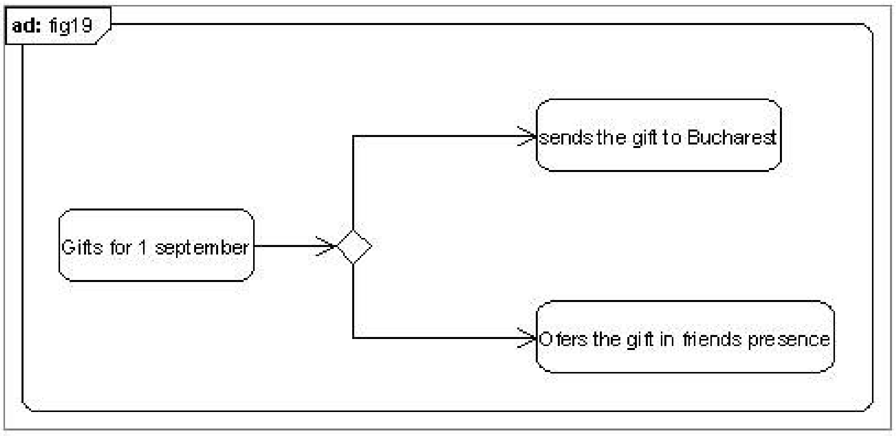
A decisional node is a control node which allows for the choosing of an exit flux taking into account a boolean expression. Each decisional node possesses an input action interface and several output action interfaces. When a datum reaches a decisional node, a single output action interface is selected, mainly the one the datum is oriented to.

A decisional node allows for the selection of an input action interface by means of a guarding condition for each interface. A guarding condition is a boolean expression which allows for the performing of a test for a value accessible by activity.

The decisional node is represented as a heart-shaped figure towards which various values are targeted and which generate other fluxes. We have to mention a guarding condition by means of an expression placed between brackets, near the activity interface.

Figure 19 represents an activity diagram containing a decisional node.

Figure 19


ad: fig19
sends the gift to Bucharest
Gifts for 1 september
Ofers the gift in friends presence

As it can be noticed, there is no specific guarantee as to the order necessary to perform guarding conditions (see figure 19).

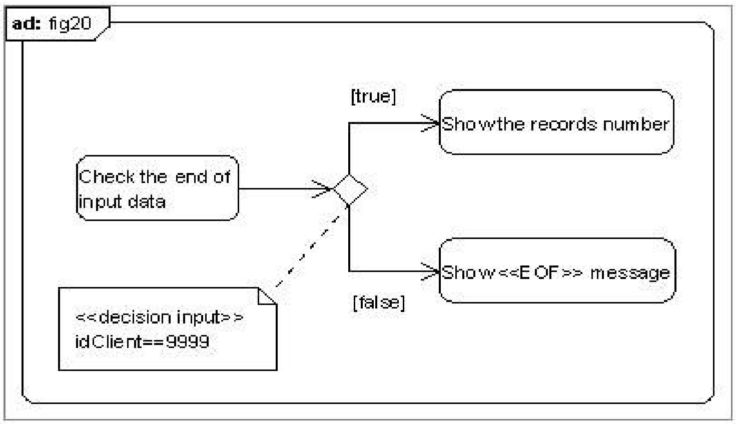
The only certainty is that a decisional node will open on the selection of a single output action interface.

We can specify a functionality to be executed when a datum arrives in a decisional node. This functionality is called decisional entry behavior.

A decisional entry behavior is represented by means of a note containing the key word "decisonInput".

Figure 20 represents an activity diagram which contains a decisional node mentioning an entry behavior.

