

Dynamic Model

Using the Object Model, we can specify the static architecture of the system classes, determining:

- The component classes and their properties, namely attributes, services and integrity constraints.
- The relationships between classes, namely association / aggregation / composition and specialisation / generalisation.
- The agent relationships, which determine who is authorised to do what, and to see what.

Obviously, this information is very important for the successful construction of a Conceptual Schema. Nevertheless, some parts are still missing from the puzzle of representing a System via a correct and complete Conceptual Schema. Once the static class structure is fixed, the specification must be completed by adding dynamic (related to intra-object control and inter-object communication), functional (characterization of service functionality) and presentation (characterization of the interaction mechanisms between user and system) aspects. It is necessary to view this under the global perspective of the Conceptual Schema, which unites all the abovementioned views (static, dynamic, functional and presentation).

In this section, we start by the specification of the conceptual primitives that are required to characterize the dynamic aspects. Specifically, the following aspects must be determined:

- Once the services available for each class are specified in the Object Model, it is necessary to decide in which order they may occur. In other words, it is necessary to determine which service sequences are allowed as valid lifetimes for the objects of each class.
- It is also necessary to determine which communication mechanisms are possible amongst objects of different classes. In particular, two mechanisms are particularly relevant:
 - How global transactions, which glue together services from different classes but comprise a single unit of execution, can be specified.

- Which services must be automatically activated in certain objects whenever some trigger conditions are met within other objects.

This captures the information that will be specified by the Dynamic Model of OO-Method, which is described now. In summary, the Dynamic Model will represent those aspects of the organizational system that refer to intra-object control, i.e. the determination of the potential service sequences that may occur during an object's lifetime, and to the interaction between objects that may occur in order to compose inter-object execution units. The set of primitives or conceptual constructors that OO-Method offers to specify all this information is presented now.

Following our strategy of adopting the notational standard defined by UML, we use the most adequate diagram types for graphical depictions. As we see in this section, a State Transition Diagram is used to describe the possible event sequences for each class, and an Interaction Diagram (a Collaboration Diagram, more specifically) is used to describe the communication and interaction between objects.

8.1 State Transition Diagram

As we have said, one of the main objectives of the Dynamic Model is to define the behaviour of objects over time, explicitly indicating the valid sequences of events that may occur during an object's lifetime. These are known as *possible lifetimes* of the objects of the class. In order to describe the valid sequences of events, a State Transition Diagram is used for each class. In these diagrams, states are used to represent each of the different *situations* in which an object of the associated class can be found, and to support the potential state transitions.

State Transition Diagrams are very important in the model, since they establish when an object can accept or must reject a service request that an agent object tries to activate, depending on their state (their particular *situation*). In other words, given a specific state, only a subset of all services can be activated. The states of diagrams of this type therefore constrain the potential valid transitions.

From a philosophical viewpoint, the existence of each object of a class passes through a sequence of states that characterize its life. The current state of an object is the result of its lifetime history up to the moment. From this perspective, an object's lifetime can be seen as the sequence of services that have occurred for the object. In addition, an object's current state determines what can happen to the object in that precise instance; in other words, the set of services that can be potentially activated for an object depends on the state of that object at that precise moment.

The graphical representation of the State Transition Diagram of each class includes:

- *States*, shown as circles that contain a name for the state.
- *Transitions*, shown as labelled arrows between two states, named "source state" and "destination state" or, more simply, "source" and "destination".

Example. Although the notation for each of the elements that compose state transition diagrams is introduced first, the reader can refer now to the example in Fig. 8.5 to obtain an overall view of the diagram type being discussed.

8.1.1 States

As we have said, the graphical representation of the dynamic model of OO-Method consists of a state transition diagram for each class, in which states appear as circles containing the state name. States in this diagram represent each of the different situations in which an object of the associated class may find itself at any point during its lifetime. States have the following types:

- *Pre-creation state.* Represents the situation in which objects are immediately before they are created. Only creation events (new) leave states of this type. No incoming transitions can exist. It is graphically depicted by two concentric circles, as Fig. 8.1 shows.

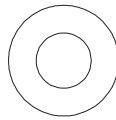


Fig. 8.1 Graphical notation for a pre-creation state.

- *Destruction state.* Represents the situation in which objects are immediately after they are destroyed. Only destroy events (destroy) come into states of this type. No outgoing transitions can exist. States of this kind are depicted by a black filled circle surrounded by a larger circle (bull's eye), as Fig. 8.2 shows. State changes associated with pre-creation and destruction states are considered to be strong state changes, because they take the object from non-existence to existence, and vice versa.

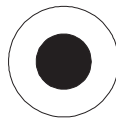


Fig. 8.2 Graphical notation for a destruction state.

- *Simple state.* These have a name, and both incoming and outgoing transitions are possible, originating from the activation of services (events or local transactions). They are depicted by a circle labelled with a representative name, as Fig. 8.3 shows.

Graphically, a state transition diagram cannot show more than one pre-creation state. With regard to destruction states, this may be zero or one, since it is not

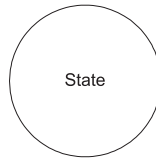


Fig. 8.3 Graphical notation for a simple state.

required that an object has a destruction state. If no destruction event is specified, objects that cannot be deleted become possible; they are “immortal” objects. There is no limitation to the number of intermediate states.

8.1.2 Transitions

State transitions represent a change of state of an object. This is the way to represent the move from one situation to another caused by the activation of a service by an agent, i.e. as the consequence of the activation of an action.

