Chapter 7

Real-Time Behavioral Study Beyond UML

The dream of a software maker is how to get running software directly from a CASE tool, to make a jump from abstraction to functionality. The passage is targeted either from the PSM to code or from the PIM to code. The later joins the former with a preliminary transformation from PIM to PSM.

> Presently, visual programming environment based on C# or Java languages allows programmers to automate many programming tasks in a graphical window-based system. List boxes, combo boxes, text boxes, buttons, etc. and dynamical components can be created graphically on a design area, then events added to realize a windows-based interface frame that needs in the past hours and hours of programming tasks linking with complex libraries. When an event are created, for instance a click on a button, only its headers *button_click()* is added and the event inserted in the event list. The programmer has to add manually codes to say to the program how to handle this event.

Presently, many tools implement automatic code generation for different programming language out of static UML class diagram. When making classes, attributes and operations are already named, their types and visibilities specified, so theoretically, CASE tools can generate programming class headers and variables in any language.

For complex or user-defined data types or data structures, there must be some means that allows the developer to specify the complex structure in terms of existing types, by associations or by any other methods. Therefore, we suppose that this feature is not a theoretical issue when the complex structure can be mapped into elementary types available in the target language.

As for the code inside functions, programmers still need to fill in the blank if the corresponding diagram cannot establish the algorithm unambiguously, and if activities are not sufficiently decomposed to reach elementary actions and class/object operations. Operations in the PIM or PSM are considered as elementary if they correspond to action/activity not decomposable at their respective level. At low level, an elementary action is translated into one line of code in high level language (Basic, C, C++, C#, Java, SQL, etc.) or eventually into a code snippet. With a low-level language like assembler, an elementary action will be translated into more than one instruction, e.g. an assembler routine

D. M. Bui, Real Time Object Uniform Design Methodology with UML, 381–416.

or assembly code snippet. In this case, a library of code snippets identical to those used in compiler techniques can be used for this purpose.

For instance, in a real-time system, instructions like

$$\begin{aligned}&\text{Timer.Start()}\\&\text{Button.On()}\\&\text{Timer.Delay} = 10000\end{aligned}$$

are considered as elementary operations.

When all the previous conditions are validated, there is effectively a possibility to get a run-time environment out of the UML diagrams. If the operations are not elementary, in the PSM, we can store *high-level language source code snippets* (sometimes called *code patterns*) that can be exported directly from UML diagram specifications towards target language code as contents of class operations. This method is not recommended as it is like making code at diagrammatic level or bringing the low level up to the UML level.

> In some specific situation, code snippets are necessary for instance for importing libraries, defining macros, etc.

Programming languages are a mature domain. To really be able to support the automatic code generation between higher models and code, it is necessary to have a means of specifying unambiguously the behavior of system. The Behavior has been addressed with three important concepts in UML: interactions, activities, and states. They are related to other concepts: use cases at the Requirement Analysis, business processes at Design, and operations inside classes at Design and Implementation. "Behavioral" is a global concept merging the two subaspects, functional and dynamic, together. Functional aspects are preponderant when dealing with operations defined in classes, analyzing elementary actions, activities, processes, use cases, etc. Dynamic aspects are more important when discussing control structures, control flows, or workflow patterns. When reasoning with states, the two aspects are mixed in an inextricable way. To contribute to the effort of designing an unambiguous diagram for automatic code generation, in this chapter, we will expose a Petri-like network SEN (State-Event Network) (Bui, 2006). The SEN diagram proposes a unifying formalism melting together interactions, states, and activities that are finally three different views of the same reality. If interactions, states, and activities are used in the PIM and PSM, we need an intermediate model between PIM/PSM and the programming model that is code. Figure 7.0.0.1 locates the proposed model in the whole architecture.

The second aspect that will be discussed pertains to the safety of systems. As computers are ubiquitous in any applications, their failure (software and control aspect) is very costly, in terms of monetary loss and/or human suffering (aircrafts, railway traffic, safety in security systems, industrial plants, military

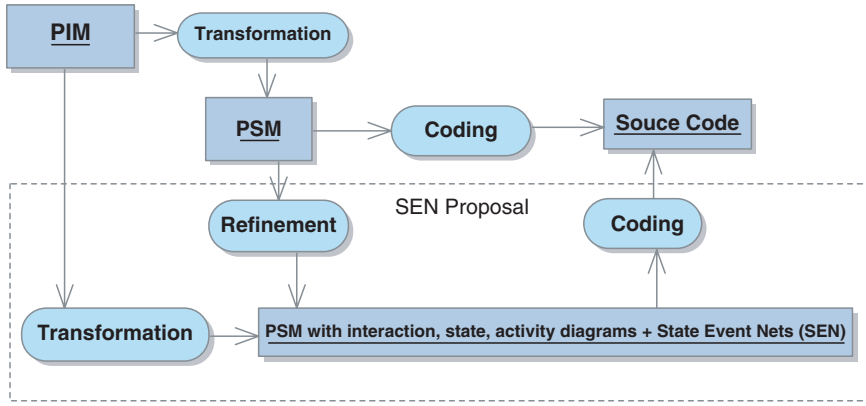


Fig. 7.0.0.1 The Dynamic Model developed with SEN (State Event Net) acts as intermediate model between PSM and Code. Three development paths are displayed: a standard PIM- > PSM- > Code, three other possibilities add the SEN to other dynamic diagrams as intermediate model before coding

weapons, banking systems, commercial worldwide transactions, etc.). Very often, the subtle design fault is at the origin of a catastrophic failure and the problem would be how to isolate such a design flaw that is often masked by the fact that the whole system seems to perform well and pass all functional approval tests. Is it one of the reasons why we still need a "black box" inside all aircraft when it passes all regular and maintenance tests? Some failure may arrive as a combination of circumstances. Its occurrence is so complex and unpredictable so that normal humans cannot anticipate. Complexity and unpredictability could hide design problems or design management issues. The problem is very complex because we can build a system that fulfills all SRS clauses, and however, this fact does not mean that the system is safe. Safety–critical situations may be disregarded in the specs even at the starting point of the development process. But, as demonstrated later, safety problem can be evidenced by developers. So, we will discuss all those aspects that are very important in real time embedded and embarked systems and propose a combinatorial methods called "Image attribute analysis" that, coupled with the SEN diagrams, can be an interesting research avenue for investigating safety-critical systems in a deterministic way. The first thing that must be done is to track all design faults and to realize the safest design.

7.1 Sen or State-Event Net Diagram

The expressiveness of UML dynamic diagrams is unquestionable, but there are some problems probably due to their high expressiveness.

> Let us take the sequence diagram that represents lifelines in vertical direction and messages between objects in horizontal direction. If this diagram shows objects, messages, and sequences at the same time, real system may suffer from the

> richness of displayed information. For presentation, we choose judiciously an application not decomposed completely or containing a limited number of objects. In real projects, the number of objects is more important, so the sequence diagram is often shown with a landscape paper view. If objects are very crowded, horizontal messages may start from the left side of the paper and reach its right side and an impression of complexity and density is perceived even with a relative limited number of messages. The communication diagram, its counterpart, can therefore be used to lighten the visual surcharge due to the message communication network, but we must accept to lose the time axis with the communication diagram.

The sequence diagram coupled with the interaction overview diagram are the best set of tools to show interactions at high level between components or between composite objects, but it would be problematic if a whole project must be realized completely and exclusively with diagrams of the interaction suite. Moreover, drawing an interaction diagram is a time-consuming task. If sequence diagrams can be used to display some important sequences of the system, building complete algorithms is time consuming. It is interesting to notice that class diagrams already contain operations that are images of the messages in the sequence diagram, so information is somewhat redundant. The power of the sequence diagram is showing messages exchanged between objects, so a sequence diagram is naturally tied to collaboration. From this remark, sequence diagrams are more predestined to collaborative tasks than for internal algorithms of objects (messages will be mostly "self messages" in this case). Interesting works have been done in the fields of formalizing scenarios (Alur et al., 1996; Uchitel et al., 2003).

The state machine diagram is recommended for representing the state evolution of an object in reaction to external events or internal stimuli. To represent the collaboration and synchronization among several objects, statecharts provide constructs like AND/OR states. State charts may have substates, of a higher level state, all active at the same time. Substates can communicate with each other. Orthogonal regions of the chart accept events sent to the object and regions can generate events to other regions. Guards are used as preconditions for state evolutions. Contour properties added by Harel increase the expressiveness of statecharts, so modern statecharts offer powerful facilities to succinctly specifying state evolution. Statecharts are therefore an image of the algorithms and is therefore an interesting candidate for formal methodologies (Golin Reiss, 1989; Harel and Gery, 1996; Gnesi et al., 1999; Harel et al., 2005).

In Wohed et al. (2005), the activity diagram has been analyzed with YAWL (Yet Another Workflow Language) (Aalst and Hofstede, 2005). Workflow control patterns (Aalst and Hofstede, 2002) are known issues. The syntax of the activity diagram has changed considerably passing from revisions 1.4–2.0 and for the first time, the name of Petri has been mentioned in the standard.

Research efforts are made to separately adopt interactions (in scenarios), activities, or states (in statecharts) to build tools for checking systems, for

generating code, etc. In this chapter, we try to unify these concepts and propose a network based on the Petri Net. Voluntarily, all redundant information (e.g. operations already defined in class diagrams) is pruned off. All visual surcharges are leveraged to maximum in order to achieve a very compact dynamic network that can be exploited as a visual algorithm before coding. As the interpretation is different from conventional ones, we coined in 1999 (Bui, 1999) the term SEN to name this Petri-like network.

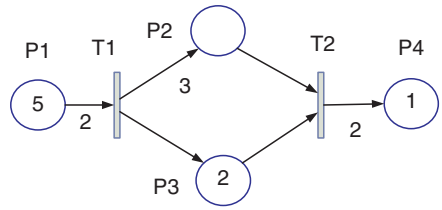
7.1.1 Mathematical Foundations of Petri Network

Petri net theory has developed considerably from its beginnings with Dr. Petri's (1962) dissertation. Later, Petri networks have been widely studied by researchers, specifically in modeling of systems (Peterson, 1981; Shlaer and Mellor, 1992; Sowa, 1984). French GRAFCET modeling, widely used in Europe for teaching, derives from Petri net as well (David and Alla, 1992; David, 1995). A Petri net is composed of four parts, a set of places P, a set of transitions T, an input function I that is a mapping from a transition T_j to a collection of places I (T_j) called "input places of transition T_j," and an output function O that is a mapping of transition T_j to a collection of places O (T_j) called "output places of transition T_j. The Petri net has its structure mathematically well defined so that it can be retrieved from its four-tuple (P, T, I, O). As multiple occurrences of places are allowed, we have "bags" of places that are generalizations of "sets."