Figure 20

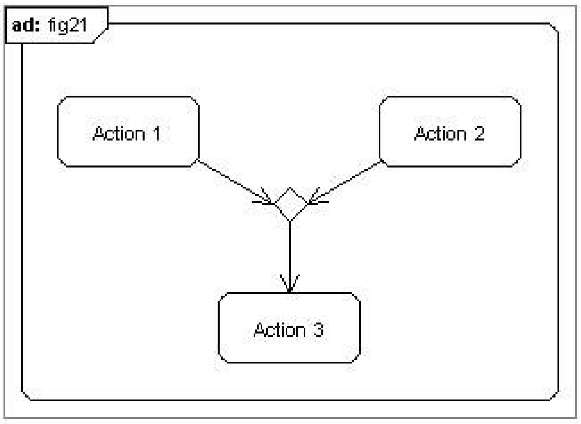


A node of confluence is the opposite version of a decisional node. It allows for the reunifying of various fluxes within the same exit flux. A node of confluence contains more entry action interfaces and a single output action interface.

A node of confluence is represented by means of an empty heart-shaped figure, the same as the decisional node, with the difference that there are more input action interfaces and a single output action interface.

Figure 21 represents an activity diagram containing a node of confluence.

Figure 21



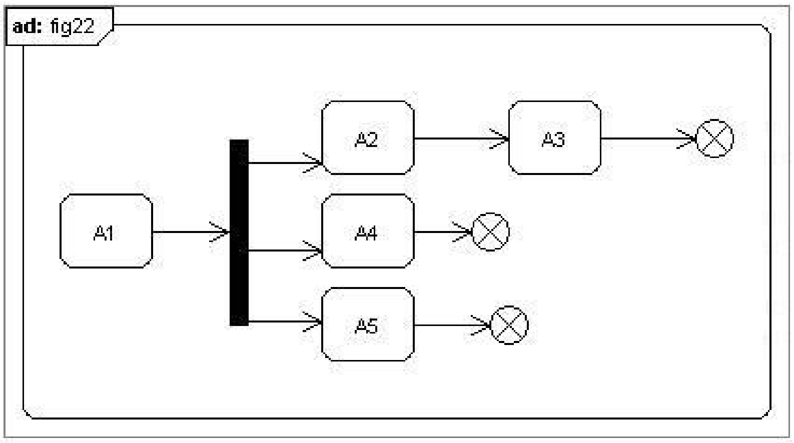
Fork-type Nodes and Join-type Nodes

A fork-type node allows for the severing of the current flux, within an activity and sharing more concurrent fluxes. Such a node possesses an input activity interface and several output action interfaces. When a datum reaches a fork type nod, it is duplicated in order to be sent to each of the output action interfaces.

A fork-type node is represented by means of a vertical thickened line on which we place an input activity interface and from which spring several output action interfaces.

Figure 22 represents an activity diagram which contains a fork-type node. All the branches of this activity diagram are concurrently performed. Each branch (control flux) is represented by means of X which states the fact that this particular control flux is finished but the overall activity can still be performed on a different control flux.

Figure 22



A joint type node is the opposite of the fork-type node. It synchronises several fluxes of an activity and it gathers them back within the same performing flux.

A joint type node possesses several input activity interfaces and a single output action interface.

A joint type node is represented by means of a vertical thickened line which possesses several input activity interfaces and a single output action interface.

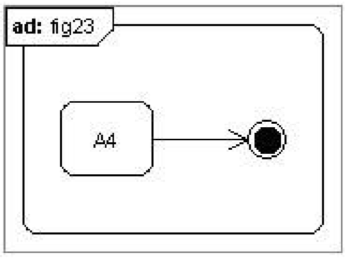
Figure … represents an activity diagram which contains a joint-type node. Each branch (control flux) is to be performed until reaching the node. The activity waits for all branches to reach this node in order to trigger the action.

Final Nodes

In an activity diagram we use two types of final nodes: activity final nodes and flux control final nodes.

