M2P GLRE

Génie Logiciel, logiciels Répartis et Embarqués

Modélisation de comportements de système en UML

Z. Mammeri

Dans le contexte de modélisation avec UML, cinq types de diagrammes peuvent être utilisés pour modéliser la partie comportement d'un système : diagrammes de cas d'utilisation, diagrammes de séquence, diagrammes de collaboration, diagrammes d'activités et diagrammes d'états-transitions. A ces cinq classes de diagrammes connus dans les versions UML 1.x, il faut ajouter les diagrammes de temps introduits dans UML 2.0

Les diagrammes de collaboration (appelés diagrammes de communication dans UML 2.0) et diagrammes de séquence sont équivalents (isomorphes). Pour la modélisation des systèmes temps réel, les diagrammes de séquence sont plus appropriés, car ils illustrent bien la chronologie des interactions. C'est la raison pour laquelle nous n'allons pas étudier les diagrammes de collaboration dans ce document.

La sémantique d'exécution de UML2 repose sur le modèle de causalité simple suivant : les objets répondent à des messages générés par des objets exécutant des actions de communication. La réception d'un message par un objet déclenche l'exécution d'un comportement particulier associé au message reçu.

1. Sur les objets

La qualité de modélisation de système à l'aide d'UML dépend de la maîtrise des concepts UML et de leur utilisation judicieuse. Il est évident que cette maîtrise est un processus long qui vient avec l'expérience, la répétition des modèles, l'amélioration des modèles déjà réalisés, la remise en cause de soi…

1.1. Identification des objets

C'est une phase importante dans la modélisation. Les objets retenus lors de cette phase vont conditionner le reste de la modélisation (sa clarté notamment).

Il n'y a pas de 'théorie' pour identifier les objets pertinents pour réaliser un modèle. Différents travaux ont été publiés pour donner des conseils pour bien élaborer la liste des objets. Voici une liste non exhaustive de ces conseils :

1. souligner les noms : opération utile pour identifier les objets décrits dans le cahier des charges.
2. Identifier les objets du monde réel (bouton d'ascenseur, cabine, …. par exemple)
3. Identifier les équipements physiques (capteurs, actionneurs, appareil de contrôle, appareil d'affichage…)
4. Identifier les objets actifs qui génèrent les événements ou des messages, qui exécutent les actions,
5. Identifier les objets passifs, ceux qui rendent des services à la demande mais qui ne déclenchent aucune activé ou message à leur propre initiative.
6. Identifier les messages et flux d'information entre objets.
7. Identifier les concepts clés (par exemple dans le cas d'un ascenseur on a : déplacement ascenseur, demande d'étage, ouverture de porte…) pour pouvoir modéliser les objets.
8. Identifier les transactions entre objets.
9. Identifier les informations persistantes (qui deviennent ensuite des attributs d'objets)
10. Identifier les éléments d'interface (boutons, icônes, menus, fonts…)
11. Identifier les éléments qui permettent à l'utilisateur de contrôler le fonctionnement du système.
12. Utiliser des scénarii pour réaliser les identifications citées précédemment.

Le processus d'identification des objets n'est pas linéaire. Il s'effectue par raffinement et retour arrière : extension ou suppression des éléments déjà identifiés, ajout de nouveaux éléments… Il faut boucler sur les opérations des éléments constituant les objets jusqu'à ce que les objets obtenus semblent répondre aux besoins. Il ne faut pas s'étonner s'il faut revenir plus tard sur les objets identifiés pour les améliorer, voire pour les supprimer, les renommer…

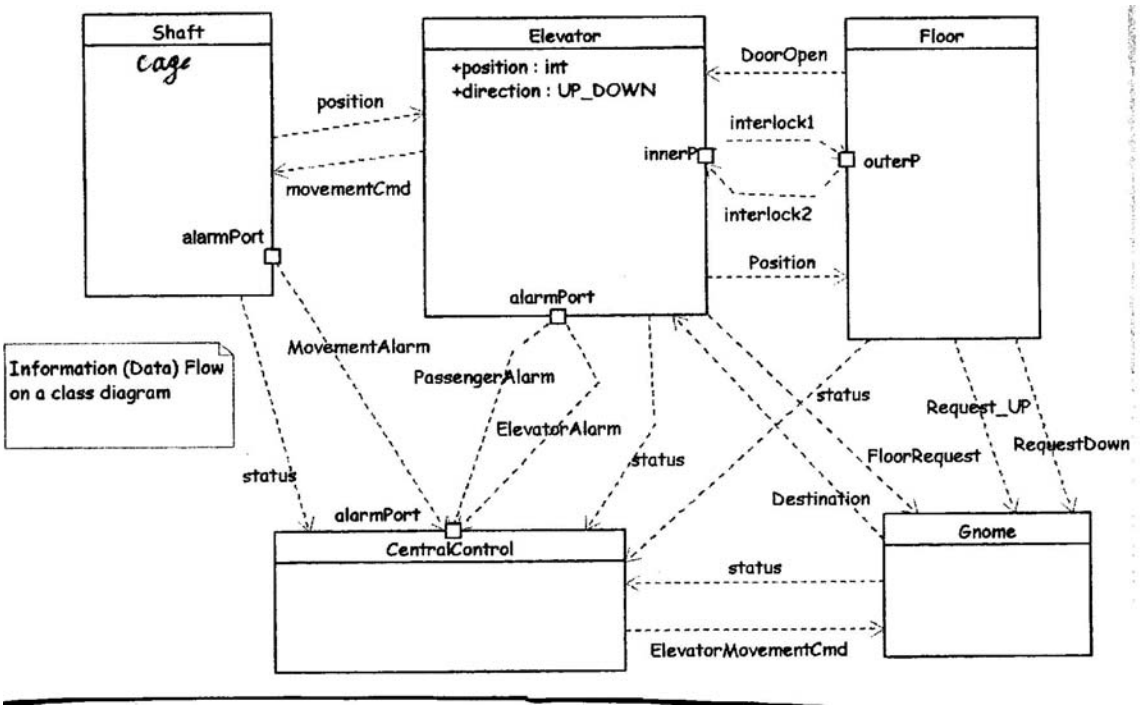

Fig. 1. Exemple de classe d'objets correspondant à la modélisation d'un ascenseur

1.2. Abstraction dans le processus de modélisation

Un objet est une abstraction. On sait apprécier un modèle bien fait (comme l'exemple ci-dessous), mais on ne sait pas dire comment y parvenir de manière méthodique. Tout vient par l'expérience et la remise en cause de soi.

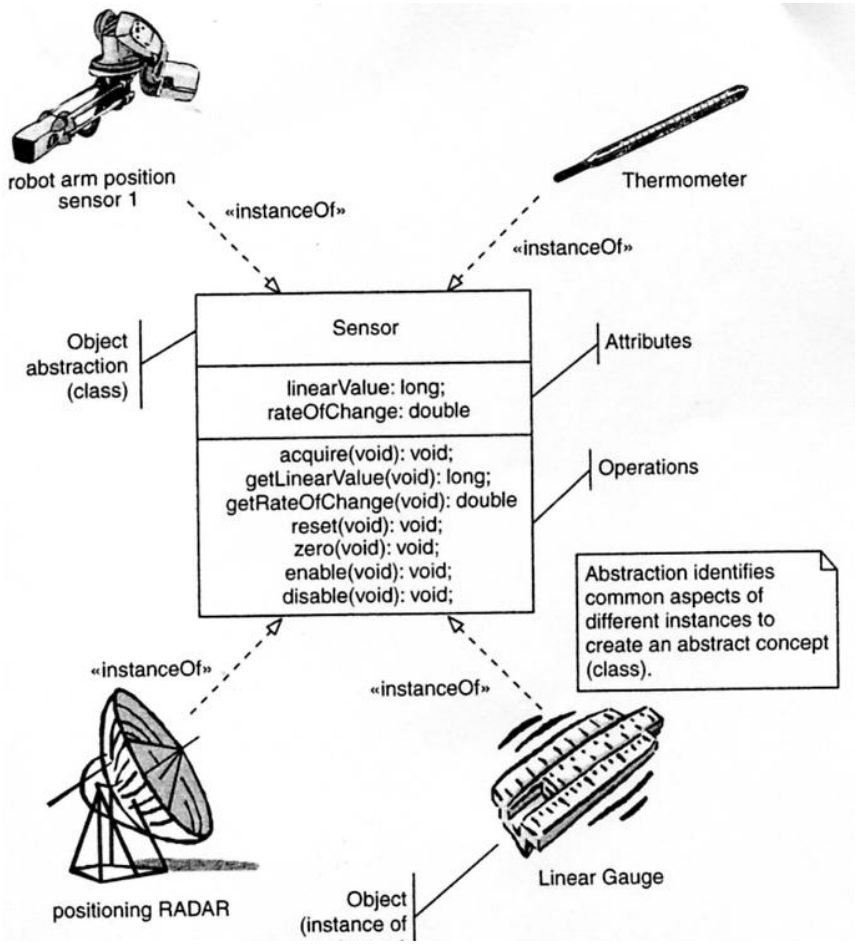

Fig 2. Exemple d'abstraction de la notion de capteur

2. Classes actives, événements, temps, interactions

2.1. Classes actives, processus, *threads*

Dans un système, plusieurs choses peuvent se passer en parallèle, on dit qu'il y a plusieurs flots de contrôle dans le système. En UML, on représente chaque flot de contrôle indépendant comme un objet actif. Un **objet actif** est un objet est représenté un processus ou un thread qui peut lancer une activité de contrôle. Un **processus** est un flot *lourd* qui peut s'exécuter en concurrence avec d'autres processus. Un **thread** est un flot *léger* qui peut s'exécuter en concurrence avec d'autres threads à l'intérieur d'un même processus. Un processus s'exécute dans un espace qui lui est réservé. Les threads d'un même processus partagent un même espace d'exécution.

Une **classe active** est une classe dont les instances sont des objets actifs. Une classe normale (dite aussi classe passive) ne met pas de flot en application et ne peut pas par elle-même déclencher d'activité de contrôle. Les processus et les threads sont représentés par des classes actives stéréotypées. Ils peuvent apparaître aussi comme des séquences dans les diagrammes d'interaction. Les classes actives ont les mêmes propriétés que les autres classes ; elles peuvent participer à des relations de dépendance, de généralisation et d'association.

Les mécanismes d'extensibilité s'appliquent aussi aux classes actives. UML définit deux stéréotypes standards : `process` (spécifie un flot lourd) et `thread` (spécifie un flot léger).