Graphically, transitions are depicted by a solid line with an arrowhead from the source state into the destination state. The arrow is labelled with an action and, optionally, with a control condition, as explained by Fig. 8.4 and below.



Fig. 8.4 Graphical notation for a state transition.

It must be recalled that an *action* is a service plus a list of agents that may activate it. The syntax to represent an action is as follows:

[Agents]: Service

where *Agents* may be:

- A list of names of agent class, which constrains the service agent list (all the classes authorised to act as agents of the service) to the subset of classes given by the list.
- An empty list, so that no agent (regardless of whether any agent has been defined for the service) is allowed to execute it. This situation represents that the action is part of a transaction and has no external visibility (internal action). Otherwise, an action without an associated actor cannot be activated.
- A list with the * symbol (asterisk) as the only element, which refers to all the classes that are agents of the service.

Example. In our system, there is a “rent” service for class “Vehicle”, of which classes “Client” and “Administrator” are agents. An action for which the “rent” service can be executed for all the agents of the service is represented as follows:

[]: rent*

Example. An action for which the “rent” service can be executed in the context of a transaction with no explicit agent attached to it (regardless of whether an agent for the “rent” service has been specified) is represented as follows:

rent

Note that, since the agent list is empty, the square brackets and the colon are omitted.

Example. An action for which the “rent” service can be activated only by instances of one of its class agents (for example, instances of the class “Client”) is represented as follows:

[Client]: rent

Understandably, state transitions that start from a pre-creation state must be labelled with actions that correspond to creation services (creation events or local creation transactions for which the first service is a creation service).

State transitions that end at a destruction state must be labelled with actions that correspond to destruction services (destruction events or local destruction transactions for which the last service is a destruction service, since the object ceases existence immediately after a destruction event, and no additional services can be executed on it).

No transitions labelled with creation service actions can originate from a simple state. Similarly, no destruction service actions can end at simple states.

A transition may begin and end at the same state. This is the case when, for any given action, the characterization of different situations (corresponding to the initial and final states) is not considered to be relevant from a dynamic modelling perspective. A change of state associated to the execution of the action may exist but, in these cases, the situation represented by the final state is the same as that of the initial one.

Example. Consider a class “Person” with the services “BeBorn”, “GetMarried”, “HaveChild” and “GetSeparated”. In the state transition diagram for this class we will have, in addition to pre-creation and destruction states, the simple states “Single”, “Married” and “Separated”, since we assume to be interested in characterizing the person’s marital status as relevant. The “BeBorn” service labels a transition originating at the pre-creation state and finishing at the “Single” state. The “GetMarried” service labels one transition from the “Single” state and into the “Married” state, and another transition from “Separated” to “Married”. Finally, the “GetSeparated” service labels a transition from “Married” to “Separated”. This is sufficient to capture the valid lifetimes of any person instance as far as his/her marital status is concerned.

Whether or not the person has children is not considered relevant from this perspective, since a person can have children irrespective of his/her marital status, and having children does not imply a marital status change. Therefore, the “HaveChild” service labels transitions from “Single” to “Single”, from “Married” to “Married”, and from “Separated” to “Separated”. Alternatively, this characterization (having or not having children) could be represented decomposing the state transition diagram into two different subdiagrams, composed using an AND composition for determining the object’s state.

A **control condition** is a well-formed formula (wff) that is used to determine the destination state that is achieved after executing an action that labels multiple transitions with the same source state but different destination states. The wff is constructed using constant values, object attributes (including those visible in related objects) and arguments to the service of the action. The syntax for a control condition involves using the *when* keyword to separate the action from the control condition:

[Agents]: Service when Condition

Example. *The state transition diagram for the “Vehicle” class includes:*

- the simple state “VRegular”
- the simple state “VAirCon”
- the action *[Administrator]: InsExtras*
- a transition labelled with the action just mentioned from “VRegular” to “VRegular”
- a transition labelled with the same action from “VRegular” to “VAirCon”

Whenever an “Administrator” agent executes the “InsExtras” service to add an extra to the vehicle, and if the vehicle is in the “VRegular” state, how can we determine the final state of the vehicle?

Control conditions can be added to both transitions in order to resolve this ambiguity. The transition from “VRegular” to “VAirCon” has this condition added:

pvc_Extra.Description = “air conditioning”

where “pvc_Extra” is the object-valued argument to the insertion event “InsExtra” that represents the extra being added to the vehicle, and “Description” is an attribute of the “Extra” class.

The transition from “VRegular” to “VRegular”, in turn, has this condition added:

pvc_Extra.Description <> “air conditioning”

In this way, it is easy to distinguish precisely the two situations in which a vehicle can be found: “VRegular” (regular vehicle), in which a vehicle remains as long as air conditioning is not added as an extra; and “VAirCon” (air-conditioned vehicle), to which a vehicle transitions as soon as an air-conditioning extra is added.¹

8.1.3 Basic Statechart Diagram

Every class has, by default, a basic state transition diagram that is built from the services of the class. This basic state transition diagram represents the essential

¹ It is worth mentioning that control conditions could have been avoided in this example by introducing “InsAirCon” as an independent event. This would have constituted a different modelling alternative. For the sake of the example, we have opted for using the “InsExtras” event with a description argument that refers to the kind of extra being selected (such as “air conditioning”). It is not our intention to discuss here the subjectivity factor already treated above and that is inherent to conceptual modelling, which often results in different conceptual schemas for the very same problem, depending on the point of view of the person doing the modelling.

valid lifetimes for the objects of that class, which begin with a creation event, continue with an intermediate state, on which most services operate, and finalize with a destruction event. The basic state transition diagram (shown in Fig. 8.5) is composed of three states (pre-creation, destruction and an intermediate simple state), and the following transitions:

- those starting at the pre-creation state and arriving at the intermediate state, for every creation service.
- those starting at the intermediate state and arriving at the destruction state, for every destruction service.
- those implementing a “loop” on the intermediate state, for all other services.