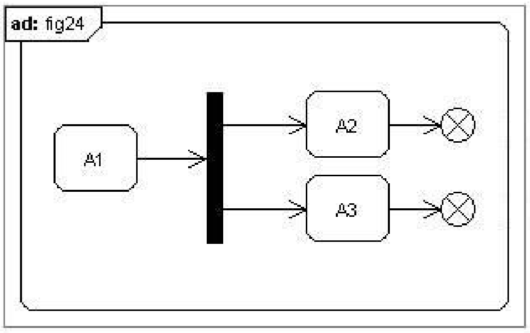
A final node for a given activity is represented by means of an encircled black dot (figure 23).

Figure 23



Flux control final nodes finish a partial execution path materializing it as an activity diagram but it does not finish the entire activity. Flux control final nodes are used when an activity flux reaches a splitting node. A flux control final node is represented by means of a circle circumscribing an "X" (figure 24).

Figure 24



Advanced Concepts

The main role of an activity diagram is constituted by the a priori description of sequential transformations: an activity starts only after the anterior one is finished.

Yet, it is possible to describe concurrent mechanisms by means of activity diagrams. Competition administration adds to activity diagrams another two new graphic elements:

- join and fork type synchronizing bars;
- final flow type control nodes.

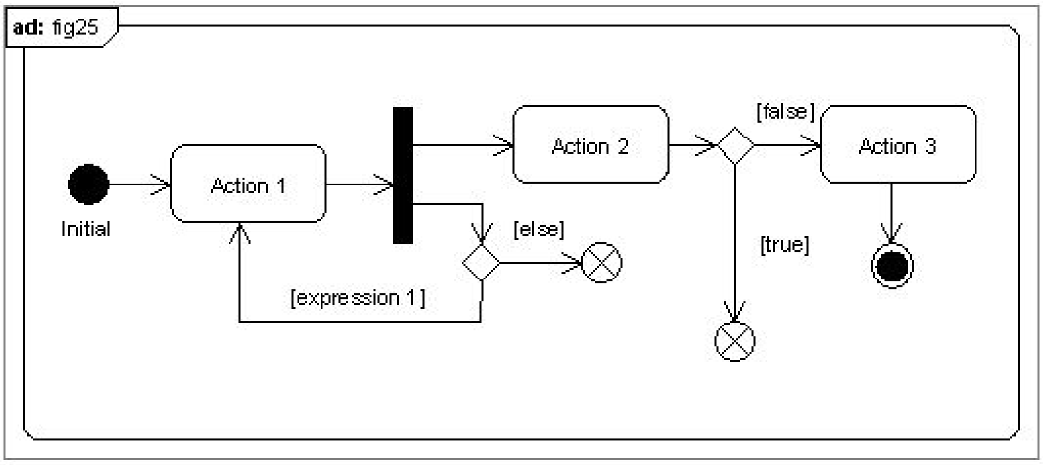
Join and fork type synchronizing bars are represented by means of a short thick black line. More transitions can have as source or target a synchronizing bar.

When the synchronizing bar has more exit transitions, we say that we deal with a fork type transaction which corresponds to a duplication of the control flux into more independent fluxes;

when the control bar is touched, a control token is generated on each exit bend. When the synchronizing bar has more entry transitions, we say that we have to deal with a join type transition which corresponds to a joining between control fluxes; the transition cannot be realized if a token is not present on each of its entries.

For a bigger accessibility, it is possible to join the join and fork type bars and thus to have on the same bar more entry and exit transition bars.

Figure 25



Final flow type control nodes are represented by means of a circle enclosing an X and it corresponds to the end of the life span of a token. A control token which reaches such a node is destroyed. The other control fluxes are not affected. Contrary, if a control flux reaches a final node; all the other control fluxes of the activity are interrupted and destroyed.

Figure 25 presents an example which illustrates a fork type synchronizing bar and two final flow control nodes.

Administration of Exceptions

The object oriented programming presents an important concept. They allow for the interruption of a transformation when a situation different from the normal transformation may occur. This ensures a self administration of errors which may occur during a transformation. Example: the zero division administration within an arithmetical operation.