On each place, we can store tokens. If we consider monochrome tokens, their number is written inside the place and the multiplicity of I and O functions are integers put on arrows connecting places to transitions. Remember that arrow can connect only places to transitions, never graphical objects of the same kind. By default, in the absence of numbers, places contain 0 token but arrow has multiplicity of 1. A Petri net with tokens are said to be marked with an n-vector, n equal to the total number of places. Coordinates of n-vector are positive integers.

A Petri net executes by "firing" or "triggering" transitions. A transition fires by removing tokens from its inputs places and creating tokens in its output places. The numbers of tokens removed and created are not necessarily equal but depend upon I and O functions.

The mathematical formulation of Petri net is known for a long time. But, it is not the only network that has a mathematical foundation. We suspected that all diagrams or networks must have under the cover some mathematical formulation so as to handle just their connectivity (CASE tools must handle more information than this minimum set, e.g. the location and the size of places and transitions in a paper area). However, the movement of tokens or markers and the way multiplicities of In() and Out() functions are used to relate the movement of tokens has transformed to Petri net into an executable diagram



P = {P1, P2, P3, P4}
T = {T1, T2}
I(T1) = {P1, P1}
O(T1) = {P2, P2, P2, P3}
I(T2) = {P4, P4}
O(T2) = {P2, P3}
Marking vector : M0 = (5, 0, 2, 1)
After firing T1 –> M1 = (3, 3, 3, 1)
T2 not authorized to fire for lack of 1 token

Fig. 7.1.1.1 Petri net with four places P1–P4 and two transitions T1–T2. Each transition has one In() and one Out() functions. Marking vectors memorize the number of monochrome tokens inside the ordered set of places. Firing is subject to the number of tokens in the original place, and the In() function. The number of tokens generated at arrival places depends upon the Out() function

and all dynamic diagrams based on Petri formalism is doubly dynamic (current dynamic diagram like interaction, activity, state diagrams show only all dynamic paths but cannot be executed, the reader must imagine a virtual marker when examining these diagrams).

7.1.2 Using Petri Nets in Modeling

In the real world, a task can be executed if the resource is available. Some task needs more than one resource and this fact can be specified as the multiplicity of the In() function. The Figure 7.1.2.1 illustrates the way we can use a Petri net to model task execution.

Bars are used to represent primitive events with 0 execution time (*Start processing* with $\Delta T0 = 0$ and *End processing* with $\Delta T1 = 0$). *Primitive events* theoretically cannot have overlapping zone as their $\Delta T = 0$. *Nonprimitive events* with some duration can have overlapped time zone. This property will be used later to make subnets managing net hierarchies to handle complex applications.

Originally, there is no inherent measure of time in a Petri net, so we have a partial ordering of time and occurrence of events. As an event can fire when it has enough tokens and appropriate multiplicities, more than one transition can be enabled at a time and nondeterministic results could result with a complex Petri nets with many firing possibilities. Many researches on timed Petri nets,

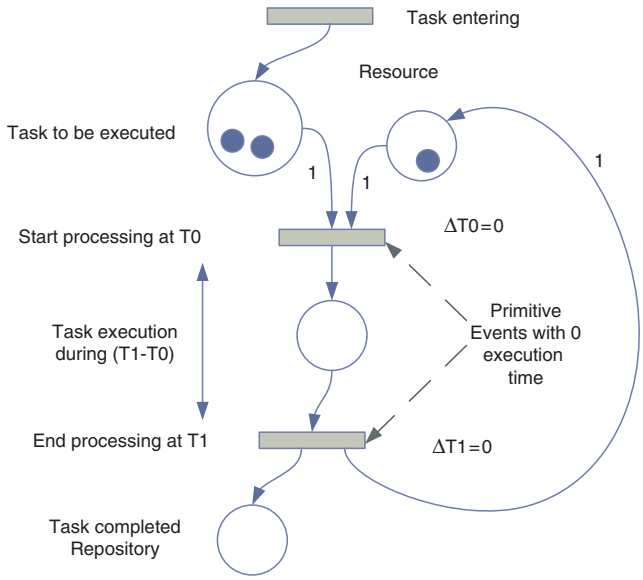


Fig. 7.1.2.1 Modeling task execution with a Petri net. Tasks need available resources to be executed. After execution of the 1st task, the resource token is returned to its original place and the system will execute the 2nd task. If a task needs two resource tokens, just mark right arrows with two instead of one. In this case, we need two resource tokens to execute one task

deterministic or stochastic nets have been made in the past. The reader can refer to a survey made by (Zurawski and Zhou, 1994).

7.1.3 State-Event Net Abstractions

As announced, the SEN diagram is a Petri-like net, so it inherits its essential characteristics (mathematical support, movement of tokens). As stated before, we need:

1. A diagram compatible with object paradigm
2. A *compact* diagram in order to express visually highly complex algorithms before attacking coding phase, a diagram for designers and programmers
3. A *nonredundant* diagram that does not express knowledge already inherent to other UML diagrams so as to leverage visual surcharge caused by redundant information
4. A *universal dynamic diagram* that can be used to express a large range of algorithms (from a simple operation defined within one object towards a complex inter operation involving a collaboration of multiple objects/components/agents)

5. A diagram belonging to an *intermediate model* located between the last PSM and the code. The UML diagrams are very expressive for problems taken at high level and could eventually be perceived as disproportioned if used at very low level

The definition of a new diagram must satisfy many preliminary conditions:

1. Possibility of mapping all dynamic concepts into visual elements and vice versa.
2. Presence of a *decomposition mechanism* for handling complex applications.
3. Presence of a *set of control structures* (for instance those defined in programming languages). Higher control structures can be found in Wohed et al (2005) or at the web site http://www.workflowpatterns.com. In assembly language programming, we know that this low level language has only an IF instruction. Associated with other system resources (counters, variables, multiple IF), more complex control structures can be built from the IF. So, this condition can be easily satisfied if we accept at the beginning that the SEN is closer to a low level programming language. More complex control structures can be added later if necessary.
4. Possibility of handling asynchronous events like interruptions.
5. Possibility of handling parallelism (e.g. UML statecharts, fork join control structures in the activity diagram).

For handling real-time system, it would be highly desirable to have a way of specifying time. Moreover, a deterministic interpretation of token movements is an advantage.

In the "Fundamental Concepts" chapter, we have already mentioned a close relationship between action, activity, task, event, condition, state, and message. These concepts lie on a same semantic continuum and the way we interpret a concept depends upon the observer viewpoint, its position on a cause–effect chain, the formalism used to interpret concepts. For instance, when an object (transmitter) executes an action, the receiver object catches it as an event. Where the state of the receiver commutes to could be viewed as a condition.

> For instance, saying that "car A hits car B" relates the story as an action, saying that "B is hit by A" describes the story as an event suffered by B and saying that "B is completely unusable after the shock" refers to the final condition of B.

Action and activity are very close semantically as activity can contain many actions. State is bound to activity or action. "A plane is flying" is both an activity and a state. In a SEN, all dynamic abstractions are represented by places. In contrast with the Petri net, transitions are not named and they serve uniquely

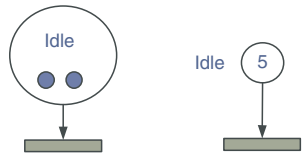


Fig. 7.1.3.1 Various graphic representations of SEN places. A SEN place represents all dynamic concepts (event, condition, state, action, activity). It can be reduced to a point with a name to save space on a diagram. The number placed near the dot give the number of tokens available in the place

to make control structures and to regulate the circulation of tokens in the SEN (except for referencing when a SEN must be partitioned into chunks in multiple sheets). Events are now considered as tokens that must appear in places. So, when an object executes an action in its SEN, it can generate a token in the SEN of another object (or in its proper SEN) and the way the two SEN are connected together mirror object interactions.

The SEN can specify any identified entity of the structural view. Thus, a system may be taken at any level of its decomposition; it can be the whole system, an external human actor, a component, an agent, a composite object, or a simple object. For graphical representation, the most "economic" visual representation would be only a dot for a place with the name written near the dot.

7.1.4 SEN Subnets

A diagram without natural support for complexity reduction through hierarchical decomposition could not be a main support for design. A SEN network supports two kinds of subnets. When entering/exiting a subnet through bars, it is defined as a *nonprimitive event* represented by a rectangular box. A primitive event is an event without zero duration and theoretically, two primitive events could never concur, only nonprimitive events can. SEN subnet with a "place" form is drawn with any means that can specify this fact (normally, we draw it with a double line and an oversized place to work in a monochrome context). Graphical representations for subnets do not add any new concepts besides place/bar/arrow; they pack only subdiagrams. There is no conservation of the number of tokens entering and exiting the subnet, we cannot know how long a subnet will keep the tokens as long as it is not decomposed and studied thoroughly.

7.1.5 SEN Control Structures

Elementary control structures in UML 2 are:

1. *Sequence of control flows*
2. *Decision node* defined in IntermediateActivities package. Decision node is a kind of IF/CASE with one incoming edge and several outgoing edges.

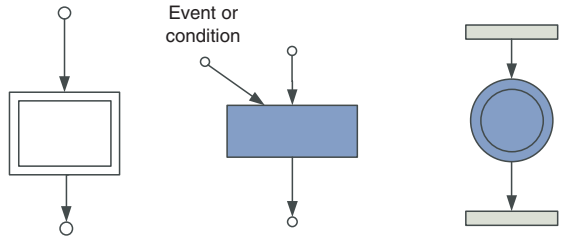


Fig. 7.1.4.1 SEN subnets. A nonprimitive event with a box materializing some duration and delimited with two primitive events can be used as SEN subnet. It must be connected to places. It can accept input conditions or events causing subnet entering. An oversized place with a double line is used to represent a subnet of "place type." It must be connected to bars

Edges can be either all object or all control flows. Each token arriving at a decision node can traverse only one outgoing edge. Tokens are not duplicated. Guards of the outgoing edge are evaluated to determine which edge should be traversed. The order in which guards are evaluated is not defined.

3. *Fork node*. A fork node has one incoming edge and multiple outgoing edges. Edges can be all object or all control flows. Tokens are duplicated across the outgoing edges. If an outgoing edge fails to accept a token, the latter remains in an FIFO queue, and the rest of the outgoing do not receive a token. The Fork Node may have guards on outgoing edges; in this case, "the modelers should ensure that no downstream joins depend on the arrival of tokens passing through the guarded edge. If that cannot be avoided, then a decision node should be introduced to have the guard, and shunt the token to the downstream join if the guards fail."

4. *Join node*. A join node synchronize multiple flows, has multiple incoming edges and one outgoing edge. "If a join node has an incoming object flow, it must have an outgoing object flow, otherwise, it must have an outgoing control flow." "If all the tokens offered on the incoming edges are control tokens, then one control token is offered on the outgoing edge." If *isCombinedDuplicate* attribute is true, then objects with the same identity are combined into one unique token.

5. *Merge node*. This node brings together multiple alternate flows. As a difference with the join node, it is not used to synchronize concurrent flows but to accept one among several alternate flows. Edges can be all object or all control flows.

Hereafter, control structures are implemented in SEN and explained.