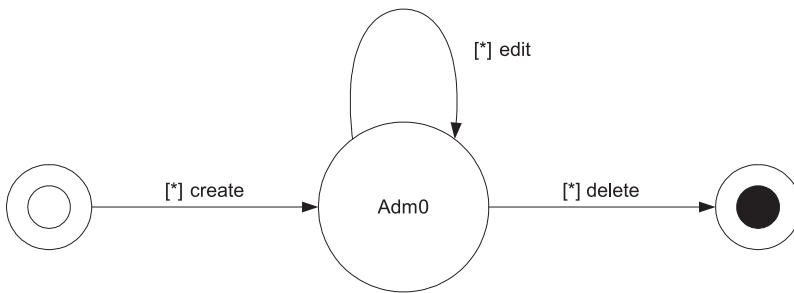


Fig. 8.5 State transition diagram (STD) for the “Administrator” class as an example of a basic STD.

If such a basic state transition diagram obtained by default is considered to be sufficient as far as modelling is concerned, then it is not necessary to modify it. If, on the contrary, states and transitions need to be refined in order to obtain a more detailed description of the possible valid lifetimes of the objects of the class, then the basic state transition diagram constitutes an excellent starting point, since it contains all the necessary information to approach the fleshing-out process.

8.1.4 Managing Complexity in State Transition Diagrams

We have explained how State Transition Diagrams are useful to characterize the valid lifetimes of objects with regard to the state changes caused by the occurrence of services. These states in the diagram are not associated to any particular sets of values for the object’s attributes, but represent relevant situations in which an object can be found due to the sequence of services executed up to that moment (and not because of changes of the values of the object’s attributes).

Sometimes, it is useful to distinguish between different subsets of states of an object in the problem domain. This happens with the description of global object state as seen in the AND composition of a set a substates, where the different valid paths for different ordered occurrences of class services are specified.

Example. Consider a class “Person”. We wish to distinguish between two types of states: those that refer to the person’s marital status (“Single”, “Married”, “Separated”) and those that refer to the person’s employment situation (“Employed”, “Unemployed”).

If we had to represent different spaces or subsets of states in a single state transition diagram, then a number of states equal to the Cartesian product of the states of each subset would be necessary; in turn, these would cause a large number of necessary state transitions. This would mean a considerable increment in the complexity of the state transition diagram, which would affect its creation, maintenance, readability and comprehensibility.

Example. In order to represent the situation outlined in the previous example in a single State Transition Diagram, we would need to create six different states, which result from the Cartesian product of the states in the two subsets: “Single-Employed”, “Single-Unemployed”, “Married-Employed”, “Married-Unemployed”, “Separated-Employed” and “Separated-Unemployed”.

This can be avoided by using a state transition diagram composition mechanism such as the one proposed by Harel with his statecharts (Harel et al. 1987).

Example. Continuing with the case from previous examples, it is possible to model an AND state that is made of two OR states. The first OR state would group the states “Single”, “Married” and “Separated”, as well as the transitions between these (“get_married” labels the transition from “Single” to “Married” and the transition from “Separated” to “Married”; “get_separated” labels the transition from “Married” to “Separated”). The second OR state would group the states “Employed” and “Unemployed” as well as the transitions between these (“find_job” labels the transition from “Unemployed” to “Employed”, and “quit_job” labels the transition from “Employed” to “Unemployed”). The creation service of class “Person” labels the transition from the pre-creation state to the AND state, and the destruction service labels the transition from the AND state to the destruction state. This can be seen in Fig. 8.6.

8.1.5 Operations and State Transition Diagrams

When we explained how actions label a state transition, we stated that an action consists of a service plus a list of agents that may activate that service. Also, we have said that classes can have three different kinds of services: events, transactions and operations.

Operations do not participate in State Transition Diagrams. The reason is simple: both events and transactions follow an “all or nothing” policy for their execution. If something fails during the execution of an event or transaction, all the changes done are undone and the system is left in the same state as that immediately before the execution. This includes state transitions. An event or transaction that causes a state change in an object from state A to state B leaves the object in state A if the execution fails, or in state B if the execution succeeds.

Operations, however, do not follow this policy, and therefore a failure during the execution of any of its component services does not cause the system to revert to

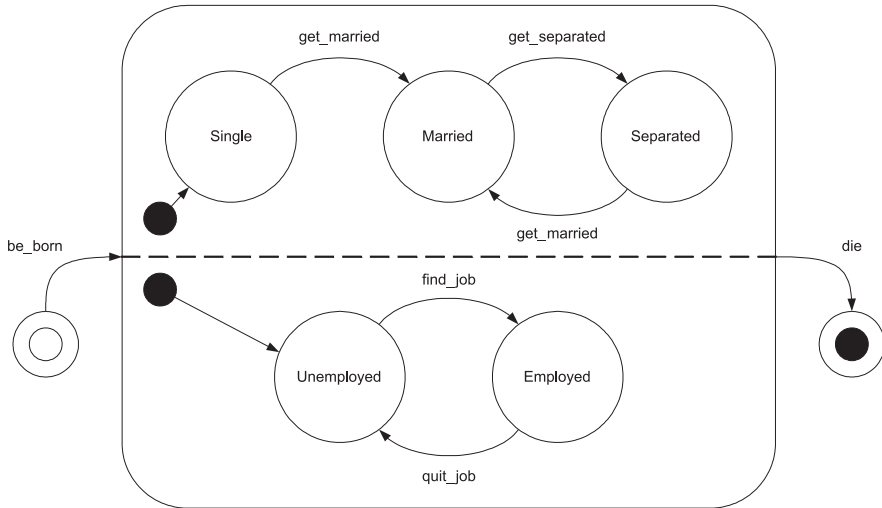


Fig. 8.6 Usage of Harel statecharts adapted to OO-Method State Transition Diagrams.