Dans un système qui comporte des objets actifs et des objets passifs, il existe quatre combinaisons d'interaction :

- Un message peut être transmis entre deux objets passifs. Une telle interaction équivaut à appeler une opération.
- Un message peut être transmis entre deux objets actifs. Dans ce cas, il y a deux sortes de communication : 1) un objet actif demande une opération à un autre objet actif de façon synchrone (il s'agit d'une communication par rendez-vous et l'objet appelant doit attendre l'objet appelé) ou 2) un objet actif envoie un signal ou demande une opération de manière asynchrone à un autre objet (dans ce cas, il s'agit d'une communication par boîte à lettres ; l'émetteur n'est pas bloqué par l'interaction).
- Un objet actif peut transmettre un message à un objet passif. Dans ce cas, si plusieurs objets actifs envoient des messages à un même objet passif le problème d'exclusion mutuelle se pose. Pour résoudre ce problème, on peut traiter l'objet passif comme une zone critique (en mettant en place une solution d'exclusion mutuelle notamment par une coordination des objets actifs). Dans ce cas, on rajoute, dans la définition des opérations de la classe active, la contrainte `{concurrent}` aux opérations à exécuter en exclusion mutuelle.
- Un objet passif peut transmettre un message à un objet actif. Ce cas a la même sémantique que le premier cas.

2.2. Evénements et signaux

On utilise le concept d'événements pour modéliser les occurrences de stimuli à prendre en compte dans différents diagrammes (états-transitions, séquence...).

Les événements peuvent être synchrones ou asynchrones. En général, les signaux, l'écoulement du temps et des changements d'état sont des événements synchrones ; les appels sont des événements asynchrones.

Signal. Un signal représente un objet envoyé de manière asynchrone par un objet, puis reçu par un autre. Un signal peut être envoyé en tant qu'action d'une transition d'état ou en tant qu'envoi d'un message dans une interaction. Un signal peut ou non transporter des informations. S'il peut transporter des informations, le signal doit être défini avec des paramètres. La communication par signaux est asynchrone : l'expéditeur du signal continue son exécution après l'envoi du signal et n'attend pas de réponse immédiate du récepteur. Les signaux ont de nombreux points communs avec les classes ordinaires : instanciation et généralisation de signaux notamment. La figure 3 montre un exemple de relation de généralisation entre les signaux `Sig1`, `Sig2` et `Sig3`. Comme le montre la figure 3, on peut modéliser les signaux comme des classes stéréotypées et utiliser une dépendance par `send` pour montrer l'opération d'une classe qui envoie des signaux ou des exceptions. Dans l'exemple de la figure 3, `OperationA` envoie l'exception `Except1` et `OperationB` envoie le signal `Sig3`.

Appel d'opération. Un événement d'appel permet de déclencher une opération exécutée par l'objet appelé. Généralement, l'appel se fait de manière synchrone pour permettre à l'appelant d'attendre la fin de l'opération appelée avant de poursuivre son travail ; il s'agit d'un mécanisme de rendez-vous.

Evénement temporel. Un événement temporel est un événement qui exprime l'écoulement du temps de deux manières : `after` permet d'exprimer une durée et `when` permet d'exprimer un temps absolu. La figure 4 montre la représentation graphique des événements d'appel et temporels. Dans l'état `Activite1`, on appelle l'opération `NomOperation`. Dans l'état `Activite2`, on attend l'écoulement de 5 secondes avant d'exécuter l'opération `A`. Dans `Activite3`, on attend midi pour exécuter l'opération `B` et passer dans l'état `Activite4`.

On peut modéliser les événements appel qu'un objet peut recevoir par des opérations sur la classe d'objet en utilisant un compartiment supplémentaire dans la classe considérée.

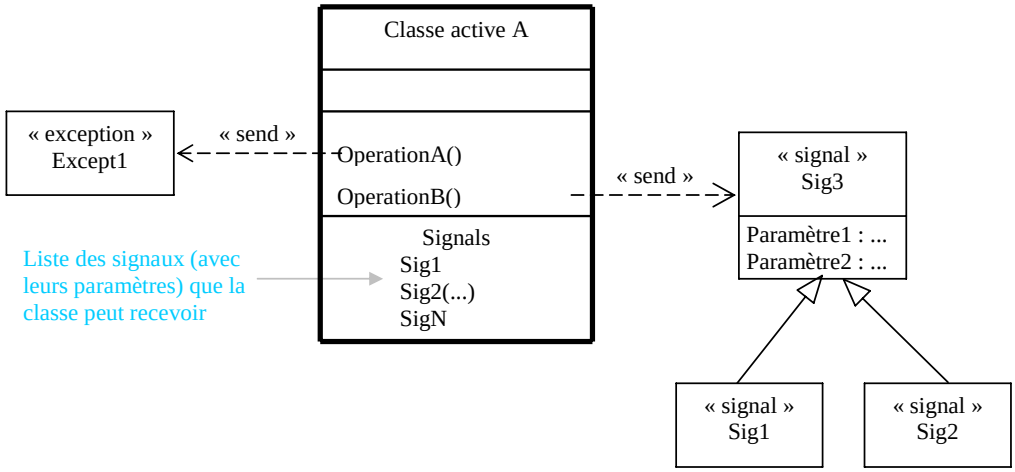

Fig. 3. Modélisation des signaux dans une classe active

2.3. Contraintes de temps

La modélisation du temps est un élément essentiel pour les systèmes temps réel et/ou répartis.

Une *contrainte de temps* s'exprime par une expression du temps relatif (durée) ou absolu (un instant). Il n'y a pas de règles précises pour exprimer les contraintes temporelles. On peut associer une contrainte temporelle à un signal ou un appel d'opération. Les contraintes temporelles sont utilisées essentiellement dans les diagrammes liés au comportement (diagramme de séquence, diagramme de collaboration, diagramme d'activités, diagramme états-transitions).

Pour exprimer une contrainte de temps sur un événement, on associe un nom unique à l'instance de l'événement concerné, ensuite on utilise ce nom comme préfixe pour spécifier la contrainte de temps. Par exemple, si à l'action `OuvrirVanne` on associe le nom `A` et au signal `Temperature` on associe le nom `M` on peut exprimer les contraintes suivantes :

- `{A.debut ≥ 12:0:0}` pour exprimer la date au plus tôt pour démarrer l'opération `A` ;
- `{A.fin ≤ 12:0:1}` pour exprimer la date au plus tard pour terminer l'opération `A` ;
- `{A.duree_d_execution ≤ 10 ms}` pour exprimer la durée maximale d'exécution de l'opération `A` ;
- `{A.periode = 10 ms}` pour exprimer la période l'opération `A` ;
- `{M.tempsTransfert ≤ 100 ms}` pour exprimer le délai maximum de transfert du message `M`.

Ainsi, on peut exprimer toutes les formes de contraintes temporelles habituellement utilisées dans les systèmes temps réel concernant les tâches et les messages. Se reporter à la figure 9a pour voir la forme graphique où apparaissent les contraintes de temps sur des diagrammes de séquence.

Pour modéliser les contraintes de temps, il faut notamment :

- déterminer, pour chaque événement, s'il doit commencer ou se terminer à un moment précis ;
- déterminer pour chaque séquence de messages s'il y a des contraintes de temps sur les échanges ;
- exprimer les aspects temporels liés aux opérations (durée, période...).
-

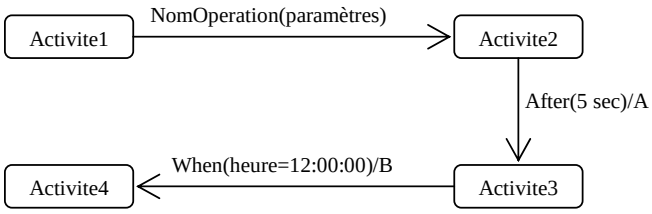

Fig. 4. Envois d'événements d'appel et temporels

2.4. Interactions

Les systèmes informatiques en général et les systèmes temps réel et embarqués en particulier, sont en interaction avec leur environnement. Cette interaction peut être sujette à différentes contraintes (contraintes de temps, d'ergonomie, de sécurité...) selon les spécificités de l'environnement.

Dans le contexte de modélisation avec UML, une **interaction** comprend un ensemble de messages échangés au sein d'un groupe d'objets pour réaliser un certain objectif (travail de coopération).
Les interactions sont utilisées pour modéliser l'aspect dynamique de collaborations entre objets (ou aussi entre des interfaces, des composants ou des cas d'utilisation).

En général, les interactions se sont à l'aide de **messages** qui impliquent l'envoi de valeurs ou de signaux, l'invocation d'actions de base (`call` : opération sur un objet, `return` : renvoi de valeur à l'émetteur, `send` : envoi d'un signal à un objet, `create` : création d'un objet, `destroy` : destruction d'un objet) ou d'actions complexes définies par l'utilisateur.

Par exemple, la figure 5 illustre des interactions entre trois objets : un objet `c` de la classe `Client`, un objet anonyme de la classe `Controleur` et un objet `p` de la classe `Planificateur`. Dans cet exemple, on a les messages suivants :

- `create` : appel d'opération de base de création d'un objet de la classe `Controleur` ;
- `destroy` : appel d'opération de base de destruction d'un objet de la classe `Controleur` ;
- `FournirItineraire(x)` : appel d'opération de l'objet de la classe `Controleur` pour déterminer un itinéraire ; On associe une contrainte de temps à cette action pour indiquer qu'elle doit s'exécuter au bout de 10 ms au maximum.
- `Chemin` : renvoi de résultat d'appel ;
- `Notifier()` : envoi de signal à l'objet `p` pour lui notifier le chemin.

On peut modéliser une interaction en mettant l'accent :

- soit sur l'ordre chronologique des messages ; cette forme de représentation est appelée diagramme de séquence ;
- soit sur le séquencement de messages dans le contexte d'une structure statique. Dans ce cas, on associe des numéros aux messages par rapport au début de l'interaction. Cette forme de représentation est appelée diagramme de collaboration (ou de communication).

Ces deux formes de représentations des interactions sont isomorphes. On peut choisir une seule forme ou les deux selon les besoins.

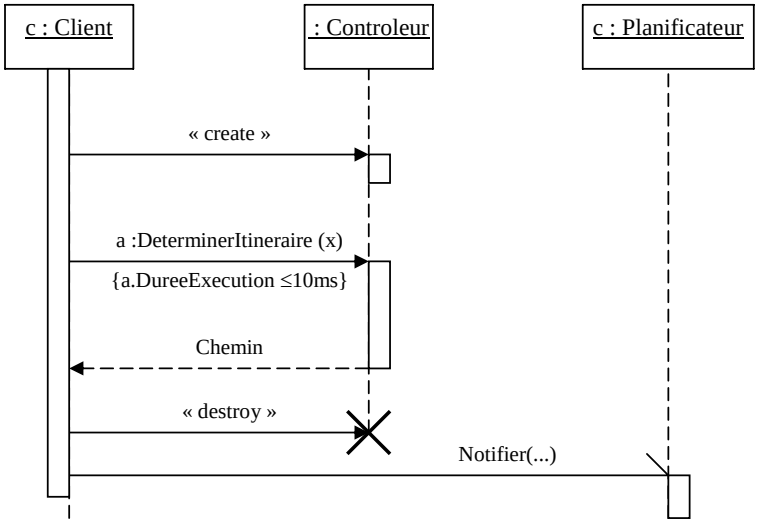

Fig. 5. Exemple d'interactions sous forme de diagramme de séquence.