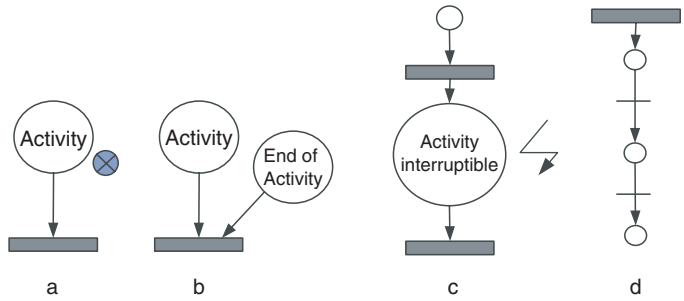


Fig. 7.1.5.1.1 Various End of Activity representation and Interruptible Activity. A global constraint can be used to characterize the whole project and force each activity to reach its end before releasing the token. In (a), a "circle with the X sign" means "End of Activity" and makes the option explicit. (b) gives an equivalent of (a) with a heavier notation. In (b), the end of the activity acts as a condition; when the activity reaches its end, the activity generates itself a token in the place "End of Activity" and the bar can fire. In (c), a lightning sign is used to express an interruptible activity. In (d), sequence of control flows without any condition can be represented with a simplified notation in which bars are reduced to a single dash put on the flow joining two places (normal bars can coexist with this simplified notation)

7.1.5.1 Sequence of Control Flows. As the SEN derives from the Petri net, places cannot be connected together directly with control flows as in the activity diagrams. We must therefore alternate places and bars. To reduce to the maximum the burden of drawing a bar after each place in the case sequential actions must succeed in time, a lightweight line can be put on the control flow to represent the bar. Moreover, each action or activities are supposed to execute to their end before releasing the place, by default, unless an interrupt forces them to leave the place prematurely. The end of process can be considered therefore as a condition not always represented for the alleviating the SEN diagram visual surcharge. In fact, in the presence of the possibility of interruption, a process can be interrupted before reaching its normal end, so the developer must weight by himself the default option (global interpretation constraints for the whole project) that can lower the burden of adding special signs or conditions.

7.1.5.2 Decision Node. The IF/CASE or exclusive choice (XOR-Split) is represented in the SEN as two or more bars with the condition (also called guard or predicate) in a place. When a predicate is evaluated to true, it is equivalent to generate a token in the corresponding place. The two bars' configuration corresponds to an IF and multiple bars corresponding to a CASE with one choice.

Conditions specified for outgoing arcs may overlap; in this case, the arc with the highest preference is selected. At design time, this preference can be specified by adding extra places filled with sequential numbers. When a condition is

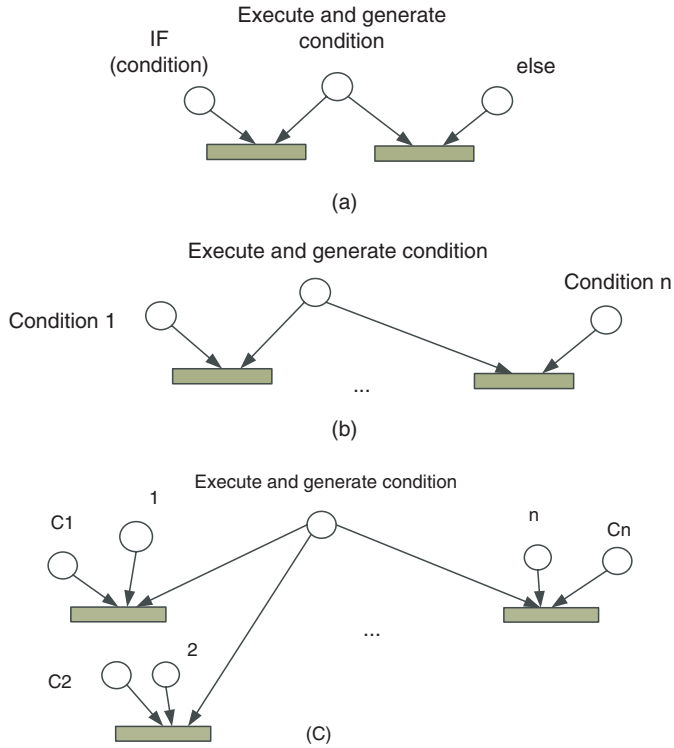


Fig. 7.1.5.2.1 Decision nodes (a) simple IF, (b) CASE with one outgoing edge, (c) handling complex case with overlapping conditions and possibility of parallel split. In (c), conditions specified for outgoing arcs may overlap; in this case, the arc with the highest preference is selected. At design time, this preference can be specified by adding extra places filled with sequential numbers (external to the place for distinguishing them from the token multiplicity). When a condition is evaluated to true, the corresponding sequential number is synthesized dynamically at this node. Conventionally, an invisible process (not represented) selects always the lowest value and affects a token to the place holding this value. If two conditions is verified at the same time (e.g. C1 true and C2 true) and the two numbers are identical (either 1 or 2), then two tokens are generated and the system behave as a parallel split on arcs 1 and 2. When $C1 = C2 = \ldots = Cn$ = true and all arcs have the same number, we have a parallel split on all arcs

evaluated to true, the sequential number is synthesized at this time and we consider conventionally that an invisible process (not represented) always selects the lowest value and affects a token to the place holding this value. With this convention, two outgoing arcs evaluated to true with the same number generate an equivalent of a parallel split with two arcs. So, there is a possibility of simultaneous generation of tokens on a subset of outgoing arcs having the same number with overlapping conditions. Therefore, by extending this idea, the parallel split is a special case of the decision node with all predicates evaluated to true and all numbered places having the same number.

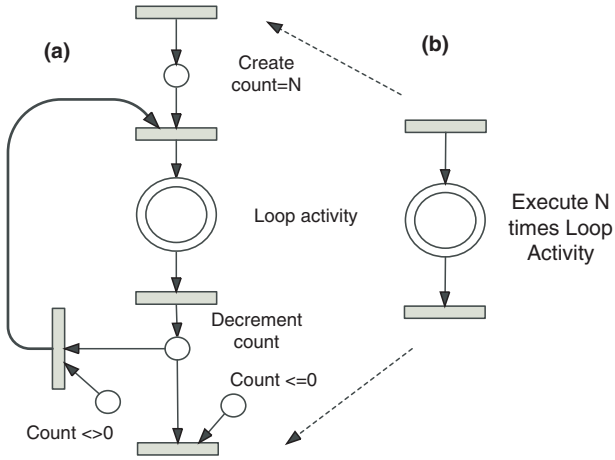


Fig. 7.1.5.3.1 Example of DO loop represented by a SEN at two levels, first at programming language level and at design level (as a sub SEN). (b) integrates all loop controls into the DO loop sub SEN. Conditions count $<>0$ or count $<=0$ come from Decrement count operation

7.1.5.3 Loop Nodes. Loop nodes can be easily made with the IF node by creating counters, extra conditions. We can define shorthand graphical notations if needed but as they are not really of theoretical concern but only conventions based on existing ones, we focus our attention only on elementary constructs. Figure 7.1.5.3.1 gives an example of the DO loop as taken at two levels: in programming language and at design time. In the later case, the DO loop is considered as a sub-SEN that does not need to be decomposed as all programmers are able to write a DO loop.

7.1.5.4 Fork, Join, and Merge Nodes. A fork node has one input on the bar and two or several outputs leaving the bar. It is also known as a parallel or AND split. A number of tokens duplicated will be equal to the outgoing edges. If conditions are used on edges outgoing from a fork, developers must ensure that no downstream join starves for token retained by conditions; there must be some mechanism for ensuring that the downstream join do work properly.

If a fork node is of configuration *1 to N*, N tokens will be generated after traversing the fork bar (if we suppose that all outgoing edges have individually 1 as multiplicity). If a join node is of configuration *M to 1*, the bar must collect M tokens (if we suppose that all incoming edges have individually 1 as multiplicity) before constituting a unique token on its outgoing edge. It is not the case for the Merge node. In the SEN, to insure a symmetrical representation of fork and join nodes, the join node is drawn as a mirror of a fork, without any added condition. The merge node has the same representation as the join node with a small M sign added to the bar to release the synchronization condition. Each

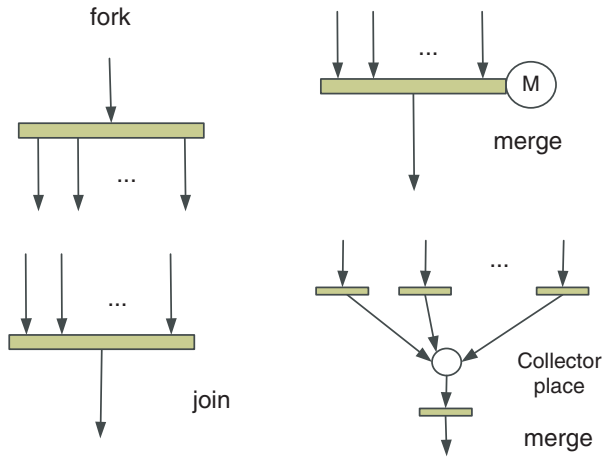


Fig. 7.1.5.4.1 Fork, Join, and Merge nodes. Fork and join are supported naturally by Petri Nets. The merge bar is a shorthand notation for a more complete arrangement shown with a token collector place

token entering one of the incoming edges of the merge node is propelled to the outgoing edge (if multiplicities are 1 and 1 at both sides). In other words, there is no conservation of tokens through fork and join nodes but there is conservation of tokens through the merge node.

7.1.5.5 Pseudoplaces. Pseudoplaces can be defined by developers, if needed, and provided that their meaning is unambiguous. As place is a unifying concept, it can thus be used to define Initial, Final, and Flow Final in Activity Diagram or Initial, Final in State Machine Diagram.

7.2 UML Diagrams Mapped into SEN

As the SEN is targeted to be an intermediate model before coding, it is therefore essential to demonstrate that it has the capacity of mapping all high-level constructs used in UML diagrams. We do not really make a full demonstration (that could be very boring for the reader) but will take most representative cases.