the previous state. For this reason, operations are not used to label actions in a State Transition diagram, delegating the triggering of transitions to their component services as long as they are not, in turn, operations.

8.1.6 Specialisation and State Transition Diagrams

This section describes how an object's valid lifetime is the sequence of services (events or transactions) that take place for it. This is represented via a State Transition Diagram in OO-Method, which is related to the State Diagram in UML.

We have also stated that, when a class is specialised, the child class inherits the static structure as well as the behaviour of the parent class. Given two classes, PC (parent class) and CC (child class), the following must be taken into account:

- All the attributes and services defined for PC are also attributes and services of CC. In addition, CC can incorporate new attributes and services.
- If the lifecycle specified for CC is restricted to the services defined for PC only, a subset of the lifecycle specified for PC must be obtained. Consequently, we can say that the lifetime of an instance of CC is a valid lifetime when this instance is seen as an instance of PC. Specifically, and in the case that CC is a permanent specialisation of PC, its lifecycle (restricted to the services defined for PC) must coincide with that of PC. If, on the contrary, CC is a temporary specialisation of PC, then its lifecycle (restricted to the services defined for PC) may be a subset of the lifecycle of PC.

In other words, if the states not inherited from the parent class and the transitions labelled with actions of services not inherited from the parent class were removed from the State Transition diagram (STD) of the child class, then the resulting STD would be equivalent to the STD of the parent class.

As we have said, temporary specialisation can happen by event or by condition. In order to maintain the compatibility between the parent and child classes, the temporary specialisation by event must take into account the following:

- All the destination states of transitions labelled with *carrier events* (i.e. events that cause the object to specialise) in the parent class must be inherited by the child class and cannot be renamed.
- In the STD diagram of the child class, all the *freeing events* must have an inherited destination state as destination state. This allows the object, once it starts behaving as the parent, to continue its lifecycle.
- In the statechart of the child class, all the *carrier events* must start at the pre-creation state.

The case of temporary specialisation by condition is similar to that of temporary specialisation by event, the difference being that carrier and freeing events are not explicitly stated. These events, however, can be identified as those events that may make the specialisation condition become true or false, by modifying the variable attributes that define the said condition. (This information, as we explain elsewhere, is specified in the Functional Model).

8.2 Object Interaction Diagram

We have explained how the intra-object control mechanisms associated with the valid lifetimes of a class can be specified. Inter-object interaction mechanisms must now be determined. The need to model object interactions in the system leads us to introduce an object communication/interaction diagram.

The Model of Objects supported by OO-Method involves two mechanisms for inter-object communication: *triggers* and *global transactions* (*global operations*).

- *Triggers*. A condition on an object's state becoming true may cause this object to trigger events or transactions on itself or on other objects in the system. This communication mechanism is called *Trigger* in OASIS (Pastor 1992; Pastor and Ramos 1995).

The syntax of a trigger in OASIS is:

`<destination>::<condition>:<service>`

where `<destination> := self | object | class | for all`

Each of these options is discussed below. Also, service arguments can be initialized to constant values or values of attributes of the class causing the trigger.

- *Global Transactions and Global Operations*. Within a transaction, the sequence of services that become activated constitutes a unit of execution. If the events, transactions or operations that compose it belong to the same class, then the transaction is a local transaction (operation). If the sequence is composed of events, transactions or operations of multiple classes, then the transaction is a global transaction (operation; Pastor and Ramos 1995). The former case corresponds

to an intra-object transaction, while the latter corresponds to an inter-object transaction.

Formally, a global transaction (operation) may be seen as a local service of a class that would be obtained by aggregating all the classes that contribute services to the global transaction. This aggregate class would have at least one relevant emergent property, namely the abovementioned global service seen as a local service within the aggregate class. For the sake of simplicity, global services can be seen as services of the parallel composition class that is the Conceptual Schema, already introduced, this making an appropriate context for the specification of any property of a general kind.

Every global transaction (operation) has a name assigned, and arguments may be defined as well. The arguments of the component services can be initialized to constant values or arguments of the global transaction (operation). The details of this are explained in another section.

Now, we describe the notation of the interaction diagrams that are used in OO-Method to represent the communication/interaction between classes in the system. The adopted notation is a subset of the standard proposed by UML (UML 2004) for Collaboration Diagrams. As on previous occasions, the primitives used by these Collaboration Diagrams adapted to OO-Method are precisely defined, and their semantics are not ambiguous. This makes their use much simpler than that of generic Collaboration Diagrams, as described by UML.