3. Cas d'utilisation

3.1. Concepts

Comme la plupart des systèmes à modéliser sont en interaction avec leur environnement, une des meilleures façons d'aborder la modélisation du comportement d'un système est d'identifier et clarifier la manière dont ce système interagit avec son environnement. Les **cas d'utilisation** permettent de répondre à ce besoin. Ils précisent les **comportements essentiels** d'un système (ou d'une partie de ce système) en décrivant les **acteurs** qui interviennent et les séquences d'actions permettant l'interaction avec ces acteurs.

Définition (cas d'utilisation)

> Un cas d'utilisation décrit un ensemble de séquence d'actions, y compris des variantes, qu'un système exécute pour produire des résultats pour un acteur.

Chaque cas d'utilisation est identifié par un nom unique et parlant, ce nom peut une chaîne de caractères, un nombre ou même une phrase. Il est recommandé d'utiliser un verbe à l'infinitif suivi d'un complément ce qui permet d'utiliser le point de vue de l'acteur et non celui du système. Par exemple, utilisez « retirer de l'argent » plutôt que « retrait d'argent ».

Définition (acteur)

> Un acteur représente un ensemble cohérent de rôles joués par les entités qui interagissent avec le système. Ces entités peuvent être des utilisateurs humains ou d'autres sous-systèmes.

Les acteurs sont connectés aux cas d'utilisation uniquement pas association. Les acteurs et cas d'utilisation communiquent entre eux.

Un cas d'utilisation sert à préciser les comportements du système sans dire comment ces comportements seront réalisés. Ces comportements sont en fait des fonctions du système. Les cas d'utilisation représentent les exigences fonctionnelles sur le système.

Dans les systèmes complexes, les cas d'utilisation se présentent sous forme de variantes, des cas d'utilisation sont des spécialisations d'autres cas pour prendre en compte des besoins de plus en plus spécifiques.

Les cas d'utilisation s'appliquent à un système entier ou à certaines de ses composantes à différents niveaux d'abstraction.

3.2. Représentation graphique

Généralement, un acteur est représenté par un bonhomme. Il est recommandé d'utiliser des icônes différentes du bonhomme, quand les acteurs ne sont pas des êtres humains. A l'aide des mécanismes d'extensibilité d'UML, on peut stéréotyper un acteur et créer des icônes différentes qui, en fonction des besoins, offre un meilleur repère visuel.

Les figures 6a et 6b montrent des exemples de cas d'utilisation.

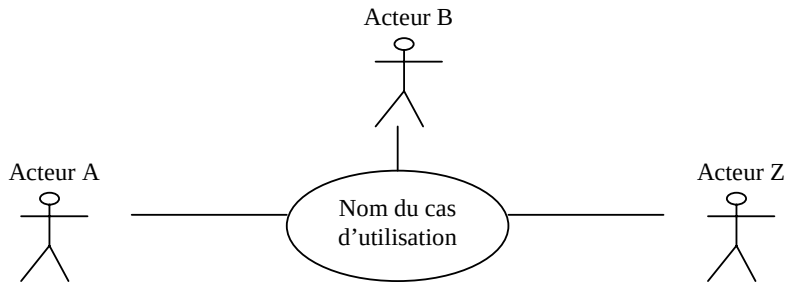

Fig. 6.a Représentation d'un cas d'utilisation

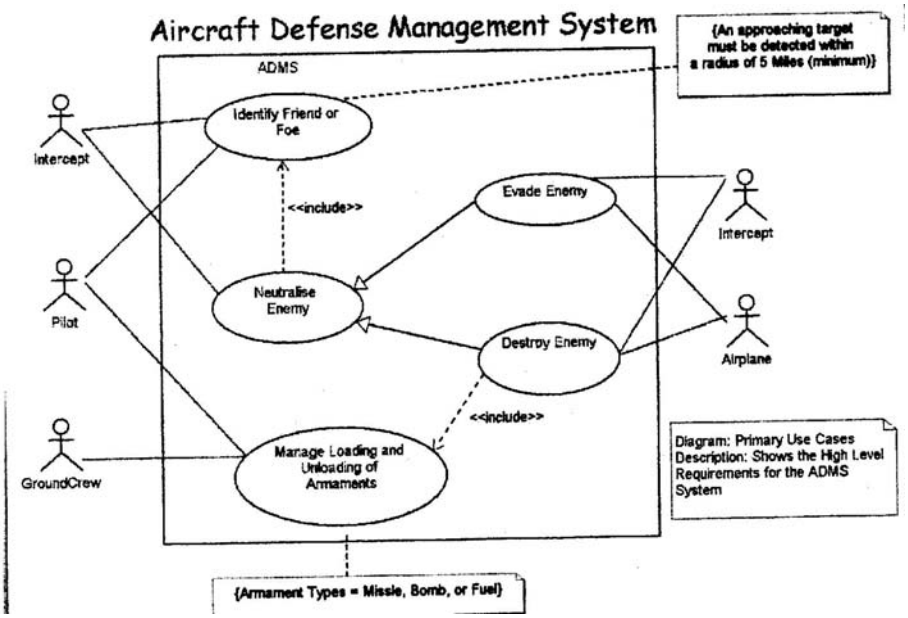

Fig. 6.b. Exemple de cas d'utilisation pour un avion militaire

3.3. Représentation textuelle

On peut préciser le comportement d'un cas d'utilisation en décrivant les flots d'événements à l'aide de texte suffisamment clair pour être compris facilement par une personne extérieure. Il faut décrire quand le cas d'utilisation commence et quand est-ce qu'il se termine, ainsi que les moments où il interagit avec les acteurs. Il existe aussi d'autres approches plus au moins formelles pour décrire les cas d'utilisation (par exemple, en associant un *statechart* au cas d'utilisation, Figure 7b).

On peut associer à chaque cas d'utilisation une fiche de description textuelle éventuellement structurée de la manière suivante (figure 7a) :

- *Sommaire d'identification* : préciser le titre, le but, les dates de création et révision, le ou les responsables, les acteurs...
- *Description des enchaînements* : décrire le scénario nominal, les enchaînements alternatifs, les enchaînements d'exception, les préconditions et les postconditions.
- *Besoins d'IHM* : ajouter éventuellement les contraintes sur les interfaces homme-machine.
- *Exigences non fonctionnelles* : rajouter éventuellement les informations sur : la fréquence des opérations, les temps de réponse, la fiabilité, la sécurité...

Name: Alarm on Critical Event

Purpose
The purpose is to identify when the patient is at imminent risk and identify this to the attending anesthesiologist so that appropriate action can be taken.

Description
ACE-1: When an alarming condition occurs, it shall be annunciated—that is, a meaningful alarm message (including the time of occurrence, the type of alarm, the source of the alarm, and the likely cause of the alarm) shall be displayed and an alarming tone shall be sounded.
ACE-2: When multiple alarms are being annunciated, they shall be displayed in order of severity first, then in order of occurrence, newest first.
ACE-3: If an annunicated alarm isn't displayed (because higher criticality alarms are being displayed and there is insufficient space to display the alarm in question), then it cannot be acknowledged without first being displayed on the screen.
ACE-4: Alarms must be explicitly acknowledged by the user pressing the Alarm Ack button after they have occurred even if the originating alarming condition has been corrected.
ACE-5: If the originating condition of an alarm has been corrected but the alarm has not yet been acknowledged, then the display of the alarm message shall be greyed out. All other alarm messages shall be displayed in the normal color.
ACE-6: The Alarm Ack button shall cause the audible alarm sound to be silenced but does not affect the visual display of the alarm message. The silence shall hold for 2 minutes. If the alarm condition ceases after the acknowledgement but before the silence period times out, then the alarm shall be dismissed. If, after the silence period has elapsed, the originating condition is still valid or if it has reasserted itself during the silence period, then the alarm shall be reannunicated.

Preconditions
1. System is properly configured and has been initialized.
2. Alarming parameters have been set.
3. Alarming is enabled.

Postconditions
1. Alarming conditions are displayed and audibly announced.

Other Constraints
1. Alarms shall be filtered so that each patient condition results in a single annunciated alarm.
2. Alarms shall be detected within 9 seconds of their occurrence.
3. Alarms shall be annunciated within 1 second of the detection of the alarming condition.
...

Fig. 7a. Exemple de cas d'utilisation (description textuelle)