7.2.1 State Machine Diagram Mapped into SEN

Figure 7.2.1.1 illustrates the way the state machine diagram packs two general states and a transition between these two states. Each state is defined with its *On Entry* action, *Do* action, *On Exit* action. Transition is defined with its complete characteristics (trigger, guard, action). Actually, this definition overlooks all the timing of various actions involved. When the object receives the event that triggers the state change, a series of actions is scheduled within the state

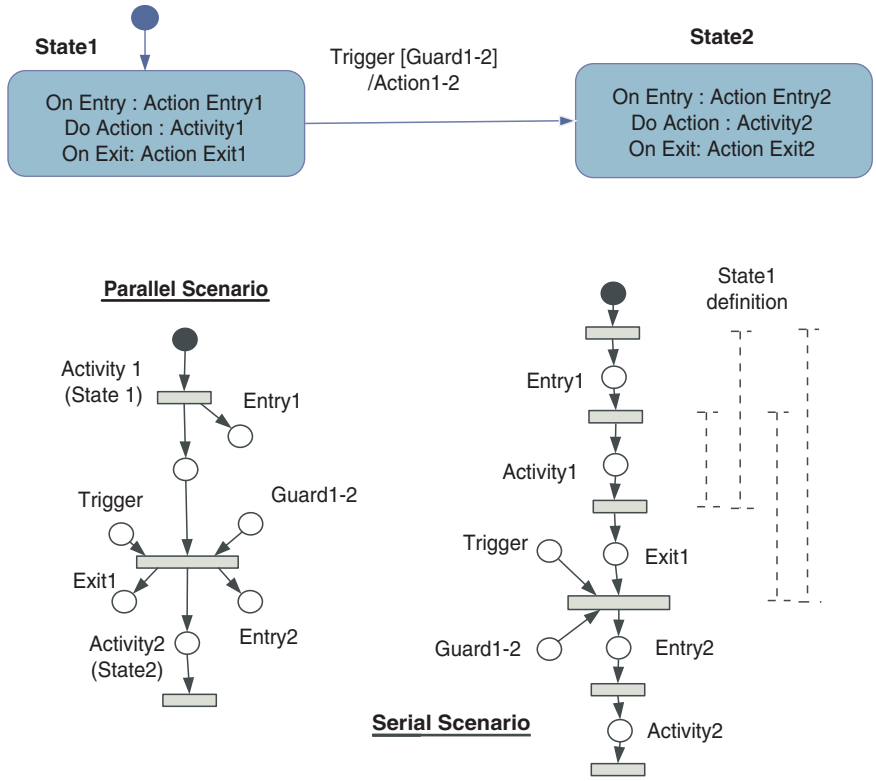


Fig. 7.2.1.1 Mapping of a state diagram into SEN. Many details overlooked on the timing of actions inside the state machine diagram appear now with the SEN. The actions Entry2, Action1–2, and Activity2 may start in parallel in the first scenario or they can be executed serially in the second scenario. Many other scenarios mixing parallel and serial actions are possible. The definition of a state is "elastic" in serial cases as we can include Entry or Exit actions

change: the *action1-2* for passing from the *state1* to *state2*, the *Exit1* action for leaving the *state1* and the *Entry2* action executed for entering the *state2*. These three actions may occur either in parallel, either sequentially (the order must be specified in this case) or possibly in a mixed mode with some time overlapping. Moreover, the notion of "state" has now been replaced by "state component", in our case, by *Activity1* and *Activity2* that defined *state1* and *state2*. But, when actions are executed serially or with some time overlap, it is not clear that states must be defined when *Action1-2* starts, when *Entry2* starts or when *Activity2* starts. Modelers that must translate the state diagram into code must therefore decide when a state begins and when it ends. But, the SEN can already highlight early all these timing subtleties. The SEN appears as a magnifying glass over a state diagram.

7.2.2 Activity Diagram Mapped into SEN

The mapping of an activity diagram towards a SEN is straightforward. All elementary actions and activities are translated into places. If they must be decomposed, the presence of SEN subnets allows future decomposition. So, mapping a control structure with control flow is somewhat trivial with a SEN. However, the presence of the object flow made explicit with UML 2 needs some comments.

Figure 7.2.2.1 takes an activity diagram in the Superstructure Book and converts it into SEN. In the SEN, the token in a place means many things. When a place is a state, a place owning a token is in this state. When a place is an action/activity, a place owning a token is currently executed. If the place is a condition, this condition is verified. If a place is an event, a place owning a token means that the event is occurring. A condition can be a result of an

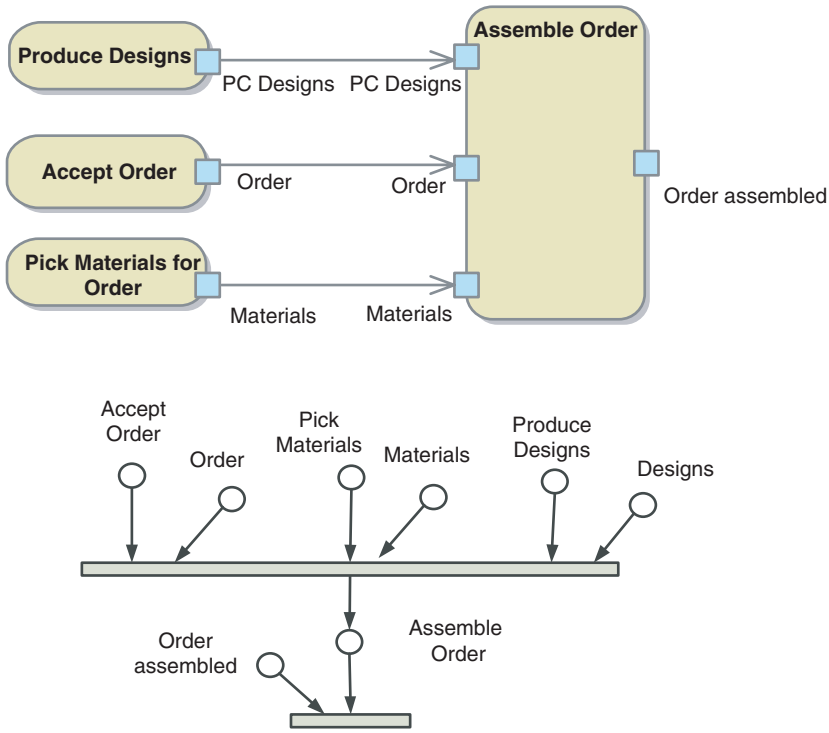


Fig. 7.2.2.1 Conversion of activity diagram with object flow into SEN. Example 12.126 of Superstructure Book. When each activity ends, it produces the necessary object so a token will be made available. We suppose that all activities must end before their tokens are made available. If needed, an end of activity sign (Fig. 7.1.5.1) can be added to make the interpretation more explicit

action/activity but this dependency between places are either not represented or are rather implicit (the same consideration is observed for an activity diagram, in which a guard that characterizes an outgoing edge may result from the execution of an upstream action located in a decision node or in an action node). So, a monochrome token cannot be really used to represent objects circulating in a SEN network. Following this logic, an object node is also a place, with a name that represents the object type. If a token is found in an object place, the interpretation is: object is available and made ready by the upstream activity and can now be used to feed as input to the next activity.

7.2.3 Sequence Diagram Mapped into SEN

If the state diagram and the activity diagram generally describes internal algorithms of systems or algorithms of many systems taken as a whole (previous activity diagram), intrinsically, sequence diagrams highlight the communication between objects, their interaction. As said earlier, arrows in the SEN syntax participate to the interpretation of the control flow, are used to support control structures. Dashed arrows exchanged between objects/systems are interobject communication. All arrows are of the same nature in a SEN, the distinction allows us to distinguish *intraobject interactions* or *interobject interactions*. In the activity diagram, some places are interdependent in the SEN and, normally arrows can be represented to highlight this dependence. As the interpretation is straightforward, they are generally removed. The Figure 7.2.3.1 shows this dependence. The same consideration happens for places in two different SEN, but now, the communication path must be made explicit to show the synchronization between many SEN places.

This synchronization concept with dashed arrows can be applied to other diagrams as statecharts (with regions) or parallel execution in the case of activity diagram (for instance, communication between places inside a fork/join region).

The interaction suite contains a second diagram, very closed to the sequence diagram, named *communication diagram*. This diagram shows all objects that collaborate together to perform a given task. Sequence of messages can be read with a numbering scheme affected to messages. This kind of diagram is halfway between structural and behavioral concepts. The structural part can be assumed by an *object diagram*. The behavioral part can be assumed by a sequence diagram in which the sequence is more explicit and easier to grasp. This diagram can be considered as a kind of "dynamic" object diagram.

7.2.4 Exception Handling

A token does not stay in SEN place unless there is a condition to force it to remain there (for instance a global *end of process* condition that forces all actions

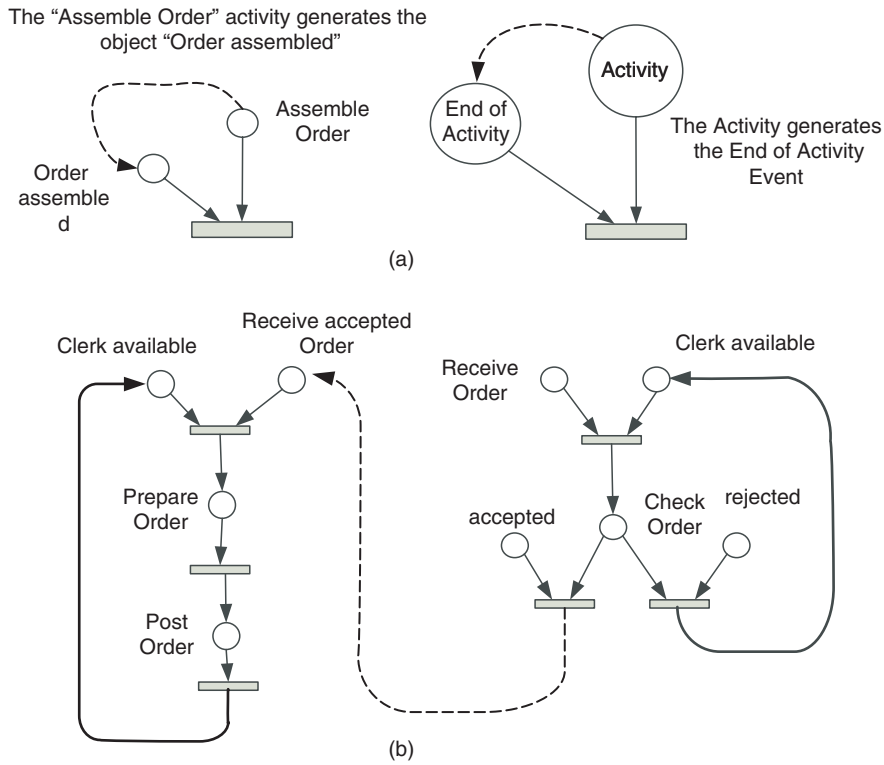


Fig. 7.2.3.1 Intraobject interactions and interobject interactions. (a) Shows the dependence of SEN places when interactions occur in the intra object mode. In (b), many SEN are synchronized through the same kind of interactions made explicit with dashed arrows. Intraobject interactions are generally not represented in SEN except if necessary

or activities to reach their end before exiting the place). Other conditions that retain token are, for instance, "next event not occurred", "condition not met", etc. on the bar connected to an outgoing edge.

When an interrupt occurs, it acts as a normal event on the bar connected to an outgoing edge. However, we must decide if this interrupt must follow the global policy (waiting for the action/activity to terminate) or force a precocious interruption of the currently executing task and entering an exception subnet. As the term "interrupt" means stopping a current process, it seems more normal to opt for the second possibility but it is not impossible that the first possibility still prevails in some situations. So, it is the burden of the modeler to decide on the appropriate scenario. If a global *end of process* condition is specified and we decide that the interrupt must stop the current activity, a lightning sign can be used to cancel this global condition locally.