For this notation, a class is depicted as a box labelled with the underlined text `Instance:ClassName` across the top. The Instance label denotes a specific object of the class (optionally), whereas the `ClassName` label refers to a class in the system. The latter must always be specified, and the former is necessary only if multiple objects of the same class are interacting. In this way, the diagram conveys that interaction happens between objects (instances of the class “`ClassName`”) and allows for the individual identification of each object via the “Instance” label. Arrows flowing from a class box to other boxes (or to itself) depict communication/interaction. Now, we augment the information about the graphical notation, explaining how the different kinds of communication/interaction that have been introduced can be depicted correctly and clearly.

8.2.1 Triggers

The triggering of a service occurs as a consequence of the satisfaction of a condition on the state of an object. The label associated with the communication/interaction arrows has the following format: [`<destination>::<condition>:<service>`]. The definition of a trigger involves the following properties:

- Definition Class, which is the class in terms of which the trigger condition is specified.
- Trigger Condition, which is a well-formed Boolean formula involving constants, functions and attributes of the definition class (or related classes). The trigger is activated in every instance of the definition class for which the condition is satisfied.

- Destination, which refers to the entity that receives the triggered service, and that determines the trigger kind. Possible options for this destination type are:
 - Self, if the service being triggered is to be executed on the same object as that satisfying the trigger condition.
 - Object, when the service being triggered is executed on a specific object of a class, which does not need to be the same class as that for which the trigger condition is satisfied. In this case, the identifier (or identifiers) of the object on which the service is to be executed must be provided.
 - Class, when the service being triggered is executed on the complete population of a class (i.e. all the instances of that class). As in the previous case, this class does not need to be the same class as that for which the trigger condition is satisfied.
 - For All, when the service being triggered is executed on a subset of the population of a class reached through a sequence of role names from the definition class. This fourth option is, in fact, a special case of the third, in which the destination population is a subset of the class' population, rather than the whole set.
- Path to Destination, which is a deterministic sequence of role names that are visible from the definition class.
- Object Selection Clause, which is a well-formed Boolean formula that may involve constants, functions and attributes of the class reached by navigating the role name sequence from the definition class (or related classes). This clause restricts the set of objects on which the triggered service is executed to those reached from the definition class through the role name sequence that also satisfies the condition.
- Destination Class, which is the name of the class of the object or objects on which the triggered service is executed. This class depends on the destination kind:
 - Self: The destination class is the same as the definition class.
 - Object: The destination class may not coincide with the definition class.
 - Class: The destination class may not coincide with the definition class.
 - For All: The destination class is the one reached from the definition class through the role name sequence indicated by the “Path to Destination”, or a predecessor of such a class.
- Service, which is the name of the service that is triggered, plus initialization values for its arguments.
- Comments, which is an optional property that allows documenting the trigger definition.

The graphical specification of triggers is undertaken, as we have said, using a Collaboration Diagram, in which each kind of trigger is depicted as shown below. Depending on the destination, triggers are graphically depicted in one of the following four ways:

- **SELF:** If the trigger is directed to the same object as that where the condition is satisfied, then an arrow is used going out of the box representing the class, and coming into the same box and labelled with the word “self”. The trigger

condition and the service to be executed, together with its initialized arguments, are shown afterwards.

Example. In order to implement the fact that vehicles must be retired once they travel 100,000 km, the following SELF trigger can be modelled. The properties of this trigger can be seen in Table 8.1 below, and graphically in Fig. 8.7.

The trigger condition

Kilometres > 100000 AND Status <> "B"

specifies when the triggered service

Retire(pt_thisVehicle, "Retired after travelling 100,000 km")

will be activated and where (SELF indicating that the same object satisfying the trigger condition will be the recipient of the triggered service).

Table 8.1 Trigger to SELF in class "Vehicle".

Definition Class	Vehicle
Trigger Condition	Kilometres > 100000 AND Status <> "B"
Destination	SELF
Path	
Object Selection	
Destination Class	Vehicle
Service	Retire(pt_thisVehicle, "Retired after travelling 100,000 km")

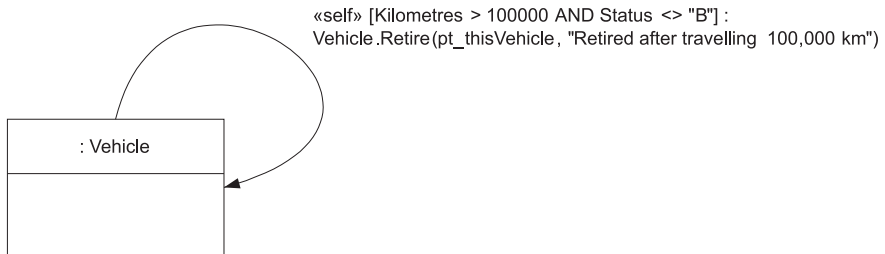


Fig. 8.7 Trigger to SELF in class "Vehicle".

As shown, the trigger condition not only checks that the number of travelled kilometres is over 100,000, it also checks that the vehicle's status is not "B" (meaning retired vehicles), so the "Retire" service is not invoked repeatedly. The "Retire" services sets the attribute "State" to the value "B", so it is invoked only once for the vehicle.

The "Retire" service has two arguments: "pt_thisVehicle", of type "Vehicle", refers to the vehicle to be retired; "pt_Reason", of type "String", describes the reason why the vehicle is retired and, in this example, is initialized to the literal "Retired after travelling 100,000 km".

- **OBJECT:** If the trigger is directed to a *specific object* of a different class (or other object of the same class), then an arrow is used going out of the box representing the class where the condition is satisfied, and coming into the box representing the destination class, and labelled with the syntax described earlier. The identity of the specific object is provided by initializing the object-valued argument of the triggered service to the destination object.