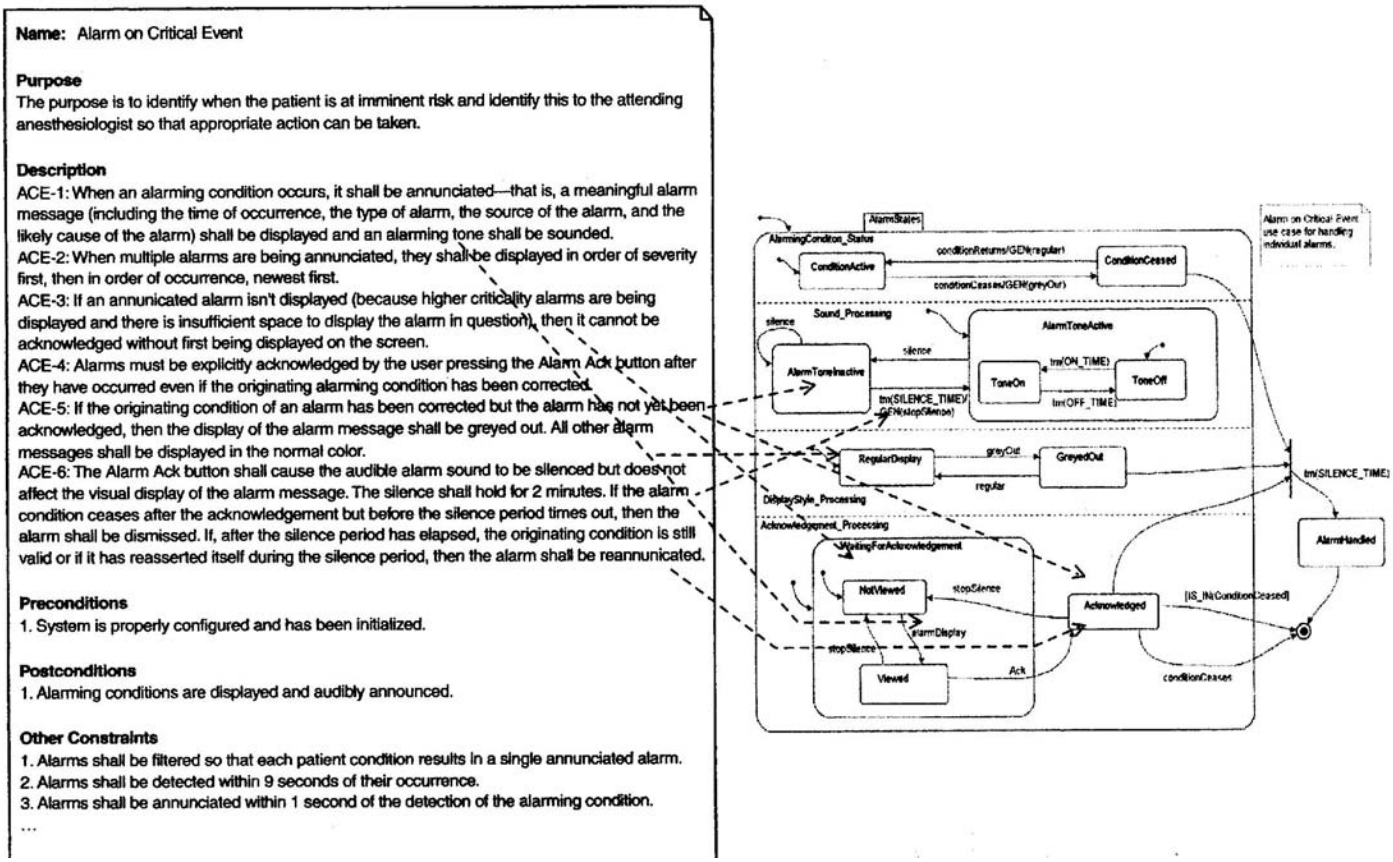

Fig. 7b. Exemple de cas d'utilisation (description textuelle et *statechart*)

3.4. Organisation de cas d'utilisation

Lorsque les cas d'utilisation sont nombreux, on peut les organiser en paquetages, comme on organise des classes.

Pour pouvoir réutiliser de manière efficace les cas d'utilisation, il est souvent intéressant de les traiter comme des classes, en identifiant les cas d'utilisation basiques ensuite de les enrichir selon les besoins. On peut organiser les cas d'utilisation en précisant leurs relations mutuelles de généralisation, d'inclusion et d'extension. Ces relations ont les mêmes objectifs que pour les classes (héritage...).

On introduit une relation d'inclusion par une dépendance à laquelle on associe le stéréotype `include`. Cela permet de décrire simplement un cas d'utilisation en mentionnant les cas d'utilisation qu'il inclut sans avoir à les redéfinir.

On introduit aussi une relation d'extension par une dépendance à laquelle on associe le stéréotype `extend`. On utilise une relation d'extension pour différentes raisons : modéliser une partie facultative dans le comportement d'un système, modéliser un flot secondaire qui n'est exécuté que dans certaines situations cas, modéliser plusieurs flots qui peuvent être insérés à certains points selon l'interaction avec un acteur.

La figure 8a donne la forme générale de diagramme de cas d'utilisation. La figure 8b donne un exemple de diagramme de cas d'utilisation pour un système de gestion de commandes de clients. On y trouve deux types d'acteurs, `Client 1` et `Client 2`, qui peuvent passer des commandes normales ou urgentes. L'identité du client est validée soit par un mot de passe soit par une emprunte digitale. Dans cet exemple de système, on voit que le cas d'utilisation `Passer commande urgente` est une extension de `Passer commande` et que le cas d'utilisation `Valider utilisateur` est une généralisation des deux modes de vérification `Vérifier mot de passe` et `Scanner emprunte digitale`.

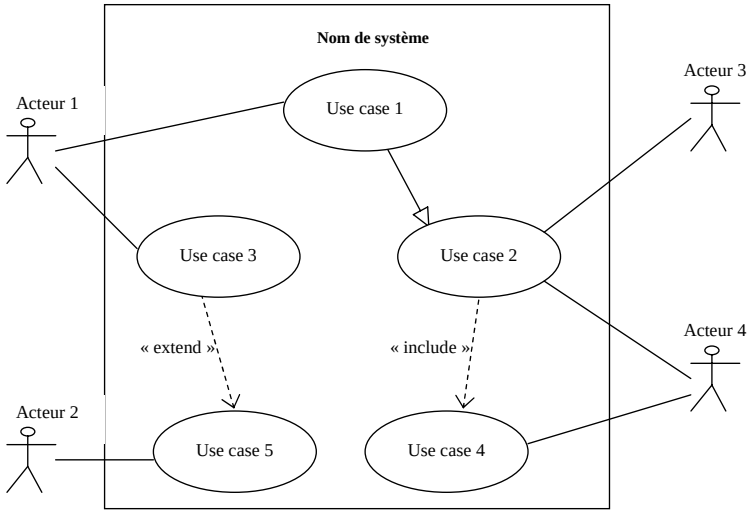

Fig. 8a. Diagramme de cas d'utilisation

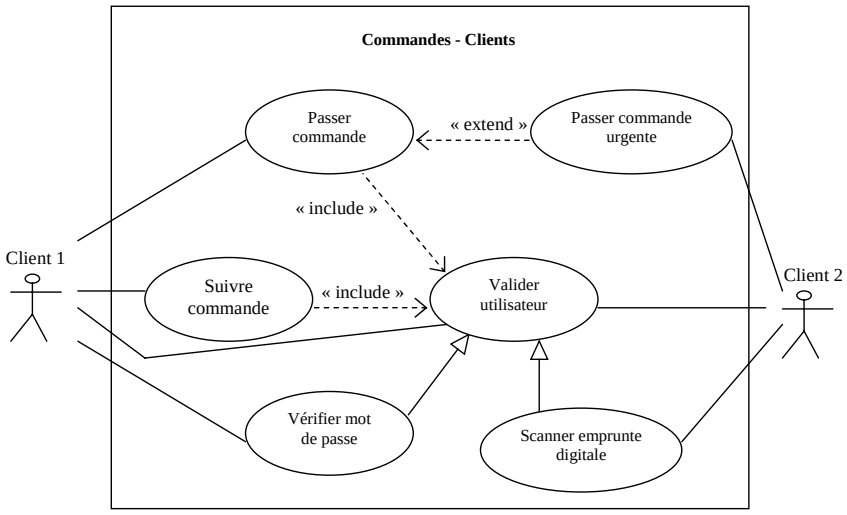

Fig. 8b. Exemple 2 de diagramme de cas d'utilisation

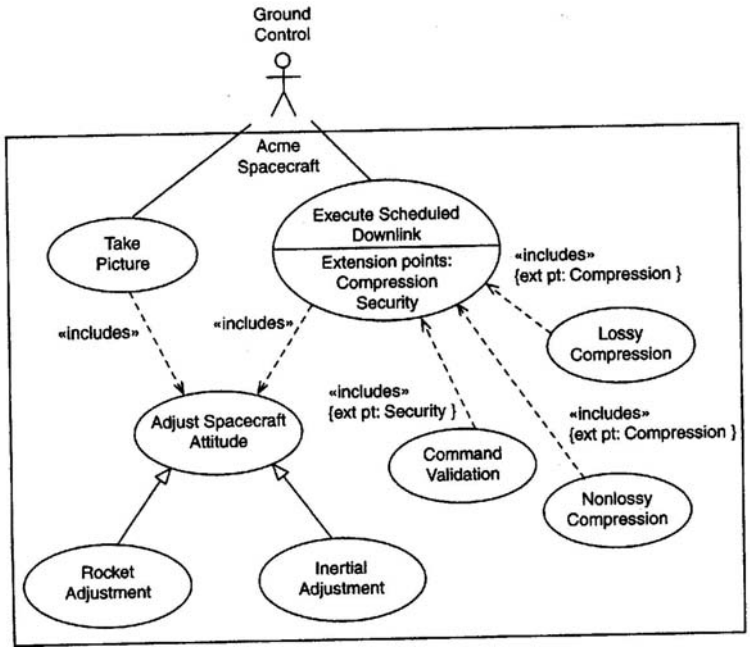

Fig. 8c. Exemple 2 de diagramme de cas d'utilisation

3.4. Conseils pour élaborer des cas d'utilisation

Il faut toujours se rappeler que les cas d'utilisation sont élaborés pour trois raisons :

- Ils permettent aux experts du domaine (non informaticiens en général) de donner leur point de vue du système à développer. C'est à ce niveau qu'il y a échange de points de vue entre informaticiens et experts du domaine cible du logiciel (domaine de la chimie, médecine, commerce...).
- Ils permettent aux développeurs d'aborder les différents éléments du système et de les comprendre.
- Ils servent de base, pendant le développement, pour tester chaque élément du système au fur et à mesure que son développement évolue. Si le comportement d'un élément, à un stade donné du développement, ne correspond pas aux cas d'utilisation, il faut revoir cet élément.

Pour modéliser le comportement d'un élément du système, il faut :

- identifier les acteurs avec lesquels il dialogue et les organiser selon leurs rôles ;
- pour chaque acteur, considérer les interactions (identifier les événements et messages) ;
- pour chaque acteur, identifier les situations d'interaction exceptionnelles ;
- organiser les comportements identifiés comme des cas d'utilisation ;
- décrire chaque cas d'utilisation en écrivant des scénarios de fonctionnement. Un scénario est l'exécution complète d'un cas d'utilisation. Par exemple, dans le cas d'un système d'automatisation d'un cabinet médical, le scénario `Prendre rendez-vous` peut se décrire ainsi : « le patient appelle pour prendre un rendez-vous, la secrétaire cherche le créneau horaire disponible qui convient au patient, le patient confirme et le rendez-vous est noté » ;
- factoriser les comportements des cas d'utilisation en précisant ce qu'ils ont de commun ensuite appliquer des relations d'inclusion, d'extension... pour mieux organiser les cas d'utilisation.

Il faut éviter de représenter des acteurs qui sont en réalité des éléments internes du système à réaliser. Il faut essayer toujours de privilégier les acteurs physiques qui vont utiliser le système.

4. Diagrammes de séquence

Un diagramme de séquence met en évidence la chronologie des messages échangés entre objets. La structure d'un diagramme de séquence est la suivante :

- on place d'abord les objets participant à l'interaction. On place en général à gauche l'objet qui commence l'interaction.
- On place, le long des axes verticaux associés aux différents objets, les messages envoyés et reçus par les objets en respectant la chronologie des échanges. Cela permet d'avoir une indication visuelle du flot de contrôle dans le temps.