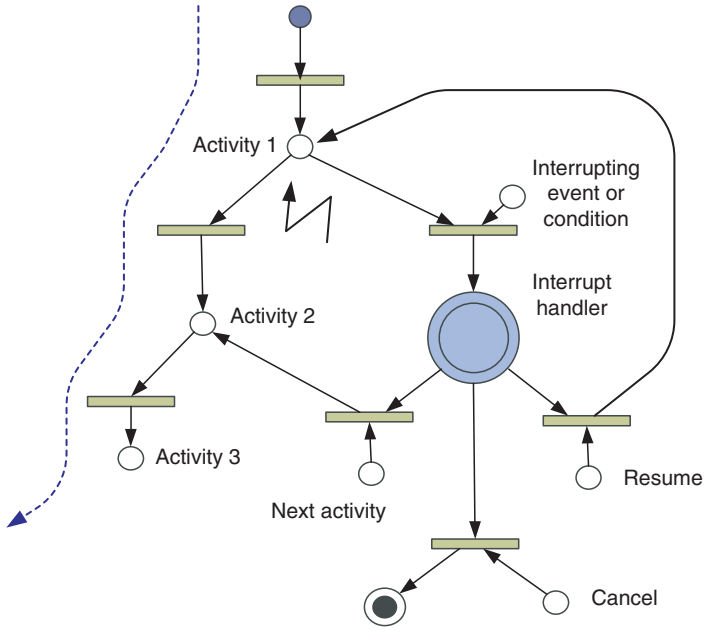


Fig. 7.2.4.1 Exception handling in a SEN. The exception is considered as a condition or an event that can force the place to stop its execution and go into an exception handler that is considered as a subnet

Figure 7.2.4.1 shows a standard way of representing an interrupt in a SEN. We suppose that globally, an *End of Process* condition is imposed on places. A lightning sign put on *Activity1* allows this activity to be interrupted by the *Interrupting Event or Condition*. Special processing after the interrupt will be defined in Interrupt handler that is supposed to return a token after ending its job. The token will have three choices: go to a *Cancel* state stopping everything, *Continue* with the next process, or *Return* to the interrupted process with other parameters.

7.2.5 Competition for Common Resource

If two persons need to print their job and the printer can print only one job at a time, the first job will monopolize the printer and the second job that arrive epsilon time after the first one must wait. A SEN can easily represent this case and many other cases of resource contention (Fig. 7.2.5.1).

7.3 Timing Constraints with SEN

For research on the use of Petri nets for controlled discrete Event Systems, please refer to survey article by Zhou and Dicesare (1993) and Holloway et al. (1997). Our goal in this direction is limited to the specification of time constraints

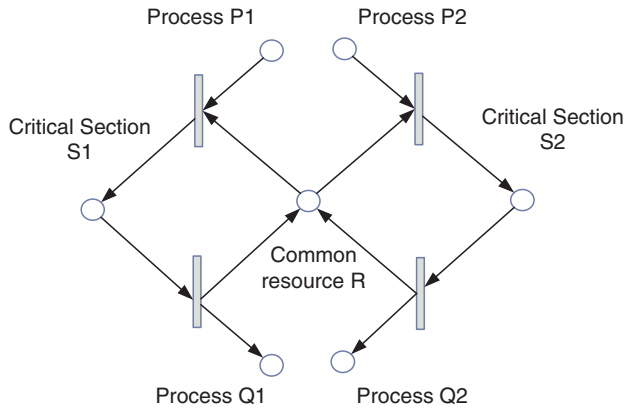


Fig. 7.2.5.1 Competition for a common resource

(Berthomieu and Diaz, 1991; Ghezzi et al., 1991; Tsai et al., 1995) accompanying the movement of the token. We suppose that, when all conditions on the bar are verified, the token will be propelled through the bar which is a primitive event with zero time duration. This attitude does not mean that the timing of the system is oversimplified. Actually, elementary actions are already detailed at each step to reach the most elementary action chunks (see the decomposition of UML state transition in Figure 7.2.1.1 which isolate various actions *Do, On Entry, On Exit* in the states and action accompanying the transition). Secondly, in a SEN, timing constraints are reported in places instead of arcs. When a timing constraint is satisfied, a token is generated in the corresponding place and the SEN evolves normally to the next state. If the timing constraint is not validated, the net must provide an exception to explain explicitly the way the system must handle such a case.

The dynamic analysis of a system can be done in two phases, the *functional analysis* and the *timing analysis*, in this order. We must assert the correctness of the functionality earlier without any timing constraint consideration. The timing analysis is then performed to assert the correctness of the timing behavior. This correctness is obtained if the system complies with the timing constraint specifications. We address to the first step of timing analysis: the *timing specification*.

> The timing analysis needs a reachability analysis. A place is reachable if it can get a token, even under restricted conditions. A SEN can be reachable in absence of timing constraint and not reachable after timing constraints are imposed. In complex cases, computer simulation can be used to test this reachability.

Abadi and Lamport (1994) mentioned that "the old fashioned methods handle real time by introducing a variable, which we call *now,* to represent time. This

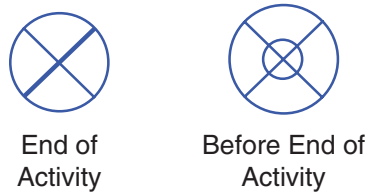


Fig. 7.3.0.1 Representation on the End of Activity (EOA) and Before End of Activity (BEOA) conditions. The EOA has been discussed in Figure 7.1.3.1. BEOA is the opposite logical condition, so a token is present in this place when an activity starts and disappears when it ends

idea is so simple and obvious that it seems hardly worth writing about, except that few people appear to be aware that it works in practice." Berthomieu and Diaz (1991) have analyzed concurrent systems whose behavior is dependent on explicit values of time. The number of scenarios for time specification is very large but fundamentally we can identify some current time constraints.

A process *duration* (or *end* variable if we suppose that this variable is reset each time a given process starts) can be bounded by a low limit and a high limit, for instance, $low \leq duration \leq high$ generates three cases to be considered. When process *duration* must exceed a low limit ($low \leq duration$) or when process *duration* must not exceed a high limit ($duration \leq high$), two cases have to be considered.

Another kind of specification relates a given point of time to another point of time (Fig. 7.3.0.2). For instance, if a process A must terminate before another process B, but their respective durations are not specified; Figure 7.3.0.1 elaborates a *Before End of Activity* (BEOA) condition that is the negation of *End of Activity* (EOA) condition already defined in Figure 7.1.5.1. Figure 7.3.0.3 shows an example of specification.

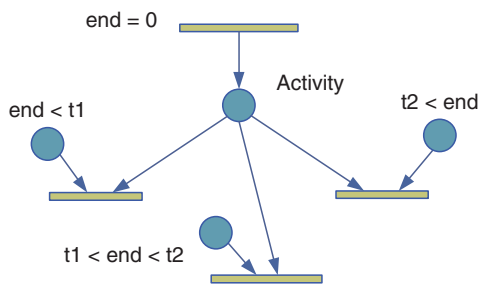


Fig. 7.3.0.2 Timing specification with variable "end". Timing limits can be specified with an end variable reset to zero when the activity starts. With two limits t1 and t2 (t1 < t2), the system may show three different behaviors considered in the SEN as three conditions

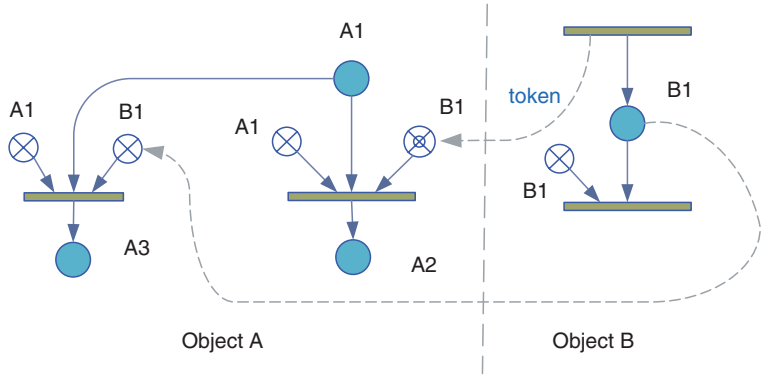


Fig. 7.3.0.3 Timing dependence between two objects A and B with End of Activity (EOA) and Before End of Activity (BEOA). Dashed arrows show the timing dependence of the two SEN but they are they can be omitted in real situations. The same name used for all the places is sufficient to connect these diagrams. An EOA place receives a token when an activity reaches its end. A BEOA must receive a token when the activity starts (when B1 receives a token, a second token is given to the BEOA named B1). When an activity ends, its corresponding BEOA disappears. If A1 finishes before B1, then A1 goes to A2. If A1 finishes after B1, then A1 goes to A3

7.4 Case Study with SEN

A Tamagotchi (or Tamaguchi) is a digital pet (virtual digital companion) created in 1996 by Ali Maita and sold by a Japanese Toy Manufacturer Bandai Co., Ltd. Hereafter, we describe an imaginary and simple toy that simulates the working principle of a Tamagotchi to study the dynamic behavior of this system with a SEN. Hereafter is the description of an oversimplified Tamagotchi (Table 7.4.0.1).

> When starting a Tamagotchi cycle, he enters first in his Happy state. But this state lasts only during TA seconds called Autonomy Time. At the expiration of TA, the Tamagotchi gets hungry and cries. A person must put the Tamagotchi at the table so he can eat. He stops crying and eats during TF or Feeding Time. At the end of TF, the Tamagotchi will cry again and we must get him off the table to avoid the overfeeding state. He will enter his Happy state and the Autonomy Time reapplies. In all cases, if the Tamagotchi cries more than TC or Critical Time, he will die.

We can analyze this pet and dissect the text with the methodology of text analysis already exposed for the RMSHC (Fig. 7.4.0.1).