Example. *Let us imagine that the system must send a sale offer to any customer who has just returned a vehicle that has travelled over 100,000 km. This can be achieved through the introduction of a trigger to OBJECT having the aforementioned client as destination.*

In order to implement this, the event “SendSaleOffer(p_thisClient, p_Vehicle)” of class “Client” is used. The argument corresponding to the destination object (“p_thisClient”) is initialized to the instance of “Client” associated to the instance of “Vehicle” on which the trigger condition is satisfied (namely that the travelled distance is above 100,000 km and that the vehicle is not retired).

The second argument, representing the vehicle that is offered on sale, is initialized precisely to the instance of “Vehicle” that satisfies the trigger condition (using the reserved word “THIS”). The properties of this trigger can be seen in Table 8.2 below and, graphically, in Fig. 8.9.

Table 8.2 Trigger to OBJECT in class “Vehicle”.

Definition Class	Vehicle
Trigger Condition	Kilometres > 100000 AND Status <> “B”
Destination	OBJECT
Path	
Object Selection	
Destination Class	Client
Service	SendSaleOffer(Client, THIS)

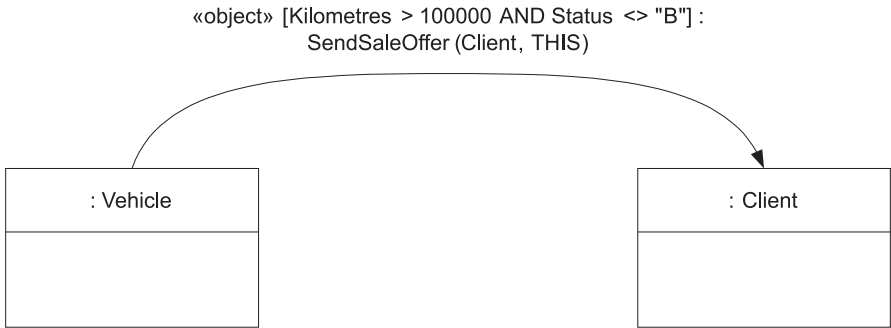


Fig. 8.8 Trigger to OBJECT in class “Vehicle”.

- **CLASS:** If the trigger is directed to *all objects* of the same class, then the same approach is used as in the case of *self*, but using the reserved word *class*, rather than *self* as trigger destination (the message may be intended for any class in the system).

Example. *If a vehicle is retired and has never been offered on sale, then the system must offer it on sale to every existing client. As this involves the whole population of class client, to accomplish this a trigger to CLASS executing the service*

“SendSaleOffer(p_thisClient, p_Vehicle)”

for each client in the system can be used. The properties of this trigger can be seen in Table 8.3 below and, graphically, in Fig. 8.9.

Table 8.3 Trigger to CLASS in class “Vehicle”.

Definition Class	Vehicle
Trigger Condition	Status = “B”
Destination	CLASS
Path	
Object Selection	
Destination Class	Client
Service	SendSaleOffer(p_thisClient, THIS)

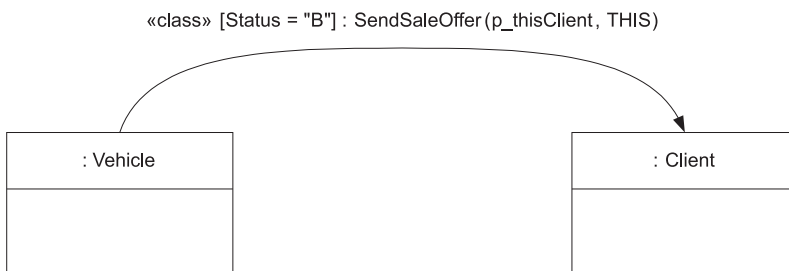


Fig. 8.9 Trigger to CLASS in class “Vehicle”.

In this case, the “p_thisClient” argument (representing the client that receives the sale offer) is not initialized, since the semantics of the trigger to CLASS will initialize it to each instance of the class “Client” in turn. However, the “p_Vehicle” argument is initialized to the reserved word “THIS”, thus indicating that the vehicle that satisfied the trigger condition will be offered to each of the clients.

- **FOR ALL:** If the trigger is directed to *a subset of the objects of a class*, the same approach is used as in the previous case but, rather than the *class* reserved word, an expression like that shown below is used:

<path>[where <path_condition>]

where

path is a deterministic role name sequence from the class defining the trigger, which determines the subset of objects that will receive the trigger, and

path_condition is an optional Boolean expression on attributes of the class reached by *path* that restricts the subset to only those objects that satisfy the condition. Therefore, and as we have already said, this fourth case is, in fact, a special case of the third case (trigger to CLASS).

Example. Assume that the system must delete the maintenance report history whenever a vehicle is retired because it has travelled over 100,000 km. Assume also that the “Retire” service, which changes the vehicle status to “B”, is triggered when this kilometre value is surpassed, as we described in a previous example.

It is now possible to introduce a trigger for the “delete” service in class “MaintReport” that affects only those instances related to the vehicle that satisfies the trigger condition, as shown in Table 8.4 below and, graphically, in Fig. 8.10.

Table 8.4 Trigger FOR ALL in class “Vehicle”.