Convention de représentation graphique :

- La ligne de vie d'un objet représente l'existence d'un objet dans le temps.
- La croix indique la fin de vie de l'objet.
- Une bande rectangulaire indique une période d'activité de l'objet pendant laquelle l'objet accomplit une action (le haut du rectangle coïncide avec le début de l'action et le bas avec sa fin. On peut symboliser un emboîtement d'une période d'activité en collant, à droite du premier rectangle, deux ou plusieurs rectangles. Cela permet de matérialiser la récursivité les appels internes.
- Dans le cas de UML 2.0, on peut avoir des parties du diagramme de séquence qui se répètent (utilisation de `loop`) de manière infinie ou pendant un certain nombre de fois.

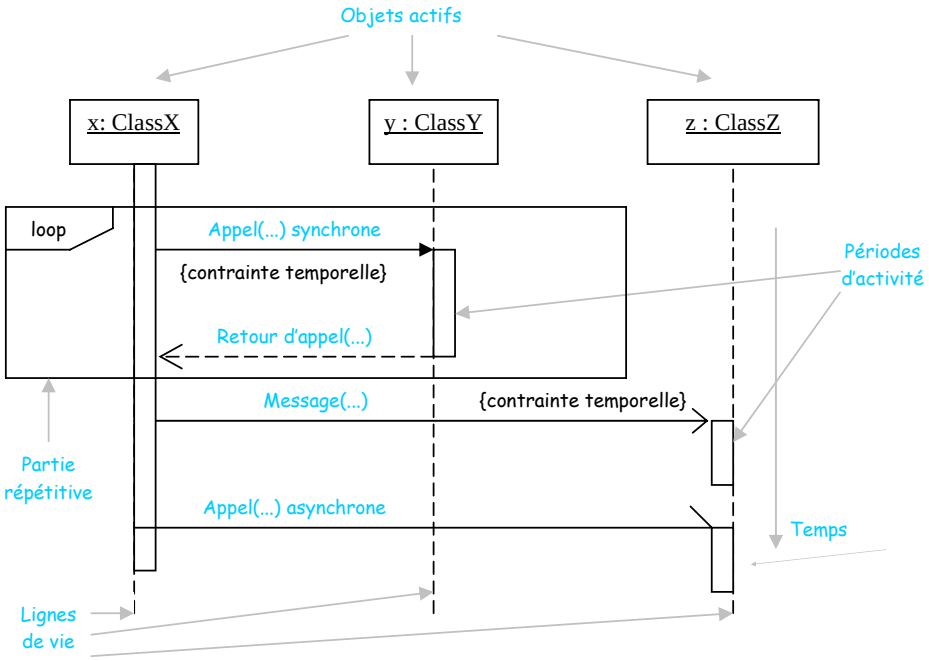

Fig. 9a. Représentation de diagrammes de séquence

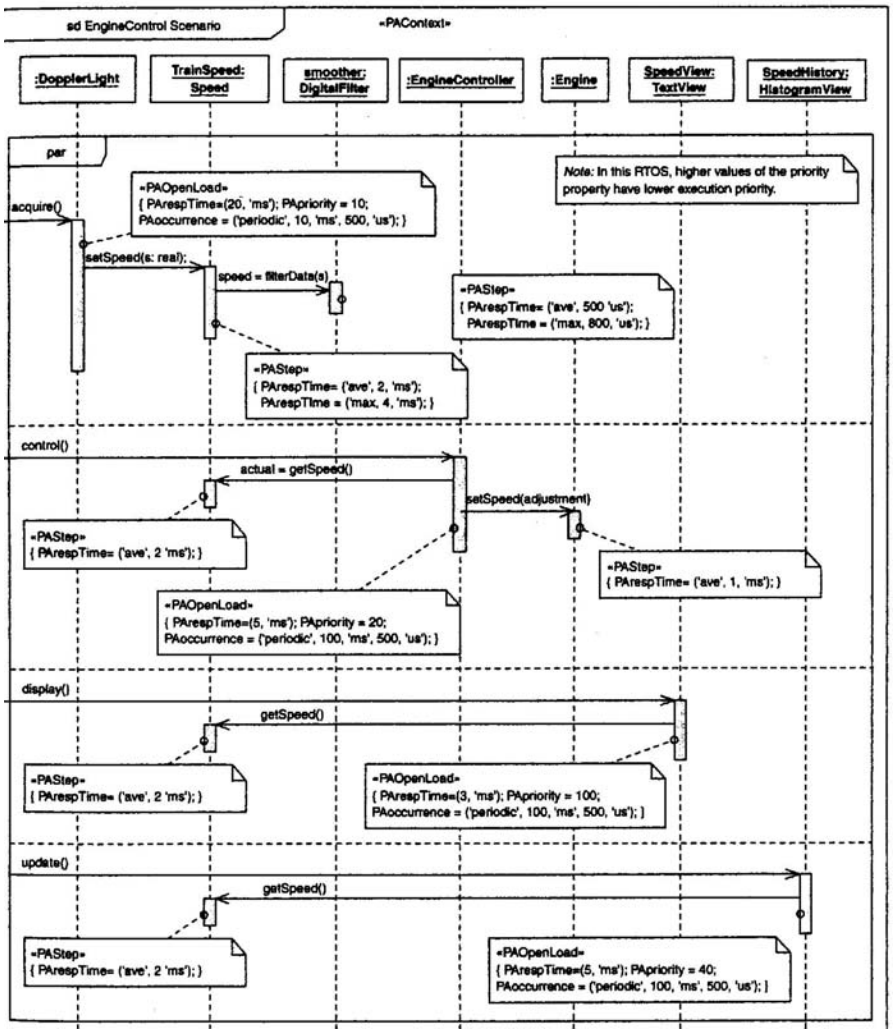

Fig. 9b. Exemple de diagrammes de séquence (avec des contraintes de temps)

5. Diagrammes d'activités

5.1. Concept et utilisation

Une **action** est l'unité de base pour la description de comportement. Elle est atomique et elle peut prendre l'une des formes suivantes :

- lecture ou écriture de variable,
- calcul,
- appel d'opération (`call`),
- retour d'appel d'opération (`return`),
- création d'instance d'objet (`create`),
- destruction d'instance d'objet (`destroy`),
- envoi de signal (`send`),
- génération d'exception.

Dans UML 2.0, les notations graphiques pour représenter les actions telles que l'envoi et réception de signaux sont celles utilisées par SDL (cela permet plus de rapprochement entre les deux langages).
Une **activité** est une exécution non atomique. Elle produit des actions atomiques conduisant au changement d'état du système. Un **diagramme d'activités** est principalement un organigramme qui montre le flot de contrôle d'une activité à l'autre. Il modélise les étapes séquentielles et concurrentes dans un processus de calcul. Les diagrammes d'activités sont utilisés pour les mêmes besoins que les diagrammes de Gantt et les diagrammes de PERT très connus, en particulier dans le monde industriel.

Un diagramme d'activités est un automate à états finis. Il contient donc des états (en particulier un état initial et un état final et éventuellement un état final d'erreur –valide dans la version UML 2.0) et des transitions. On distingue les **états d'action** et les **états d'activité**. Les états d'action représentent chacun l'exécution d'une action ; ce sont des calculs atomiques (donc les états d'action ne sont décomposables) et de durée d'exécution nulle. Les états d'activité peuvent être décomposés et ne sont pas atomiques et leur durée d'exécution peut être longue. Au niveau graphique, on utilise les mêmes notations pour les états d'action et que pour les états d'activité.

Les diagrammes d'activités sont utilisés essentiellement pour deux objectifs :

- pour modéliser un workflow : on s'intéresse ici aux activités telles que les voient les acteurs qui collaborent avec le système. Pour cela, il faut : sélectionner les objets métier qui ont des responsabilités, identifier les préconditions, les postconditions et l'état initial du workflow, préciser les activités et actions qui ont lieu dans le temps, représenter les transitions en tenant compte des situations de concurrence entre les flots, diviser le diagramme en travées (on dit aussi partitions) si nécessaire pour en accroître la lisibilité.
- Pour modéliser une opération : on s'intéresse ici aux détails des calculs associés à une opération simple ou complexe. Pour cela, il faut : clarifier les paramètres de l'opération, identifier les préconditions, les postconditions et l'état initial, identifier les situations de branchement, éventuellement utiliser des flots concurrents.

5.2. Structure d'un diagramme d'activités

Transition : une transition traduit le passage d'un état à autre. Une transition sans déclencheur est une transition franchie dès que le travail de l'état source est terminé ; elle n'a pas d'événement. Par exemple, toutes les transitions de la figure 11 n'ont pas de déclencheur.

Branchement conditionnel. Il permet d'exprimer le choix entre plusieurs transitions en fonction d'une expression booléenne (cette expression est une garde au sens des automates). Le mot clé `else` est utilisé pour indiquer qu'il faut prendre la transition étiquetée par `else` si aucune autre condition n'est vraie. La représentation graphique du branchement conditionnel est donnée par la figure 10.

Synchronisation de flots concurrents. Plusieurs flots peuvent fonctionner en parallèle dans un système. UML utilise une barre de synchronisation pour spécifier la division (un 'fork') et le regroupement (un 'join') de flots de contrôle parallèles. Par exemple, on peut lancer la production simultanée de plusieurs pièces d'un lot sur les machines de production ensuite toutes les pièces doivent être rassemblées et placer dans une boite avant de passer au lot suivant. Dans ce cas, on doit modéliser des activités concurrentes. Une barre de synchronisation lie au moins deux flots de contrôle. Si les flots se trouvent en dessous de la barre (cas d'un 'fork'), les flots concernés commencent en même temps et évoluent en parallèle. S'ils se trouvent sous la barre (cas d'un 'join'), chacun d'eux attend que tous les flots entrants aient atteint la barre pour continuer. De plus, les activités des flots de contrôle peuvent communiquer entre elles par envois de signaux. Il faut noter que les fourches (barres de 'fork') et jonctions (barres de 'join') doivent s'équilibrer (c.-à-d. le nombre de flots qui sortent d'une fourche doit être égal à celui des flots qui entrent dans la jonction correspondante). La figure 11, donne un exemple simplifié du fonctionnement d'un système audio vidéo. Dans cet exemple, les activités `Synchro bouche` et `Flux audio` sont concurrentes.