An exception which is not dealt with at any level corresponds to a programming error. The majority of the object oriented programming languages, it induces the end of process. In UML the behavior is not mentioned but the model is consolidated as incomplete.

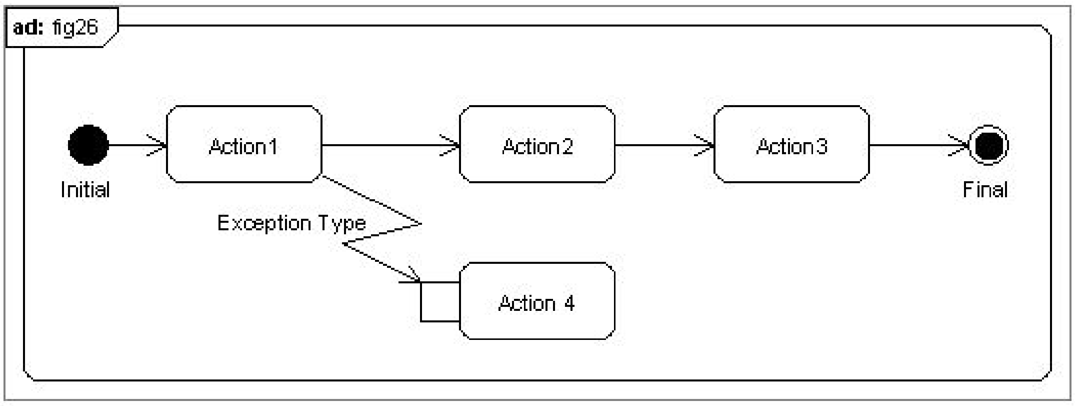
Activity diagrams for UML 2 allow for the modeling of the exception administration. An exception describes an error which can be produced while performing an activity. We can say that an exception is joined by error source and that it is caught when transformed.

We should mention the fact that an action can deal with an exception becoming an exception administration device. This device defines the exception type as well as a behavior able to catch this particular exception.

Such a device is represented as a regular node which contains a little square on its border. From the protected node, we shall draw an arrow as a lightening. In the end we shall attribute a name to this arrow able to represent the type of exception dealt with by the administrating device.

Figure 26 presents an example of exception administration.

Figure 26



Remarks:

- if an exception is produced during the execution of an action, the execution is interrupted and the action does not posses any exit datum.
- if the action possesses an error administrator, it is performed taking into account the information carried by the exception.

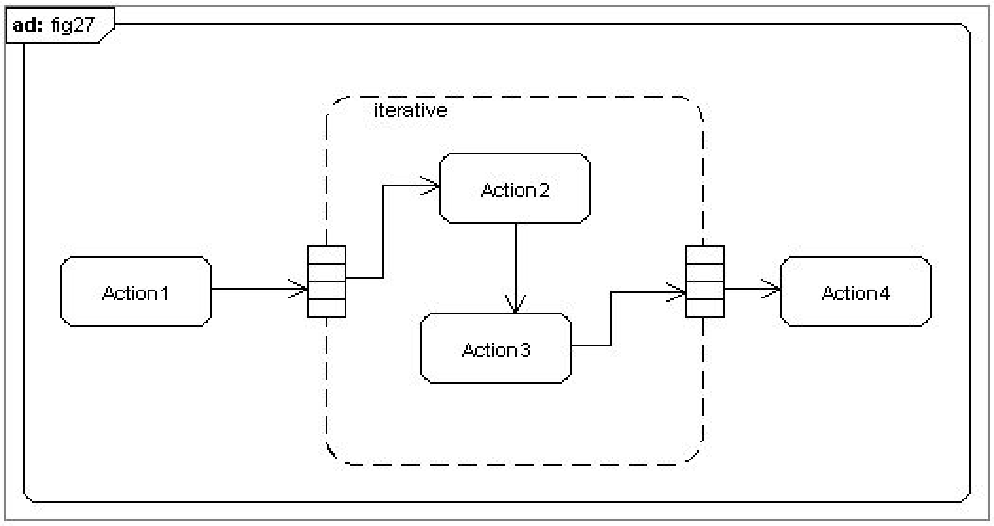
Extension Regions

An extension region is a structured activity node performed once for each element of an entry collection.