Definition Class	Vehicle
Trigger Condition	Kilometres > 100000 AND Status <> “B”
Destination	FOR ALL
Path	MaintReports
Object Selection	
Destination Class	MaintReport
Service	Delete(p_thisMaintReport)

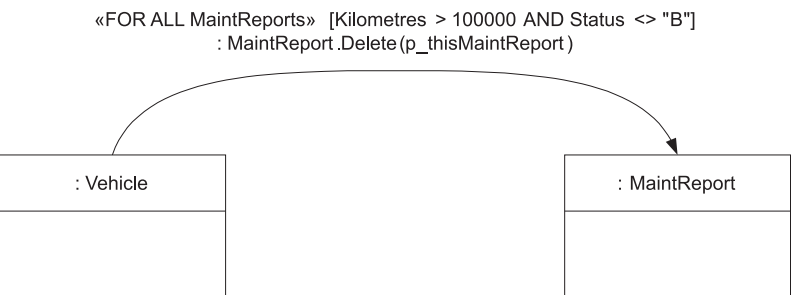


Fig. 8.10 Trigger FOR ALL in class “Vehicle”.

In this case, the path “MaintReports” determines that the population of objects of the destination class (namely “MaintReport”) for which the “Delete” service (of the same class) is invoked corresponds to those objects related to the vehicle that satisfies the trigger condition via the “MaintReports” role.

Constants, functions and attributes of the object satisfying the trigger condition (or other objects related to it) can be used for the arguments of the service to be triggered.

If the service to be triggered has an object-valued argument that represents the object for which the service is invoked (as is the case for all non-creation services), then such an argument does not need to be initialized explicitly, since the specification of the trigger destination (self, object, class, for all) carries out this initialization implicitly. Any other arguments of the service to be triggered, however, must be initialized.

Triggers represent internal activity in the system. Services are triggered when the appropriate trigger conditions are satisfied, without the user being aware of this. For this reason, it is convenient that every argument of the service to be triggered be appropriately initialized by the trigger definitions, so that the system does not need to require additional information from the user in order to trigger the services.

8.2.2 Global Transactions and Global Operations

We have mentioned the need to specify services with a granularity larger than a single class, in order to capture those cases in which a system service is composed of events or transactions that belong to multiple classes, all of them constituting a single unit of execution. OO-Method introduces the notion of global transaction or global operation in order to capture this mechanism, and its specification context is the Dynamic Model being discussed. Specifically, a global transaction (operation) is a “molecular” unit of execution (composed of multiple services), in which the component services may belong to different classes that do not need to be related.

From a methodological point of view, a global service is the conceptual representation of a given organizational task, modelled at a higher level of abstraction than the Conceptual Model. Business Processes, together with their associated tasks and activities, agents, resources, etc. are primitives in the Business Process Model or Workflow Model in the context of Organizational Modelling, which comprises the ultimate source of the services that appear in the Conceptual Schema. In fact, work is being carried out on OO-Method in order to extend its basic model-transformation metaphor to the transformation of Business Process Models into Conceptual Schemas. This work is beyond the scope of this book, but additional information can be found in Estrada et al. (2003), and Martinez et al. (2004).

In MDA terms, the core idea involves using a high-level Organizational Model as a PIM, which appropriately represents system requirements, and developing the appropriate mappings between conceptual primitives so that this model can be transformed into another, lower-level PIM (PIM-PIM transformation). The resulting PIM is the Object-Oriented Conceptual Schema that is described in this book and that serves as the starting point for the final transformation from PIM (Conceptual Schema) into the corresponding Product Software (PIM-PSM transformation).

Returning to our current discussion, a global transaction (operation) has a system-wide unique name. This name may coincide with the name of a service or class. If this happens, then no clash occurs, since local services (defined within

classes) are prefixed with their class name when referred to from global transactions (operations).

In addition, a global transaction (operation) has the following optional properties, with their usual semantics:

- Alias
- Comments
- help message

Being services, global transactions (operations) may carry arguments. As with local services, the properties of an argument of a global transaction (operation) are:

- a name
- a data type
- a data size, for arguments of type “string”
- a default value (optional)
- a flag indicating whether null values are accepted
- an alias (optional)
- a help message (optional)
- comments (optional)

The name of the argument must be unique in the context of the global service.

The default value is a well-formed formula, similar to those of local transactions/operations, of a type that is compatible with the argument. If a default value is provided for an argument, then the system will use it as a suggested value, the agent in charge of activating the service being able to change it.

In addition, arguments of a global service are one of two kinds:

- Input arguments. Their value must be determined for the global service to execute. The agent in charge of activating the service is responsible for assigning a value to them.
- Output arguments. They take a value as a consequence of execution, once the global service finishes successfully. Their value is expressed through a value expression, which is a well-formed formula built from input arguments and/or properties of any object in the system. In addition, conditions can be incorporated into its specification, so that the value of an output argument can depend on their satisfaction.

Example. *It is possible to combine the deletion of a vehicle and a client into a single service, using the global transaction*

“DELETEVEHICLEANDCLIENT(ptg_Vehicle, ptg_Client)”,

which receives two object-valued arguments that represent, respectively, the vehicle and client that are to be deleted. This global transaction composes transactions “DELETEVEHICLE(pt_Vehicle)” and “DELETECLIENT(pt_Client)” of classes “Vehicle” and “Client” respectively, as the following formula shows:

DELETEVEHICLE(ptg_Vehicle) . DELETECLIENT(ptg_Client)

This formula shows that the arguments “pt_Vehicle” (from the local transaction “DELETEVEHICLE”) and “pt_Client” (from the local transaction “DELETECLIENT”) are initialized, respectively, to the arguments “ptg_Vehicle” and “ptg_Client” of the global transaction “DELETEVEHICLEANDCLIENT”.