Figure 7.4.0.2 is not the implementation model, but just the logical model explaining the overall operation of the system. Detailed operations setting various states of the Tamagotchi are not mentioned. Dashed arrows represent communication messages between SEN belonging to different objects. In this diagram,

Table 7.4.0.1 Text Content Analysis of the Tamagotchi specification

Step	*Initial text*	*UC concepts*	*Reformulation*
1	When starting a Tamagotchi	Actor/Object	Tamagotchi
1a	cycle, he enters first in his Happy	State/Activity	Stay happy
	state		
2	But this state lasts only during	Actor/Object	Timer
2a	TA seconds called Autonomy	UC/Action	Arm Timer with TA (Autonomy
	Time		Time)
2b		UC	Decrement TA
3	At the expiration of TA, the	Event	TA expired
	Tamagotchi gets hungry and cries	UC/State/Activity	Get hungry and cry
4	We must put the Tamagotchi at	Actor/Object	Person (who feeds the
	table so he can eat		Tamagotchi)
4a		UC/Action	Put the Tamagotchi at the table
4b		UC/Activity	(Tamagotchi) eat
5	He stops crying and eats during	UC/Action	Stop crying
5a	TF or Feeding Time	UC/Action	Arm Timer with TF
5b		UC/Activity	Decrement TF
6	At the end of TF, the Tamagotchi	Event	TF expired
6a	will cry again and we must get	UC/State/Activity	Get overfed and cry
6b	him off the table to avoid the	UC/Action	Get Tamagotchi off the table
	overfeeding state		
7	He will enter his Happy state as		(No new abstractions. Come
	and the Autonomy Time will		back to 1a)
	apply		
8	In all cases, if the Tamagotchi	Constraint	In all cases (when crying)
8a	cries more than TC or Critical	UC/Action	Arm Timer with TC (Critical
	Time, he will die		Time)
8b		UC/Activity	Decrement TC
8c		Event	TC expired
8d		UC/Action	Tamagotchi dies

we have voluntarily represented sub-SEN places of Person and Timer on the same diagram to show the real interactions but it is not necessary to do so in a real project.

As said earlier, Figures 7.4.0.2 and 7.4.0.3 build the first logical model of the Tamagotchi. This model explains and identifies all high-level operations and evident attributes (e.g. *Tamagotchi state* that is an *Enum* parameter taking the following values: *Happy, Hungry, Overfed, Died*). Now, if we try to simulate the Tamagotchi with a computer or if we want to make an electronic pet, a button will be added to allow the Person to interact with the Tamagotchi (Fig 7.4.0.4). The separation of class and operation diagrams is an advantage as we can store all

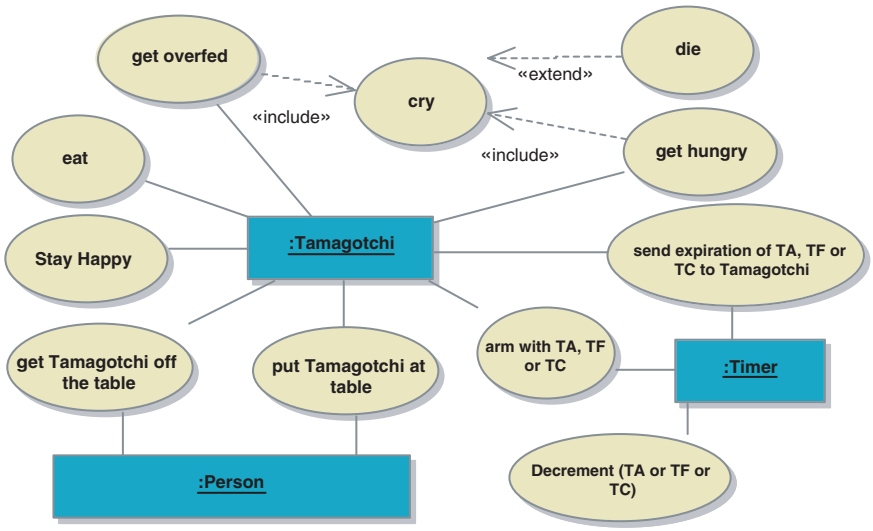


Fig. 7.4.0.1 Use Case (UC) diagram representing activities of the Tamagotchi, Person, and Timer. The object (or actor) Timer is part of the Tamagotchi but this fact is not explicit in the UC diagram. When getting hungry or overfed, the Tamagotchi cries so this action is included inside "*get hungry*" and "*get overfed*" use cases. The Tamagotchi can cry to death if he is not saved in time

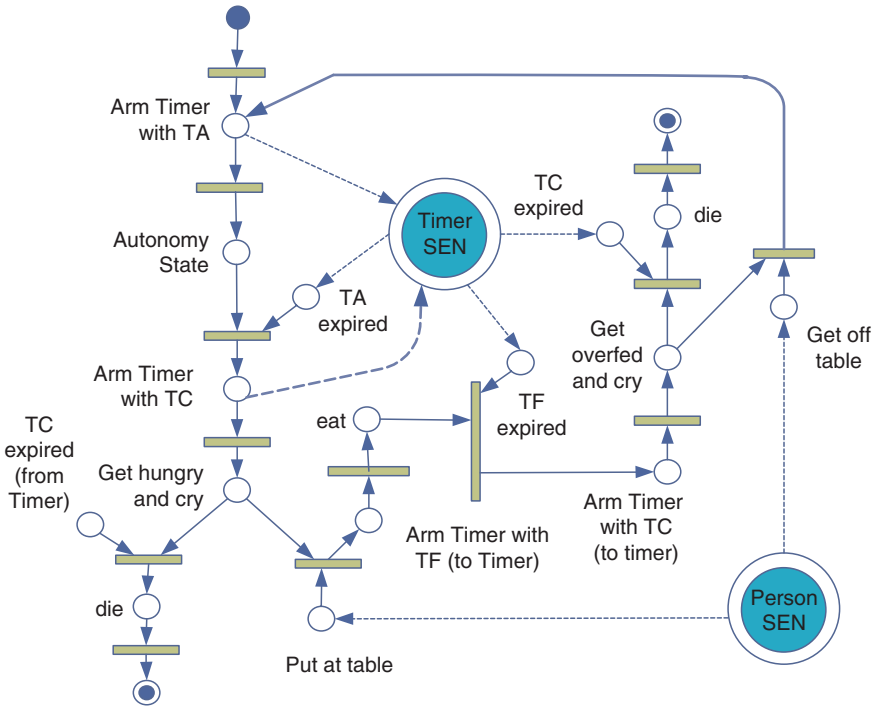


Fig. 7.4.0.2 SEN representing the state evolution of the Tamagotchi

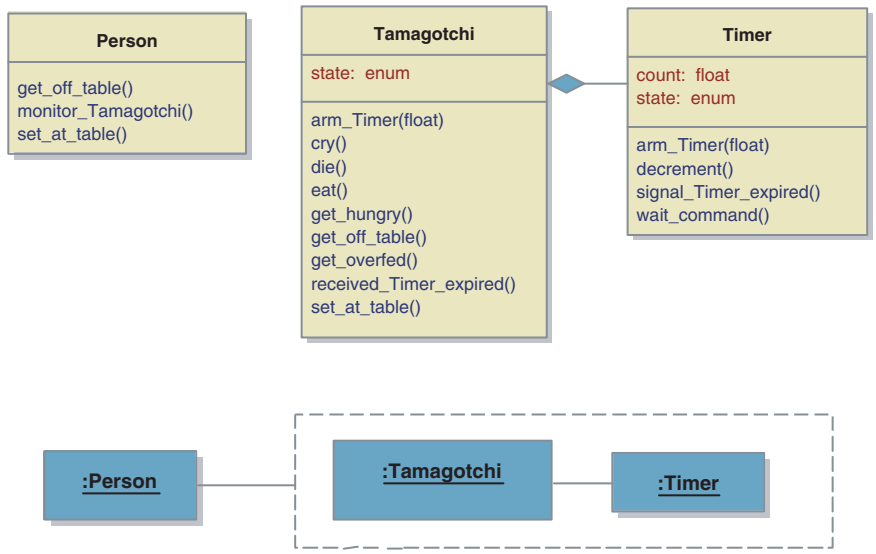


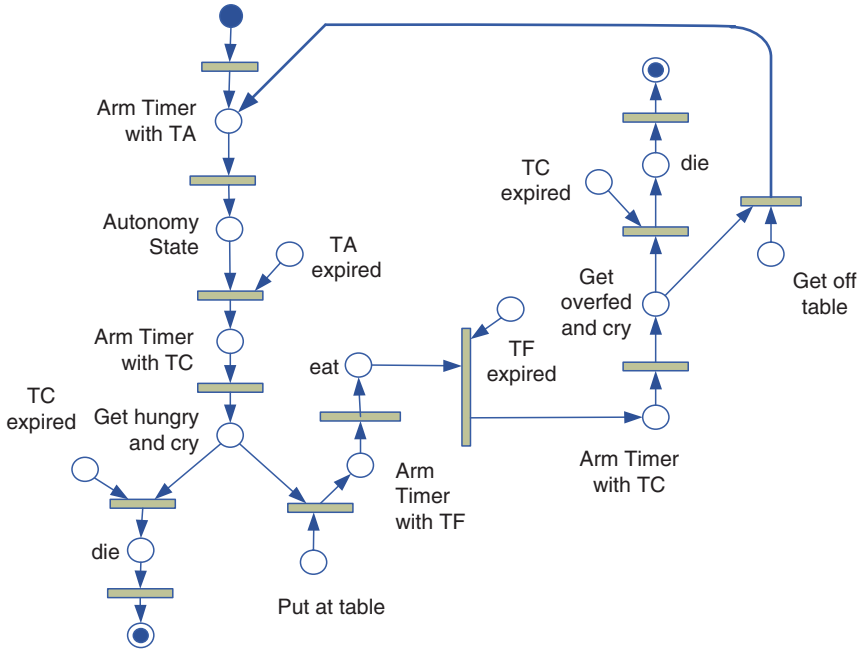
Fig. 7.4.0.3 Class and object diagrams of the Tamagotchi. In class diagram, classes are shown with their attributes, operations, and relationships. Object diagram shows communication links between the object *:Person* and the whole Tamagotchi system made of the *:Tamagotchi* himself and its internal biological *:Timer*. The model is still a logical model and does not contain any implementation details

classes of all applications inside the class package but instantiate several object system for various implementations of the Tamagotchi. For instance, when making an electronic pet, we can add a sound system to simulate various states of the Tamagotchi. New objects and new interactions are therefore concerns of subsequent models.

7.5 Safety-Critical Systems

Wikipedia.org defines *safety-critical system* as *life-critical system* and as "a system whose failure or malfunction may result in death or serious injury to people, loss or severe damage to equipment, or environmental harm." By mentioning "damage to equipment," human life may be implied indirectly and the effect cannot be immediate (environment harm). So, this concept involves a wider class of applications than expected. Safety is always considered with respect to the whole system, including software, hardware (electronic, mechanical, biological, chemical, etc.), users, and operators.

Traditionally, safety critical software has been associated with embedded, embarked, and real-time systems (military, transport, communication, medical, security, nuclear power, etc.). Its scope has expanded recently into many other contexts (internet, household apparatus, etc.). An aircraft with hundreds of passengers is totally dependent upon fully operational and safety-critical systems.



Note 1: All timer expiration signals come from the Timer and all Arming Actions are directed to the Timer
Note 2: "Put at table" and "Get off table" are operations executed by Person object.