Travée ('*swimlane*'). Les travées (appelées aussi partitions) permettent de diviser un diagramme d'activités en deux ou plusieurs rangées (dites travées) pour en faciliter la lecture. Dans un diagramme d'activités, chaque travée est identifiée par un nom et séparée de ces voisines par deux tirets verticaux. Par exemple, sur la figure 11, nous avons divisé le diagramme de transitions en deux travées : `Vidéo` et `Audio`.

Flot d'objet. Un objet peut être impliqué dans un diagramme d'activités. Dans ce cas, il est possible de le préciser dans le diagramme d'activités en le plaçant dans le diagramme et en le connectant à la transition qui le crée, le détruit ou le modifie. Par exemple, dans la figure 11, l'objet `hp` est impliqué dans le diagramme d'activités.

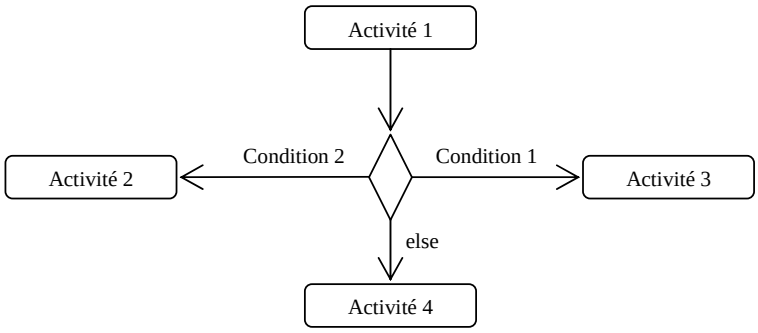

Fig. 10. Représentation graphique du branchement conditionnel

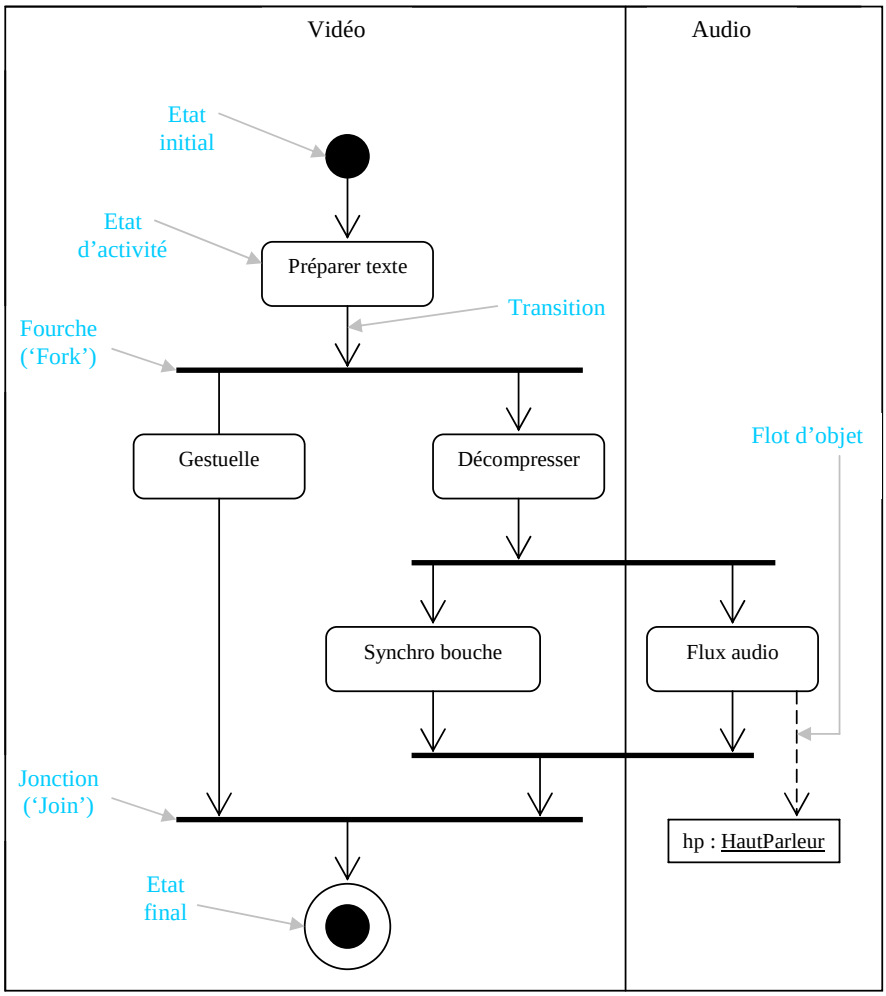

Fig. 11. Synchronisation de flux concurrents avec deux travées

6. Diagrammes d'états-transitions (Statecharts)

6.1. Automates à états finis

Les interactions permettent de modéliser le comportement d'un groupe d'objets en faisant apparaître les points où des échanges d'informations ont lieu entre ces objets. Les diagrammes d'états-transitions sont utilisés pour décrire le comportement des objets individuels. Ainsi, on associe un diagramme d'états-transitions à chaque objet dont on veut décrire les différents états par lesquels passe l'objet durant sa vie, en particulier la manière dont il réagit aux événements. Lorsqu'un événement se produit, en fonction de l'état actuel de l'objet, une activité a lieu.

Un automate à états finis peut être visualisé de deux manières différentes :

- soit en mettant en évidence le flot de contrôle entre activités (dans ce cas, on utilise un diagramme d'activités)
- soit en mettant en évidence les états potentiels de l'objet et les transitions entre ces états ordonnancées par les événements (dans ce cas, on utilise un diagramme états-transitions).

Les automates sont bien adaptés à la modélisation du comportement d'objet qui dépend de l'occurrence d'événements asynchrones et qui peut éventuellement dépendre du passé de l'objet. Les automates à états finis servent aussi à modéliser des systèmes entiers.

La sémantique associée aux machines d'états repose sur une machine d'exécution virtuelle composée de :

- Une file d'attente d'évènements qui sert à stocker les occurrences d'évènements entrantes en attendant de les consommer.
- Une politique de choix des évènements qui détermine l'ordre d'extraction des occurrences d'évènement contenues dans la file d'attente.
- Un processeur à évènements qui exécute les traitements associés aux évènements en respectant la sémantique des machines d'états-transitions de UML et, en particulier, l'hypothèse d'exécution «Run-To-Completion».

Les occurrences d'évènements sont dépilées une par une et consommées par une machine d'états-transitions. L'ordre dans lequel elles sont dépilées n'est pas précisé dans UML, cela constitue un point de variation sémantique. La sémantique d'exécution des évènements est basée sur l'hypothèse dite de « Run-To-Completion ». Cela signifie qu'un évènement ne peut être dépilé puis consommé que lorsque le traitement de l'évènement précédent est achevé.

6.2. Représentation

Il faut signaler que les règles et concepts de description des automates utilisés dans UML puisent leur source dans les *Statecharts* de Harel qui sont des automates hiérarchisés. La figure 13a illustre les principaux éléments qui composent un diagramme états-transitions.

Etat. Un état est une situation au cours de la vie d'un objet qui satisfait certaines conditions. Un objet reste dans un état pendant une durée déterminée (à moins que l'objet ne se soit bloqué indéfiniment). Deux états sont particuliers : l'état initial qui marque la création de l'objet et le début de son automate et l'état de fin qui marque la disparition de l'objet. Un état se compose de 7 parties dont certaines sont optionnelles (Fig. 12) :

- un *nom* qui identifie clairement l'état (dans de rares cas, on n'utilise pas de nom);
- *action d'entrée*: action exécutée quand l'objet entre dans l'état ; elle est spécifiée via le mot clé `entry`. Lorsqu'une action d'entrée est spécifiée pour un état cela signifie que l'on souhaite exécuter la même action à chaque que l'on rentre de l'état indépendamment de l'état par lequel on est rentré. Notez qu'une action d'entrée ne peut pas avoir d'arguments ni de garde, sauf l'action d'entrée située au début de l'automate pour une classe peut avoir des paramètres qui représentent les arguments que reçoit l'automate lorsqu'il est créé ;
- *action de sortie* : action exécutée quand l'objet sort de l'état ; elle est spécifiée via le mot clé `exit`. Lorsqu'une action de sortie est spécifiée pour un état cela signifie que l'on souhaite exécuter la même action à chaque que l'on sort de l'état indépendamment de l'état vers lequel on part. Notez qu'une action de sortie ne peut pas avoir d'arguments ni de garde. Notez aussi que la possibilité de spécifier des actions d'entrée et sortie peut être la source de certaines erreurs de modélisation si elle n'est pas utilisée correctement ;
- *activités internes* : ce sont des activités effectuées une fois l'action d'entrée est exécutée (évidemment si l'action d'entrée est spécifiée pour l'état). Quand les activités internes sont terminées, on sort de l'état. On utilise le mot clé `do` pour spécifier les activités internes ;
- *transitions internes* : ce sont des transitions internes à l'état qui ont le même sens que les transitions normales, sauf qu'elles n'engendrent pas de changement d'état. Si un événement arrive alors que l'objet est dans un état ayant des transitions internes, l'action de la transition interne (ayant pour déclencheur l'événement arrivé) est exécutée et il n'y a ni changement d'état ni exécution d'action d'entrée ou de sortie. Il faut noter que l'utilisation de ce mécanisme est parfois source d'erreur (donc à utiliser avec modération) ;
- *événements **différés*** : lorsqu'un objet ne peut pas traiter certains événements dans un état, mais peut les traiter dans d'autres, il spécifie la liste des événements qu'il souhaite conserver pour un traitement ultérieur. Avec ce mécanisme, on évite de perdre des événements pour lesquels un traitement doit être effectué. Les événements à différer sont spécifiés à l'aide de l'action `defer`.
- *sous-états* : un état complexe peut être hiérarchisé en plusieurs sous-états pour faciliter la lisibilité et manipulation de l'état global. Un sous-état est un état emboîté dans un autre état (dit état composé). Un état composé a une sous-structure qui permet de masquer à un certain niveau le fonctionnement interne de l'état lorsque celui-ci est complexe. Le mécanisme d'emboîtement permet aussi diminuer le nombre de transitions. Voir plus loin les détails sur la manière de spécifier des états composés.