The considerations to be made for the value expressions of output arguments of global services are the same as those made for output arguments of local services, so we will not repeat these here.

The specification of a global transaction or operation needs to define the corresponding service composition through a formula, as do local transactions and operations.

The syntax used to specify such formulae is, basically, the same as that used for local transactions and operations. There is a single difference, due to the special nature of these services. Since these are global services, they can be seen as being defined for the class that represents the complete Conceptual Schema. As we have said before, this class is obtained by parallel composition of all the system classes, and is implicitly related to them.

This global class relates to a system class A via a role name A, so that every role name sequence that appears within the specification of a global transaction or operation must always start with a role name that matches the name of a class in the system.

Example. *Let us assume that a service to delete all vehicles, clients and rentals is needed. The classes “Vehicle”, “Client” and “Rental” already have local transactions “DELETEVEHICLE”, “DELETECLIENT” and “DELETERENTAL”, which implement the deletion of a vehicle, client and rental respectively. Using global transactions, it is possible to compose these three services into a single unit of execution, and make them run not on a single instance of each class but on the complete population of objects of each class.*

Thus, a global transaction “DELETEAVC” is introduced with no arguments and the following formula:

```
FOR ALL Rental DO Rental.DELETERENTAL(Rental) .
FOR ALL Client DO Client.DELETECLIENT(Client) .
FOR ALL Vehicle DO Vehicle.DELETEVEHICLE(Vehicle)
```

Note that role sequences used with the collection iteration operator “FOR ALL”, in the case of global transactions, must start with the name of a class, rather than the name of a role. This is so because global transactions are not defined (and, therefore, executed) within any particular class. Instead, we can see the whole conceptual model as a single class that relates to every other class via “roles” of which the names coincide with the name of each class.

Let us examine the first action of this global transaction in detail (the other two are similar).

The “FOR ALL Rental” part iterates over the complete object population of the “Rental” class.

The “DO Rental.DELETERENTAL” part selects the service to be executed (in this case, “DELETERENTAL” of class “Rental”, since “Rental” is the class name preceding the service name “DELETERENTAL”).

Finally, the argument “pt_thisRental” of local transaction “DELETE RENTAL” is initialized to the same role sequence as that used in the “FOR ALL ...” part, thus indicating that the rental on which the local transaction “DELETERENTAL” is to be executed corresponds to each iteration of “FOR ALL Rental”.

Since the implicit relationship with each of the classes in the system gives visibility and access to the complete population of each class, global transactions and operations often use the FOR ALL operator with a WHERE clause in order to restrict the population on which services must be executed.

Example. It is possible to delete from the system all those vehicles that have been retired, by using a global transaction on the “Vehicle” class restricted to those instances with a retired status. This global transaction, called “DELETERETIRED-VEHICLES”, does not need arguments, and is defined by the following formula:

```
FOR ALL Vehicle WHERE Vehicle.Status = “B”
DO Vehicle.DELETEVEHICLE(Vehicle)
```

This formula restricts the population of vehicles on which the FOR ALL clause iterates to those of which the status equals “B” (meaning retired).

A different alternative would be to define object-valued arguments in the global transaction (operation) that represent the objects on which we wish to act.

Example. It is possible to issue invoices for all rentals of a given vehicle by using a global transaction “ISSUERENTALINVOICES(ptg_Vehicle)”, to which the vehicle of which the rentals must be invoiced is passed as an argument. The formula of this global transaction is as follows:

```
FOR ALL Rental WHERE Rental.Vehicle = ptg_Vehicle
DO Rental.IssueInvoice(Rental)
```

Note how, in this case, the population of “Rental” on which the FOR ALL clause iterates (and, therefore, on which the service will be executed) is restricted to those rentals associated to the target vehicle by the comparison to the object-valued argument “ptg_Vehicle”.

The graphical representation of global services in a Conceptual Schema is now described. Using a Collaboration Diagram as a starting point (like in the case of triggers), arrows are drawn between the boxes corresponding to the classes that participate in the global interaction, representing the service activation sequence, and are labelled with each service to be executed.

Each global service is assigned a unique number. Arrows belonging to the same service are labelled with the same number. This number allows the modeller to visually track the component transitions of a global service.

Arrows are labelled using the following format:

[GlobalServiceNumber] :: [Condition :] Service

where

- *GlobalServiceNumber* is the number identifying the global service.
- *Condition* represents a guard of necessary condition for the service to execute.
- *Service* can be a service local to a class or a global service, declared preserving the valid syntax for services in this context.

Example. Figure 8.11 shows the graphical notation for a global transaction, of which the textual representation would be *class1.service1.class2.service2* (the concatenation of *service1* provided by *class1* and *service2* provided by *class2*).

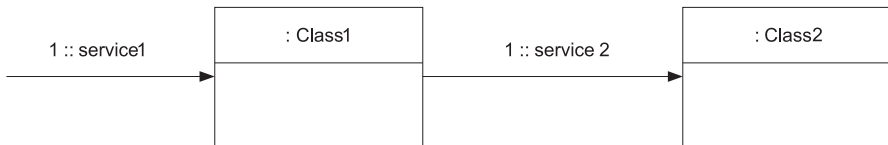


Fig. 8.11 Graphical notation for a global transaction.



<http://www.springer.com/978-3-540-71867-3>

Model-Driven Architecture in Practice
A Software Production Environment Based on
Conceptual Modeling

Pastor, Ó.; Molina, J.C.

2007, XVI, 302 p., Hardcover

ISBN: 978-3-540-71867-3