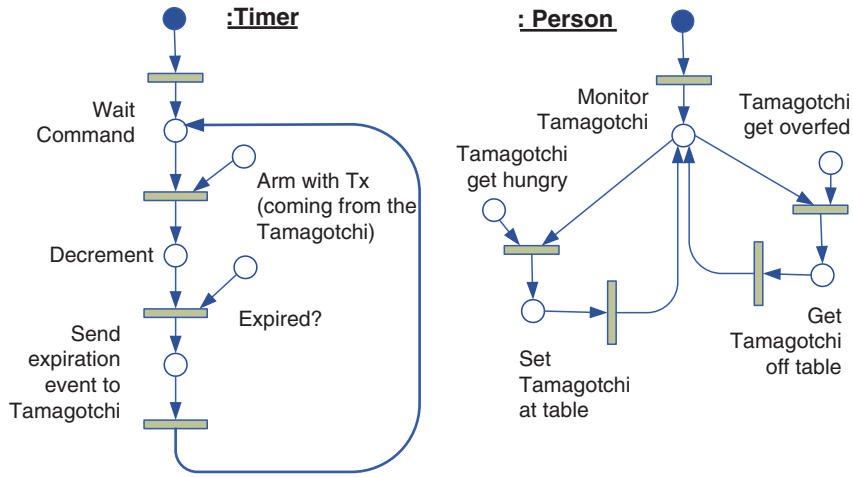


Fig. 7.4.0.4 SEN versions of the Tamagotchi application with separate SEN. Cross-interactions between objects are not shown but they are parts of class and object diagrams

Railway signaling systems preventing trains from colliding, or more generally traffic regulation, are real time and safety critical. Medical systems are safety critical as they are directly responsible for human life. Recent accidents of collapsing buildings in 2001 have thrown civil engineering structures into safety-critical class systems. Errors in software calculation or software simulation could be fatal. Many systems in a car are now software based (ABS brakes) and could potentially fail and create accidents.

The questions when developing safety-critical systems are "Can we present hazards and reduce their occurrence?" and "Can we verify that a given developed system is safe?" The vast majority of software safety is entirely in the hands and conscience of the software developers and suppliers. The first step in developing a system is performing a preliminary hazard analysis, to determine whether the system could present a hazard to safety. If the answer is yes, we must conduct a more detailed hazard analysis and safety engineering (Leveson and Stolzy, 1986; Villemeur, 1991; Isaksen et al., 1996; Lutz, 2000).

From these articles, a *hazard* is a potentially dangerous situation that can cause an accident. A hazard may come from an internal error or fault. A *failure* is seen as an event that leads to the accident. The *risk* is associated to the probability of the hazard.

Security is a concept related to the confidentiality of a system, malicious interference, unauthorized access, and privacy. In internet communication systems, there is a close relationship between security and safety; a hole in the security is a hazard for safety.

When discussing safety of software, software can become important when it is used to control potentially dangerous hardware. Moreover, safety can be dependent on human factors (a bad man–machine interface can cause misinterpretation and hazard; a patient could be given the wrong medication even with the best hospital database, so control mechanisms must be implemented to avoid even human error). Even if the software is bullet-proof and it can be established that there is no software error when an accident occurs, a badly designed system is implicitly a faulty design as it can create potentially hazardous situations for normal humans with normal reactions. The large number of states found in software usually makes exhaustive testing impossible. There is no proportionality between the hazard and the disaster it can cause. A very small omission or error may have a strong impact on safety. So, software safety is an important issue and no details can be forgotten in a development if safety is the main concern. In a multidisciplinary system, it would be more difficult to assert the safety factor when dealing with various components coming from many disciplines. This fact pleads for a uniform way of analyzing and designing a whole system (that is the subject of this book) so all factors can be judged and analyzed on

the same conceptual and logical basis regarding safety independently from the discipline a piece of hardware or software. The safety consideration starts at the beginning of the development process (specification) then takes into account all the design, implementation, and testing. Special tools must be developed to assert that safety has been observed correctly. A well-designed system is the first step towards safety.

7.5.1 State Space Search with Combinatorial Method: Image Attribute Methodology

Let us consider the example of the Tamagotchi. States of the Tamagotchi have been proposed previously without any systematic methodology. The developer thinks that the electronic pet can be in the five states:

1. Normal and Happy (Normal = Happy)
2. Get hungry and Cry
3. Eating when been sit at table
4. Get overfed and Cry at table
5. Died

He then proposes the algorithm of Figure 7.4.0.3. The questions are "Are all states enumerated?" "Are there forgotten states?" We cannot demonstrate that the states enumerated are all possible states that can be observed with the Tamagotchi.

If we consider the following internal and binary variables of the Tamagotchi: *happy, hungry, cry, eat, be at table, die*, we get six binary variables with a possibility of $2^6 = 64$ combinations. So, as a first observation, the behavior of a system depends upon the *change of its internal variables*.

But, the Tamagotchi interacts with an internal Timer and an external Person. So, his state or his state change is influenced by events sent by other objects (e.g. Timer expiration event) or the actions performed by other objects on him (when the Person puts the Tamagotchi at the Table, he stops crying and eats). As a second observation, the behavior of a system depends upon *inputs* to this system (Fig. 7.5.1.1).

More surprising, when analyzing many systems, we observe that the state of an object can be conditioned by the actions this object performed on its environment or the events it creates to influence objects in its neighborhood. If the Person performs an action on the Tamagotchi, he/she supposes it will change its behavior later and stop crying, so the person has internally an "image" of the expected behavior of the Tamagotchi. If this image does not correspond to

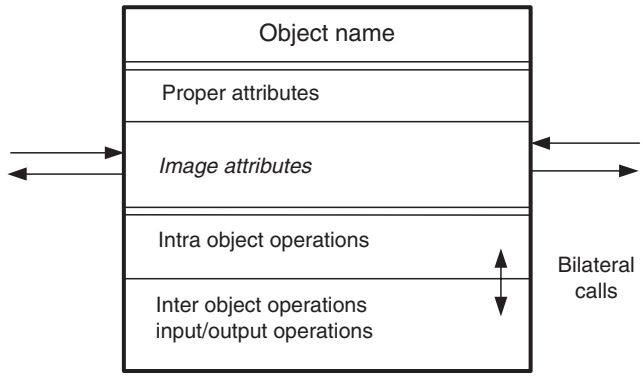


Fig. 7.5.1.1 Attributes and image attributes of an object. Proper attributes are those that describe the behavior of the object, e.g. state changes, within intraobject operations. Image attributes are those that object models its environment that impact on its proper behavior. If the object needs to communicate or exchange messages with its environment, image attributes can be built from input/output types and values. An object has internal resources (input operations) to capture all incoming events, can change its state if it is subjected to external actions. It also has internal resources (output operations) to act on the environment or send messages/events to surrounding objects. It has image attributes to compare the actual behavior of other objects to its proper references

the reality (the pet does not stop crying), in this case, more appropriate actions could be programmed. In the context of this problem, the person does not have any means to afford more appropriate actions as the person is designed with only two actions (put the pet at the table or get the pet off the table). Many real-time systems have a richer set of actions and algorithms to deal with real situations. A closed loop can therefore be established between the controlling and the controlled objects so the actual behavior can be adjusted in real time to the image of the "correct behavior." For an open loop system, the behavior of a system is effectively dependent on the actions performed blindly, for instance, if we make a redirection on a phone, we suppose it will redirect all calls and we adjust our behavior accordingly (even if this redirection is not effective). So, as a third observation, the behavior of a system depends upon *outputs* from this system.

To study a system and its behavior, it is necessary to enumerate all attributes, proper and image attributes, determine their type, and their variation interval. For analog variables, the simplest case occurs when an analog variable can be made discrete. For instance, the comfort temperature of our home is between 20°C and 24°C, under 20°C, the temperature is qualified as cold and above 24°C, the temperature is qualified as hot. Most analog systems present some threshold values settling the accepted tolerance even if the analog variables

cannot really be made discrete. Figure 7.5.1.1 summarizes attributes and image attributes of an object.

When the system evolves from one state to the next, many variables may change at the same time. However, if the system is decomposed thoroughly and the time interval is expanded to see what happened more precisely, variables change one after the other. When programming, if several instructions must be scheduled at the same time, we must often decide on the order to issue them as nothing is really instantaneous or simultaneous when time expands. For instance, when receiving an expiration signal of the autonomy time TA, the three following actions seen as simultaneous in the logical model is sequenced in a simulated system with the following order (we suppose the time required to execute them is negligible if compared with the time scale of the application).

Set hungry attribute to ON
Set cry attribute to ON
Perform Cry() operation

The next step is making an inventory of all proper and image attributes of the Tamagotchi. As explained in the theoretical part of the UML, an association between classes is translated into links between objects. Very often, in an object diagram, a unique link is drawn between two objects to mean that they can communicate. When analyzing with the *Attribute Image method*, each component (input or output) of the amalgamated link must be separated and attached to a separate image attribute.

Table 7.5.1.1 lists nine Boolean or enum variables with a maximum combination value of 3072. This astronomical value has in fact few effects on the way we set up the method as a lot of combinations are impossible. For instance, if the Tamagotchi dies, all values or the remaining variables are insignificant, or the Tamagotchi cannot be simultaneously happy and crying.

Instead of enumerating all the variables in a disordered way, we can explore them systematically by building successive real and functional sequences with a SEN diagram as shown in the Table of sequences of Figure 7.5.1.2. These sequences, packed together, will constitute the global algorithm of the Tamagotchi. Unsuspected or transient states can therefore be evidenced.

As the number of variables is important, we make three vectors, the first one with an ordered set of regular attributes and others with image attributes of their respective object to save space in Table 7.5.1.2. Each line of this table corresponds to a fundamental abstraction (action, activity, event, state, etc.) of the dynamic view.

When the table of sequences contains the same set of values for all attributes, but the interpretation is first an action then later a state (e.g. 21 and

Table 7.5.1.1 Table of attributes and image attributes of the Tamagotchi. Six are proper to the Tamagotchi, two enum variables connect the Tamagotchi to its internal Timer object, and one enum variable connects the Tamagotchi to the Peron object

N	*Variable*	*Type*	*Values*	*Comments*
Tamagotchi attributes				
1	Happy	Boolean	1–0	Normal state when entering
2	Hungry	Boolean	1–0	
3	Cry	Boolean	1–0	
4	Be at table	Boolean	1–0	
5	Eat	Boolean	1–0	
6	Die	Boolean	1–0	
Tamagotchi image attributes (facing Timer object)				
10	Arm Timer with T_x (output)	Enum	Idle, TA, TF, TC	Idle is a state meaning that the Tamagotchi issues no command
11	Expired signal (input)	Enum	Idle, TAX, TFX, TCX	Idle, TA expired, TF expired, TC expired
Tamagotchi image attributes (facing Person object)				
20	Act on the Tamagotchi (input)	Enum	Idle, PAT, GOT	PAT: put at table, GOT: get off table
Total number of potential combinations: $2^6 \times 4 \times 4 \times 3 = 3072$				

21a; 31 and 31a), same numbers with different lettering scheme were used to account for this particularity. The explanation is "when entering a state, some actions are executed to set values inside the state and these values stay unchanged."