An extension area is graphically represented by means of a rectangle with discontinuous contour and with rounded edges. This rectangle has to contain all the actions that are performed for all the entry elements. A collection can be implemented by means of a chart or a list. The type of objects contained must be known.

The entries and exists of the activity are materialised on the border of the rectangle and they can be divided in more compartments.

Figure 27



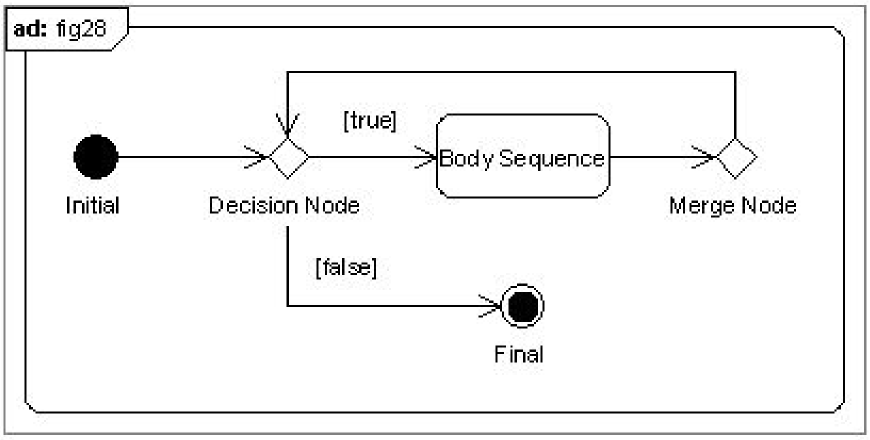
Remarks:

- The four squares placed on an extension area in order to represent entry and exit data are not obligatorily connected to the number of provided objects (they are meant to mention if a group of elements is provided).
- The actions contained within the extension region are performed for each provided element and if there is no error an exit element is provided for each element.
- An arrow allows for the identification of the origin of data and another arrow allows for the identification oft eh first action which is to be performed within the extension area. The second group of for small contiguous squares allows for the representation of exit data, an arrow identifying the last performed action within the extension area and another arrow has identified the action to be performed in that area.
- An extension area can be of one of the following three types identified by means of a key word, placed in the left corner of the extension areas: parallel, iterative and stream.
- Parallel mentions the internal activity execution mode.
- Iterative mentions the sequential execution mode.
- Stream mentions the continuous execution mode as a data flux for the internal activity which has to be adopted by the flux workers. The stream type allows for the stronger control of the action performing parallelism.

Loops

UML 2 defines a construction which allows for the modeling of loops within activity diagrams a loop node. A loop node contains three sub-regions: initializing the loop, the loop body and the loop exit test.

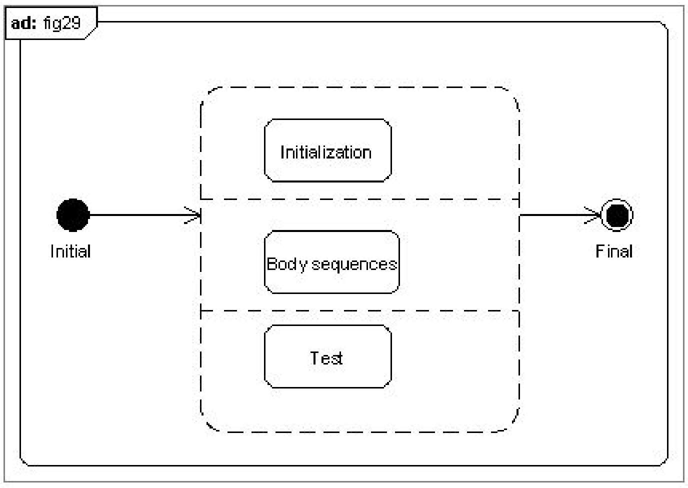
Figure 28



Remarks:

- the test sub-region can be evaluated before or after performing the sub-region which constitutes the body of the loop.
- the initializing sub-region is performed once when entering the loop.
- the other two sub-regions (loop body and the loop exit test) are performed once until the test sub-region generates a result whose value is false.
- UML specifications do not suggest any notation for loop nodes. Consequently, we can improvise one.

Figure 29

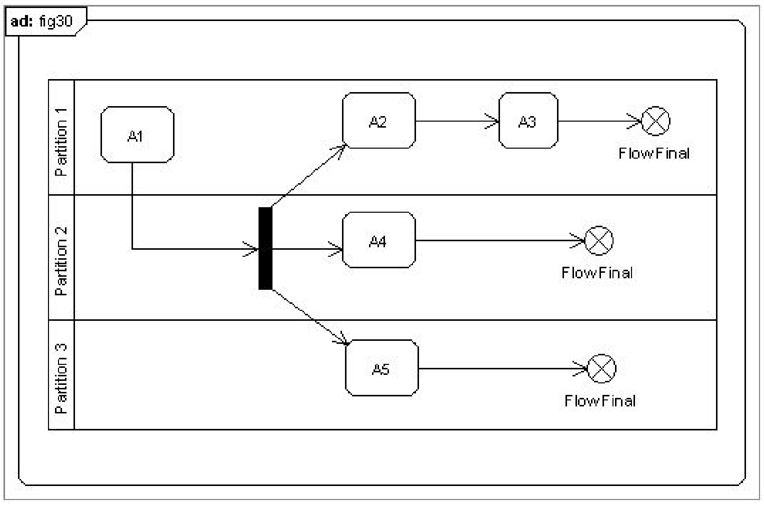


Activity Partitions

Very often, it is useful to indicate within an activity diagram which entity is responsible for an activity group.

For such situations, we decompose an activity diagram by means of an activity partition. Such a partition is represented by means of two parallel line, horizontal or vertical; the partition name is placed within a square at the end of this line. All nodes performed within this partition are placed between two parallel lines.

