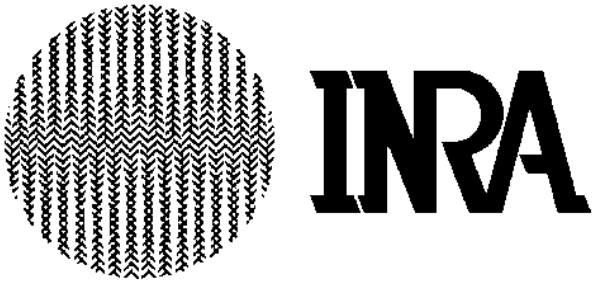


Institut National de la Recherche Agronomique

Unité de Biométrie et Intelligence Artificielle
BP 52627, 31326 Castanet Tolosan cedex, France

Fondements ontologiques des systèmes pilotés

Roger Martin-Clouaire, Jean-Pierre Rellier

Juin 2012

Table des matières

1. Introduction

Une ontologie définit rigoureusement un vocabulaire décrivant les concepts et structures utiles à l'explicitation d'une réalité factuelle dans un certain domaine et, éventuellement (comme il en sera le cas ici), les mécanismes généraux permettant d'aborder un questionnement sur cette réalité. Comme pour toute entreprise de modélisation, le choix de ce qu'il convient de spécifier et de comment le faire dépend avant tout de l'objectif visé, c'est-à-dire du questionnement qu'on entend aborder. Ici, le domaine est celui d'une production pilotée (par exemple et en particulier agricole) et la tâche visée est la prévision de son évolution au cours d'une `période`, sous une hypothèse de conditions exogènes (incontrôlables) fixée. On suppose qu'il existe une autorité unique assumant la supervision globale de la production.

L'analyse que nécessite la construction d'une ontologie clarifie la structure de la connaissance. Une ontologie permet ensuite de communiquer sur ce domaine et de rendre partageables les connaissances le concernant. Un autre rôle important d'une ontologie est de guider l'acquisition des connaissances relatives à une instance d'application particulière (par exemple, la production d'un certain type de tomate avec certains moyens matériels et humains, avec une stratégie de conduite particulière et sous des conditions exogènes particulières).

Une ontologie contient des termes qui appartiennent à différentes catégories lexicales. Ces catégories sont définies dans la section 2 par ce qu'on appelle les méta-concepts de l'ontologie. La section 3 définit des concepts généraux, à un niveau d'abstraction suffisant pour couvrir le domaine des systèmes en général. Nous les utiliserons largement ensuite pour définir les termes spécifiques à notre domaine : la section 4 présente ainsi le point de vue structurel sur un système piloté. Dans une perspective de simulation du comportement du système, la section 5 précise les points de vue fonctionnel et dynamique, par lesquels on perçoit comment et quand le système évolue.

2. Méta-concepts

Un TERME est un élément du discours sur le domaine. Il possède une définition dont la portée est limitée au domaine.
Certains TERMES désignent des concepts par lesquels on perçoit la réalité factuelle dans le domaine, par ses aspects statiques et dynamiques. Ils sont nommés en majuscules non soulignées.
D'autres TERMES désignent un des mécanismes, plus ou moins agrégés, par lesquels on exploite ces concepts pour aborder une tâche particulière (ici, la simulation). Ces mécanismes sont généraux, au sens où leur définition ne fait aucunement référence à un domaine particulier. Ils sont nommés en majuscules soulignées.
Les éléments du discours nommés en italiques sont des noms propres, donnés à certains TERMES (par exemple les noms donnés aux PROPRIETES -*cf.* § 3.1- des objets).
Les éléments du discours nommés en caractères « courrier » sont des valeurs symboliques d'ATTRIBUTS, dont l'interprétation est en général éclaircie par le contexte.
Tous les autres éléments du discours, y compris ceux déjà utilisés jusqu'ici, véhiculent leur sens courant, non spécifique à ce texte.

Une CLASSE est la spécification abstraite commune à un ensemble de réalisations concrètes d'un concept, lesquelles sont appelées les INSTANCES de la CLASSE. La spécification est constituée d'un ensemble d'ATTRIBUTS et d'un ensemble de METHODES.

Un ATTRIBUT représente un aspect de la description des INSTANCES d'une CLASSE. La connaissance sur la description est répartie sur plusieurs FACETTES de l'ATTRIBUT.

Un ATTRIBUT DE CLASSE représente un aspect de la description de la CLASSE en tant que telle. Il ne permet pas de différencier les INSTANCES de la CLASSE.

La VALEUR d'un ATTRIBUT (respectivement ATTRIBUT DE CLASSE) est une grandeur numérique ou un symbole qui caractérise une INSTANCE (resp. une CLASSE), lorsqu'on la considère du seul point de vue de cet ATTRIBUT.