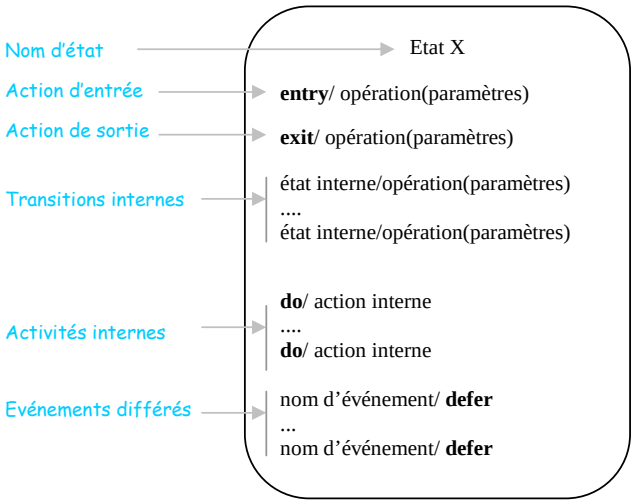

Fig. 12. Représentation graphique d'un état

Transition. Une transition est une relation entre deux états, un état source et un état cible, qui indique sous quelles conditions l'objet passe de l'état source à l'état cible. Une transition comporte cinq parties (qui ne sont toutes obligatoires) :

- **état source** de franchissement de la transition (obligatoire);
- **événement déclencheur** (optionnel) : la réception de cet événement déclenche la transition (si la garde le permet). L'événement déclencheur peut être : un signal, un appel, une durée (`after`), un instant particulier (`when`) ou un changement d'état. Si l'événement est un signal ou un appel, les informations transportées par les paramètres sont accessibles à la transition (pour l'action et la garde). Il est possible d'avoir une transition sans événement déclencheur qui est franchie de manière implicite quand l'état source a terminé son activité ;
- **garde** (optionnelle) : expression booléenne évaluée *après* la réception de l'événement. Si elle st vraie, la transition est effectuée, sinon l'objet reste dans le même état et l'événement est perdu. L'expression booléenne est placée entre crochets. A partir du même état source, on peut avoir plusieurs transitions à condition que les gardes ne se chevauchent pas ;
- **action** (optionnelle) : calcul atomique exécuté par l'objet avant de passe à l'état cible ;
- **état cible** de la transition (obligatoire).

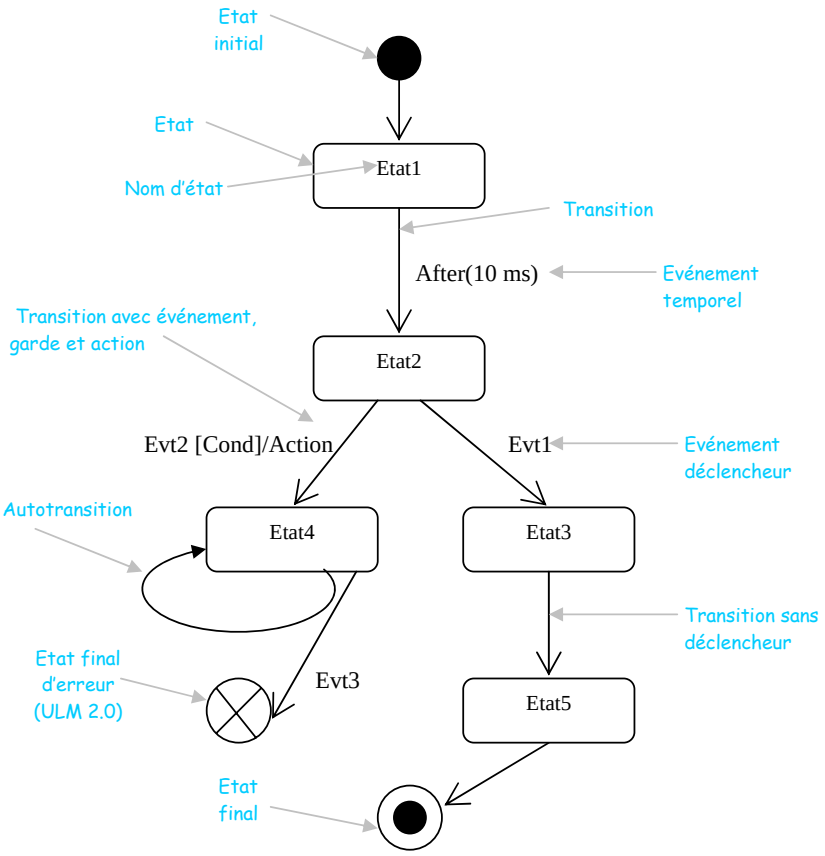

Fig. 13a. Représentation graphique d'automate à états finis

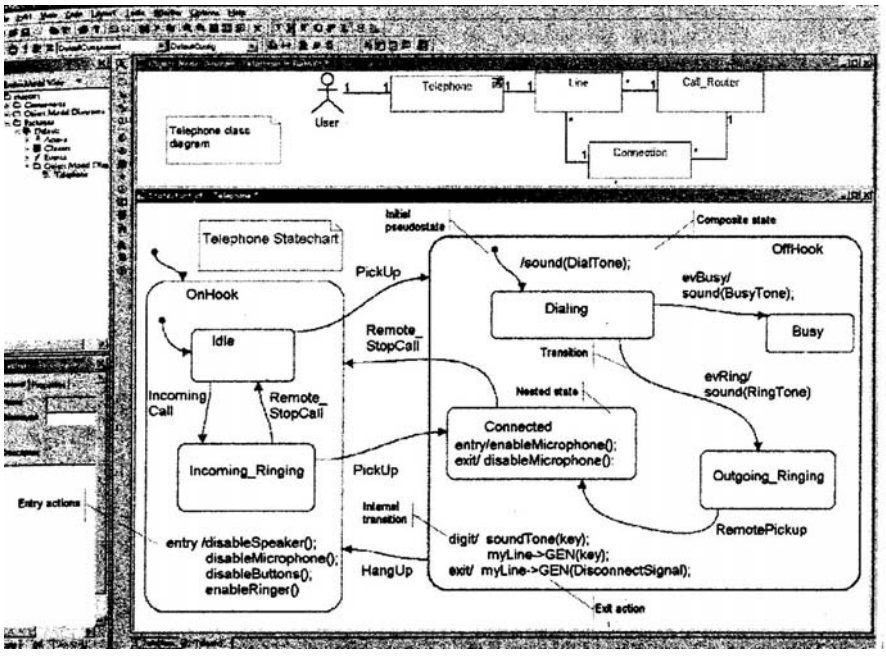

Fig. 13b. Exemple de *statechart* pour modéliser un téléphone

Etats composés. Il y a différents mécanismes que l'on peut utiliser pour emboîter les sous-états afin de former des états composés :

- **Sous-états séquentiels** (Fig. 14) : un état composé est constitué de sous-états qui se suivent. A partir d'une source externe, une transition peut cibler l'état composé ou un sous-état. Si la cible est l'état composé, l'automate emboîté doit commencer à l'état initial après exécution de l'action d'entrée associée à l'état composé. Si la cible est un sous-état, l'automate commence au sous-état spécifié après exécution de l'action d'entrée associée à l'état composé et exécution de l'action d'entrée du sous-état. Une transition qui sort d'un état composé peut avoir pour source l'état composé ou un sous-état. Dans les deux cas, l'action de sortie de l'état composé est exécutée avant de quitter l'état. Dans le cas où on sort d'un sous-état, l'action de sortie de ce sous-état est exécutée avant celle de l'état composé. Une transition dont la source est l'état composé interrompt l'activité de l'automate emboîté. Un automate emboîté peut avoir au maximum un état initial et un état final.
- **Etats à historique** (Fig. 14) : Sauf indication contraire, lorsqu'on rentre dans un état composé, l'action de l'automate repart de son état initial (à moins, bien sûr, que la transition ne cible un sous-état). Dans certaines situations, on a besoin que l'automate emboîté se souvienne du dernier sous-état où il était avant de le quitter pour la dernière fois (par exemple, se rappeler du dernier calcul avant d'aborder un mode de fonctionnement d'urgence). On peut modéliser ce besoin à l'aide d'état à historique. Un automate emboîté avec historique (marqué par un cercle contenant la lettre `H`) repart à partir du dernier sous-état où il était si la transition entrante dans l'état composé à pour cible le symbole `H`. Attention, la première fois que l'on rentre dans l'état composé, on commence à partir du sous-état initial car l'automate emboîté n'avait pas d'historique. Dans le cas où un état composé contient d'autres sous-états qui eux-mêmes sont composés, on peut, en utilisant le symbole `H*` entouré d'un cercle, aller directement vers le sous-état le plus interne où on était avant de quitter l'automate emboîté. Dans tous les cas, quand un automate emboîté atteint son sous-état final, il perd l'historique et se comporte comme si rien ne s'était passé.

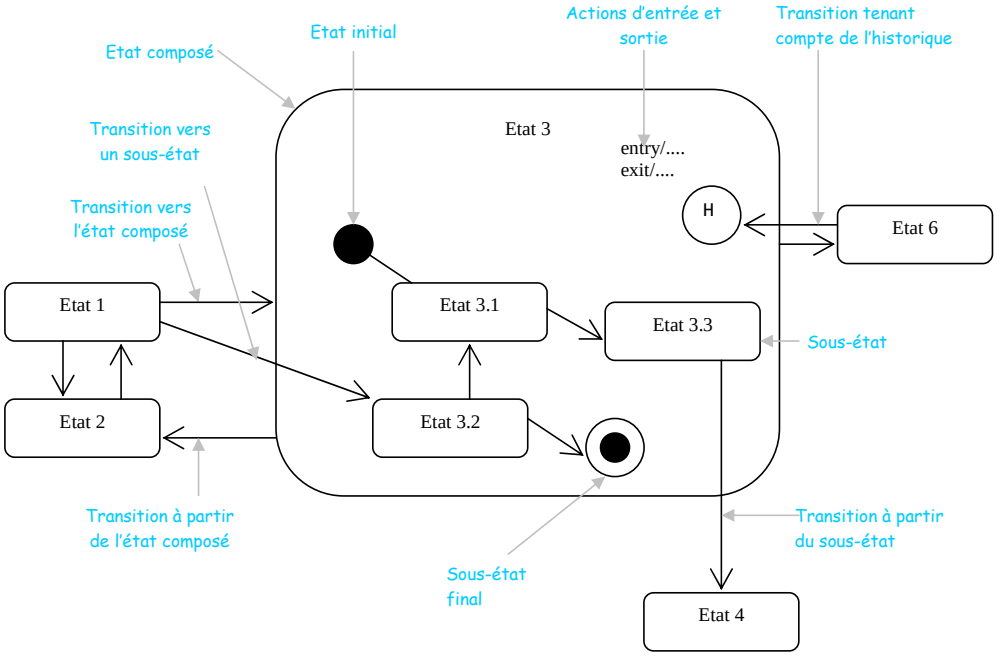

Fig. 14. Représentation d'état ayant des sous-états séquentiels

- **Sous-états concurrents** (Fig. 15) : dans certaines situations, on a besoin de spécifier deux ou plusieurs sous-états concurrents qui fonctionnent de manière parallèle à l'intérieur d'un même automate emboîté. Dans ce cas on subdivise l'état composé en sous-états concurrents. Un sous-état concurrent est composé de sous-états séquentiels. Un sous-état concurrent est un thread. Lorsqu'on rentre dans un état avec des sous-états concurrents, les exécutions de tous ces sous-états sont déclenchées et se poursuivent en parallèle. Tous les sous-états doivent atteindre leur état final pour sortir de l'état composé. Attention, un état ayant des sous-états concurrents n'a pas d'état initial (car chacun de ses sous-états concurrents a un état initial), ni d'état final (car chacun de ses sous-états concurrents a un état final), ni d'historique.