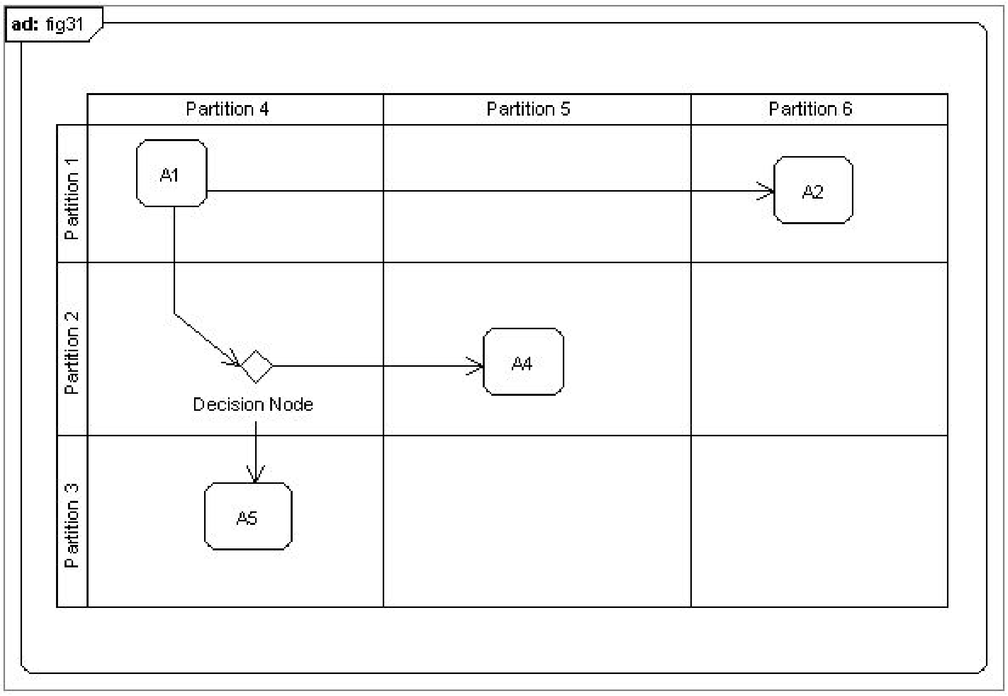
Figure 30



Remarks:

- a partition has no effect on the data flux within an activity.
- partitions can also represents attributes which contain specific values.
- partitions can be conjoined in order to materialise the complex restrictions of an activity diagram. We are talking about multidimensional partitions, a novelty introduced by UML 2 (see figure 31).

Figure 31



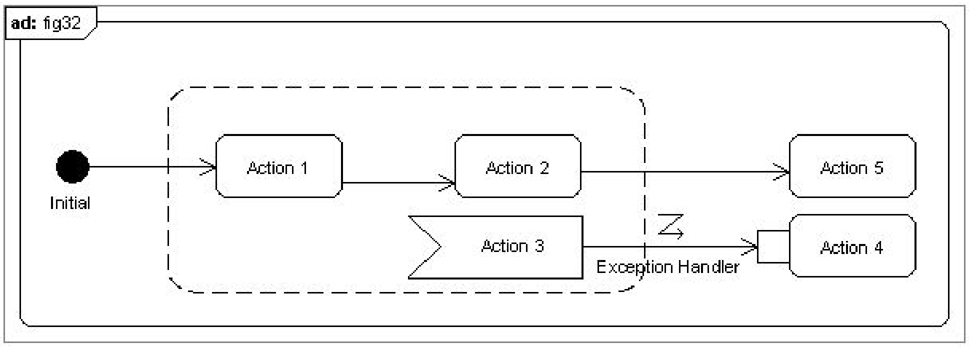
Activity Regions Sensitive to Interruption

In UML 2 we can indicate the fact that a region of an activity diagram can take a defining interruption of a token or of a transformation marking it as activity regions sensitive to interruption.

In order to represent an activity region sensitive to interruption we place the respective nodes within a rectangle with discontinuous contour and rounded edges. In order to indicate the interruption, we can use a lightening shaped arrow which starts from the region where the node should overtake the transformation. The token leaving the region is not affected.

An interruption is triggered by receiving a signal starting from an external entity. The reception of the signal is represented as a convex pentagon which contains the name of the signal (figure 32).

Figure 32



References

1. B a l z e r t , H. - *UML 2 compact*, Eyrolles, 2006
2. C h a r r o u x , B., O s m a n i , A., T h i e r r y - M i e g , Y. - *UML 2*, Pearson Education France, Collection Syntex, 2005
3. L a r m a n , C. - *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Prentice Hall, 1997
4. P i l o n e , D., P i t m a n , N., - UML 2 en concentre, O,Reilly, Paris, 2006
5. R o q u e s , P. - *UML 2 par la pratique*, 5-e édition, Eyrolles, 2006

Ciclul: UML PENTRU MANAGERI (VI) Diagrame de stare şi diagrame de activități

Rezumat

Diagramele de stare (tranziție a stărilor) permit reprezentarea comportamentului intrinsec al unui obiect. Diagramele de activități permit specificarea ordinii efectuării operațiilor (transformărilor). Acest articol prezintă atât conceptele de bază, cât şi conceptele avansate ale celor două tipuri de diagrame comportamentale – de stare şi de activități.