As loops are unavoidable in the table of sequence, the same point must be identified with the same number (without parentheses when the line is defined for the first time, with parentheses later, e.g. 4, 6a, and 10a). They correspond to the same place on a SEN diagram. When many elementary actions occur (e.g. 8, 21, 31, etc.), they are considered in this logical model as "without duration" and executed in parallel unless there is some logical dependency between these elementary actions. Figure 7.5.1.2 is a new SEN obtained after this combinatorial analysis with the image attribute method. Now, we can claim that there are no forgotten states or actions and the design is safe at this point.

7.5.2 Developing Safety-Critical Systems

Despite the large number of potential combinations, the *method of image attributes* is deterministic, combinatorial, and can be applied surprisingly in

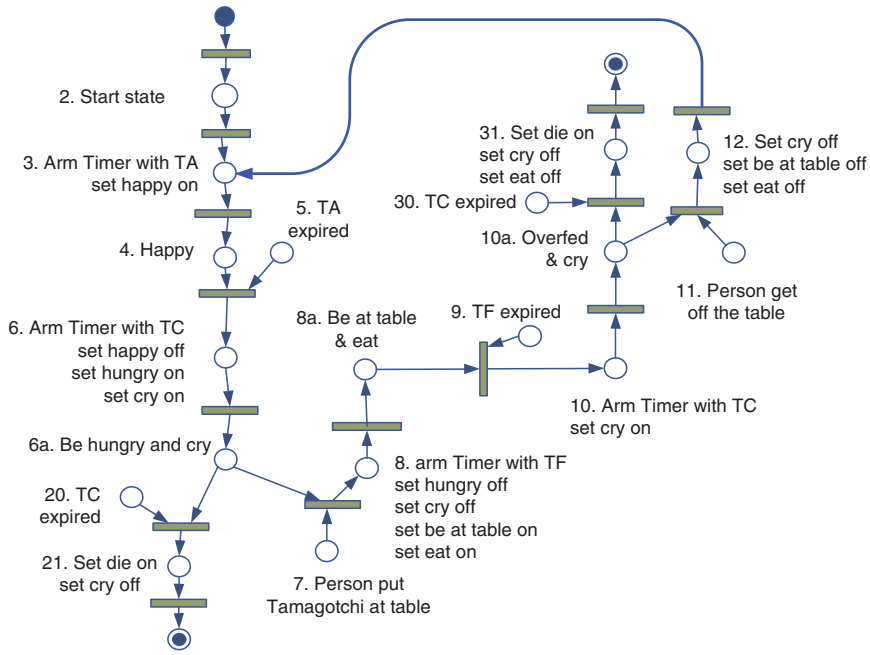


Fig. 7.5.1.2 SEN of the Tamagotchi established after an extensive study with the method of image attributes. This diagram is more precise then Figure 7.4.0.3 executed without any systematic investigation. Notice also the forgotten action labeled #12 undetected in the previous SEN. From the safety-critical viewpoint, we need just a minuscule error to get a big accident. (21a) and (31a) make a distinction how the Tamagotchi died (hungry or overfed)

several apparently complex situations. Actually, the method explores only sequences that need to be implemented. Along these sequences, all attributes and image attributes are highlighted, their values identified, and their effects in the system evaluated. By examining all sequences (to be implemented) obtained from evolutions of their attribute values, minor details and most elementary actions can be identified. Moreover, interactions of this object with surrounding objects can be evidenced by image attributes. Designing errors are naturally avoided as all elementary evolutional steps can be seriously tracked and ordered. If conducted correctly, it eliminates virtually all hazards coming from forgotten states in a design if this design can be studied in a deterministic way. Moreover, this method already offers testing steps to attest the functionalities of a system.

When developing a complex system, the problem is not to inventory all the possible scenarios but, owing to the development cost, only very well-identified functionalities and their corresponding scenarios (or sequences) need to be designed. Subsequent functionalities can be added at a later stage if needed. The system can be made safe if it cannot reach any state not listed in the table

Table 7.5.1.2 Table of Sequences built from variations of state values of Tamagotchi regular attributes and image attributes

N	*Concept identification*	*Value sets [happy, hungry, cry, be at table, eat, die] [Arm Timer T_x, expired signal] [action of Person on Tamagotchi]*	*Comments or Names*
Main sequence			
1	Initial state	[all undefined]	Starting point
2	State	[0, 0, 0, 0, 0, 0] [idle, idle] [idle]	Initialization
3	Action	[1, 0, 0, 0, 0, 0] [TA, idle] [idle]	Arm Timer with TA (transient action so TA is removed immediately) Set happy on
4	State	[1, 0, 0, 0, 0, 0] [idle, idle] [idle]	Tamagotchi happy
5	Event	[1, 0, 0, 0, 0, 0] [idle, TAX] [idle]	Event TA expired received
6	Action	[0, 1, 1, 0, 0, 0] [TC, TAX] [idle]	Arm critical time TC Set happy off Set hungry on Set cry on
6a	State	[0, 1, 1, 0, 0, 0] [idle, idle] [idle]	Be hungry and cry
7	Event	[0, 1, 1, 0, 0, 0] [idle, idle] [PAT]	Person puts Tamagotchi at table
8	Action	[0, 0, 0, 1, 1, 0] [TF, idle] [idle]	Arm Timer with TF Set hungry off Set cry off Set be at table on Set eat on
8a	State	[0, 0, 0, 1, 1, 0] [idle, idle] [idle]	Be at table and eat
9	Event	[0, 0, 0, 1, 1, 0] [idle, TFX] [idle]	Event TF expired received
10	Action	[0, 0, 1, 1, 1, 0] [TC, idle] [idle]	Arm Timer with TC set cry on (still eating to overfed)
10a	State	[0, 0, 1, 1, 1, 0] [idle, idle] [idle]	Overfed and cry
11	Event	[0, 0, 1, 1, 1, 0] [idle, idle] [GOT]	Person gets Tamagotchi off the table
12	Action	[1, 0, 0, 0, 0, 0] [TA, idle] [idle]	Arm Timer with TA Set cry off Set be at table off Set eat off Set happy on
(4)	State	[1, 0, 0, 0, 0, 0] [idle, idle] [idle]	(Come back to state 4)
Cry to death, been hungry			
(6a)	State	[0, 1, 1, 0, 0, 0][idle, idle] [idle]	Be hungry and cry
20	Event	[0, 1, 1, 0, 0, 0] [idle, TCX] [idle]	Event TC expired received

(*Cont.*)

Table 7.5.1.2 (*Continued*)

N	*Concept identification*	*Value sets* *[happy, hungry, cry, be at table, eat, die]* *[Arm Timer T_x, expired signal]* *[action of Person on Tamagotchi]*	*Comments or Names*
21	Action	[0, 1, 0, 0, 0, 1] [idle, idle] [idle]	Set die on Set cry off
21a	State	[0, 1, 0, 0, 0, 1] [idle, idle] [idle]	Died been hungry
Cry to death, been overfed			
(10a)	State	[0, 0, 1, 1, 1, 0] [idle, idle] [idle]	Overfed and cry
30	Event	[0, 0, 1, 1, 1, 0] [idle, TCX] [idle]	Event TC expired received
31	Action	[0, 0, 0, 1, 0, 1] [idle, idle] [idle]	Set die on Set cry off Set eat off
31a	State	[0, 0, 0, 1, 0, 1] [idle, idle] [idle]	Died been overfed

of sequences. So, it must be designed to fulfill a given set of functionalities (that could be extended later) and, at the same time, it cannot deviate from a set of paths already drawn in the execution tree. In the vocabulary of Petri nets, only a given set of places are reachable, all other places are forbidden because they are not studied and verified. Hazards or potentially dangerous situations are eliminated with this development attitude.

> For instance, in electronic engineering field, if a processor must not accept interruptions and there is no exception handler written for managing interruptions, hardware interrupts must not be left uncontrolled. Their voltage must be fixed to some secure value and the Enable Interrupt bit must be masked to avoid any hazard that could come from unexpected interrupts (coming from electric perturbations or strong electric or magnetic fields). When analyzing the working model of a processor, all these factors are listed as attributes and they will draw the attention of the developer.

7.5.3 Method of Image Attributes Applied to Human and Social Organizations

There are many subtleties on the way we search the image attribute. If we take this term at its original meaning, "image" does not mean reality but an "opinion" of this reality. In real-time systems, image is reality since we are an external observer, we master all the objects in the system and their behaviors are generally deterministic. The question is "could this method be applied to human and social organizations?"

The behavior of an object depends upon the image of the environment (surrounding objects) that this object is building in its microworld. Very often, the image we forge about other people or things does not agree with reality. This image may be imperfect, oversimplified, or biased. Moreover, if an

object has some form of intelligence, the image it makes of itself may not correspond to the real and objective image (that is never reachable in fact). Images are only "projections." Even so, we make projections of our proper personalities.

There is a difference in the way the images are constructed in human organizations and real-time systems. In a constellation of real-time objects devoted to accomplish high-level tasks, we search only image attributes of the first neighbors of a given object, searching their interactions at first level of interactions. This attitude is sufficient enough for many problems. In human organizations, the images go beyond the frontier of this first neighborhood. In human systems, we are accustomed to make images of everything, to judge everything and the microworld that conditions our behavior, which often oversteps our bounds. For good planning, the image of a good manager is that of someone having an accurate image on everything (human and situations). "Is it really useful or not?" We do not want to explore it yet.

> Image attributes are fundamental for deploying an advertising campaign for a company. The type of image this company wishes to convey should be elaborated from an understanding of its current image and its competitors' images. Using a series of "image attributes" (low prices, courtesy, high tech, fast delivery, etc.), the company calculates the relative importance of each attribute by establishing a mathematical model between these attributes and their impacts.

Images induce perception and dictate behaviors. While reasoning on human organizations, decision-making software can still use object technologies in a creative way by introducing a modeling concept based on the *method of image attributes* explained for real-time systems. The behavior of an object or agent is not dependent only on its original characteristics. Its behavior is highly conditioned by the interaction network it maintains with surrounding objects. To fully quantify these interactions, image attributes can be created inside each object showing how it models its microworld. If we can put values inside attributes, or at least enumerate them, computation models could then be built to study and anticipate object/agent reactions. From the object viewpoint, there is a departure from the way object technology usually handles objects. It is a really interesting research issue. A class must now have multiple compartments, both for attributes and operations. Attributes proper to an object will occupy only one compartment. Others are reserved to model the image it builds on other objects. In the simplest case, we have only the first neighbors interacting directly with this object. In a complex case, we must add more than the first neighbors. Operations available in the object are now dependent on attributes and image attributes and the behavior of the object can now be studied as usual considering that image attributes acts on the object in the same way.

This suggestion unifies the way the behavior of an object is studied and modeled with the UML and object tools. From the outside, an object/agent taken in a social or human context has the same "presentation" and characteristics as all objects encountered in programming environment. They have attributes, can work with operations, and have states. The difference is not seen on the presentation but by the way this object has been designed internally. It contains information on itself and information on the whole system as its second nature.