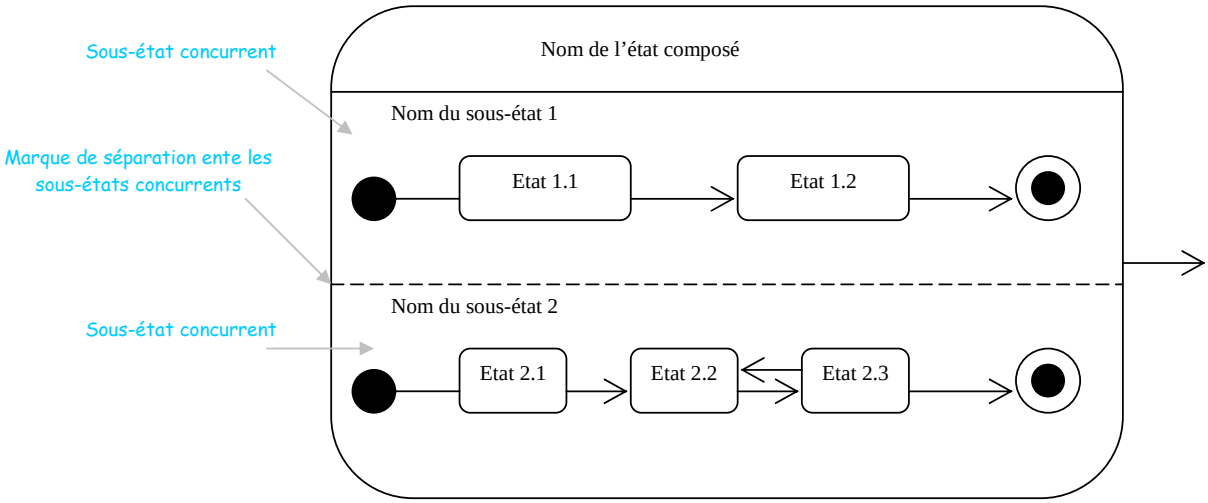

Fig. 15. Représentation d'état ayant des sous-états concurrents

7. Diagrammes de temps

Les diagrammes de temps ont été introduits par la version UML 2.0 pour être utilisés afin de mettre en évidence les aspects temporels lors d'interactions. Ils sont focalisés sur les états et conditions qui changent en fonction de l'écoulement du temps. Ils peuvent être utilisés pour décrire le comportement d'un objet ou d'une classe d'objets.

Comme le montre les figures 16 et 17, on peut représenter un diagramme de temps sous deux formes. Dans les deux figures (qui montrent deux diagrammes de temps équivalents), il y a une contrainte sur la durée des états `SaisieCode` (qui doit être comprise entre 1 et 10 secondes) et `AffichageInfo` (qui doit être comprise entre 1 et 100 secondes).

En général, les informations temporelles que l'on peut représenter sur un diagramme de temps sont les suivantes :

- Période (temps entre deux arrivées dans un même état)
- Echéance (temps pour quitter un état et entrer dans un autre état)
- Temps d'initialisation (temps pour exécuter les actions d'entrée d'un état)
- Temps d'exécution (temps pour exécuter toutes les actions d'un état)
- Temps restant (c'est le temps restant avant d'échéance après avoir exécuté les actions d'un état)
- Gigue (variation au niveau de l'instant de démarrage pour les transitions ou événements périodiques)

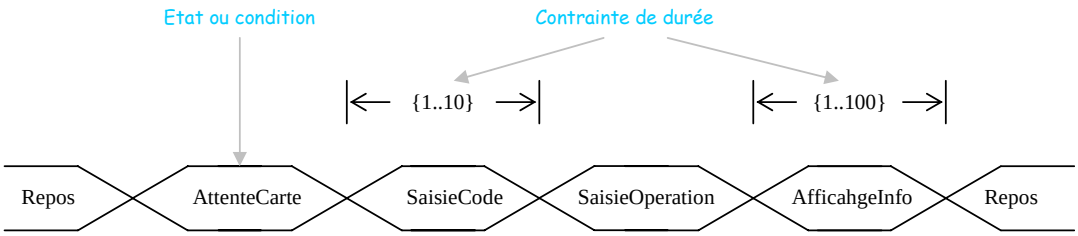

Fig. 16. Exemple de diagramme de temps avec un seul axe temporel

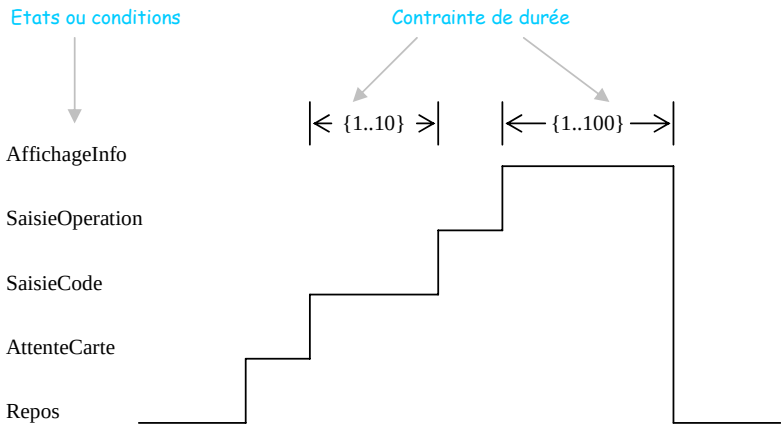

Fig. 17a. Exemple 1 de diagramme de temps avec un axe temporel par état/condition

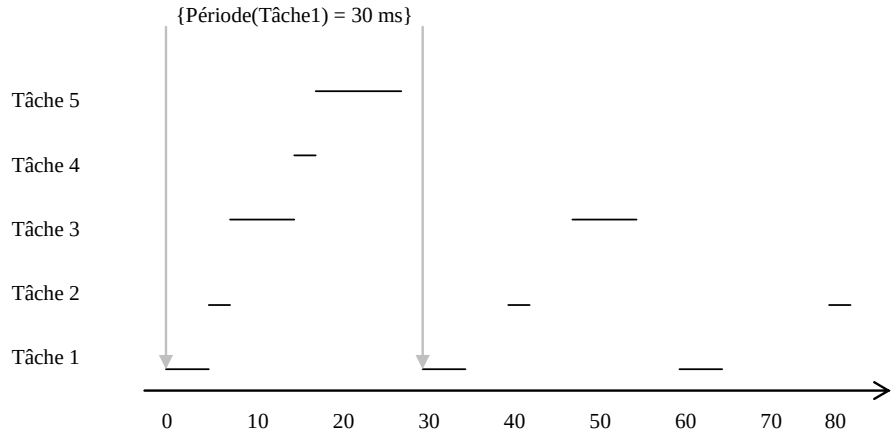

Fig. 17b. Exemple 2 de diagramme de temps montrant des tâches périodiques

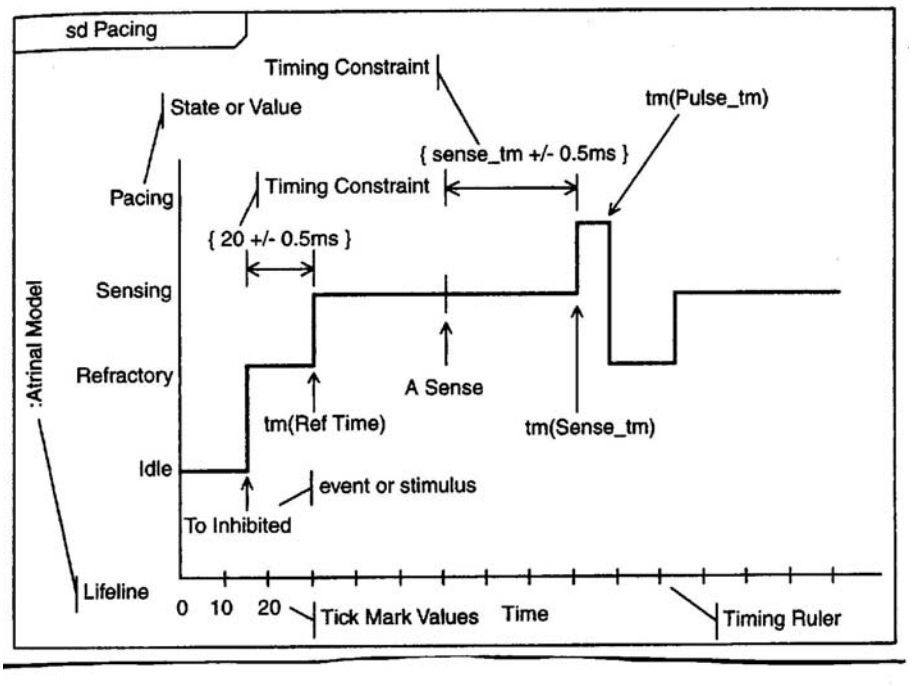

Fig. 17c. Exemple 3 de diagramme de temps

Exercice

Reprendre l'exemple du digicode vu précédemment en rajoutant les éléments suivants au cahier des charges :

- A l'initialisation un voyant vert s'allume (pour dire que la saisie de code peut être prise en compte par le système).
- La porte est verrouillée et déverrouillée par un actionneur (acteur externe au système). On suppose que les opérations de verrouillage et déverrouillage sont instantanées (durée nulle).
- Lorsque le code est bon, un ordre de déverrouillage de la porte est envoyé vers l'actionneur qui déverrouille la porte. On suppose que la porte reste déverrouillée pendant 1 minute avant de la verrouiller à nouveau.
- Après trois erreurs, un voyant rouge est activé, ensuite le système attend 2 minutes avant d'éteindre le voyant rouge et d'allumer le voyant vert afin d'accepter à nouveau la saisie de code.

1. Donner le cas d'utilisation typique (la porte s'ouvre), un cas d'utilisation avec plusieurs essais (la porte s'ouvre) et un cas d'utilisation anormal (la porte ne s'ouvre pas).
2. Donner les diagrammes de séquence correspondant aux cas d'utilisations précédents.
3. Donner le *statechart* correspondant au fonctionnement normal du système (cas d'utilisation typique).