Une FACETTE précise une partie de ce qu'on connaît sur un ATTRIBUT en tant que tel.
- `value` est la FACETTE par laquelle on peut connaître la VALEUR de l'ATTRIBUT pour une INSTANCE particulière,
- `domain`, `value_type`, `cardinality`, `default_value` établissent des contraintes sur `value`,
- `active_value_procedure` est l'éventuelle référence à une PROCEDURE qui décrit ce qui doit être fait lorsqu'on accède à la VALEUR de l'ATTRIBUT (METHODE *get-value*), ou lorsqu'on établit la VALEUR de l'ATTRIBUT (METHODE *set-value*).

Le mécanisme d'INVOCATION REFLEXE est celui qui déclenche la PROCEDURE référencée dans la FACETTE `active_value_procedure` d'un ATTRIBUT, si une telle référence a été mentionnée.

Une METHODE représente une propriété fonctionnelle de la CLASSE ou des INSTANCES de la CLASSE. Sa VALEUR est une référence à une FONCTION, à un PREDICAT ou à une PROCEDURE.

Le mécanisme d'INVOCATION INTENTIONNELLE est celui qui déclenche la FONCTION, le PREDICAT ou la PROCEDURE référencé(e) dans une METHODE. Il suppose l'existence d'une INSTANCE qui joue le rôle de client pour la connaissance fonctionnelle contenue dans la METHODE.

Une PROCEDURE est le programme de ce qui doit être fait lors de son INVOCATION[1].

Une FONCTION est une PROCEDURE qui renvoie à l'auteur de l'INVOCATION INTENTIONNELLE une INSTANCE d'une certaine CLASSE. Ce qui est renvoyé est la VALEUR de la FONCTION. La CLASSE définit l'ensemble des VALEURS possibles.

Le mécanisme d'EVALUATION est le mécanisme d'INVOCATION INTENTIONNELLE d'une FONCTION.

Un PREDICAT est une FONCTION qui renvoie une des deux INSTANCES de la CLASSE des booléens (`true, false`).

[1] On parlera du mécanisme d'INVOCATION, lorsque le propos s'applique aussi bien à l'INVOCATION INTENTIONNELLE qu'à l'INVOCATION REFLEXE.

Une RELATION est un lien sémantique entre INSTANCES ou entre CLASSES, véhiculé par un ATTRIBUT ou une METHODE de type FONCTION (ou PREDICAT). La RELATION lie l'INSTANCE (ou la CLASSE) avec la VALEUR de l'ATTRIBUT ou de la FONCTION.
Deux CLASSES peuvent être liées par une RELATION de *particularisation*, au sens où si la CLASSE *B* particularise la CLASSE *A*, toute INSTANCE de *B* satisfait la spécification définissant *A*. Pour la CLASSE *B*, l'ATTRIBUT DE CLASSE *est_un* vaut alors *A*.

Toutes les CLASSES possèdent une METHODE (dont la VALEUR est une référence à une FONCTION) qui crée et renvoie une INSTANCE de la CLASSE. Le mécanisme d'INSTANCIATION est celui qui réalise l'INVOCATION de cette METHODE.

3. Ontologie des systèmes

La réalité factuelle objet d'une étude peut être perçue ou non avec un point de vue systémique. Dans les deux cas, la perception repose sur un petit nombre de concepts de base, parmi lesquels notamment ceux d'ENTITE, de PROCESSUS et d'EVENEMENT (section 3.1), desquels dérivent la plupart des autres. Ils correspondent respectivement aux aspects structurel, fonctionnel et dynamique de la réalité. Il faut voir les extensions aux systèmes (section 3.2) puis aux systèmes pilotés (section 4) comme deux spécialisations successives des concepts de base. Le domaine considéré est représenté par l'ensemble des CLASSES correspondant aux spécialisations de ces concepts de base rencontrées dans le domaine.

3.1 Concepts de base

La CLASSE ENTITE représente un objet, dans le sens vague de chose, tangible ou intangible. Parmi les ATTRIBUTS d'une ENTITE, deux expriment une RELATION particulière :

- une liste d'ENTITES, qui définit une RELATION de *composition* : les ENTITES de la liste sont les COMPOSANTS de l'INSTANCE de la CLASSE (on parlera d'ENTITE COMPOSEE),
- une autre liste d'ENTITES, qui définit une RELATION *ensembliste* : les ENTITES de la liste sont les ELEMENTS de l'INSTANCE de la CLASSE (on parlera d'ENSEMBLE).

Une ENTITE ne peut posséder à la fois des COMPOSANTS et des ELEMENTS, c'est-à-dire être à la fois ENTITE COMPOSEE et ENSEMBLE. Il doit exister une CLASSE telle que tous les ELEMENTS en soient des INSTANCES, directes ou indirectes.
Les autres ATTRIBUTS sont dénommés les PROPRIETES de l'ENTITE.
Une ENTITE ou une PROPRIETE peuvent être dotées d'un mécanisme d'INVOCATION REFLEXE, au moyen d'une SPECIFICATION D'INVOCATION REFLEXE.

L'ETAT d'une ENTITE[2] à un INSTANT donné est défini par les VALEURS de ses PROPRIETES, l'ensemble des ENTITES qu'elle inclut (comme COMPOSANTS ou ELEMENTS) et par leurs ETATS. L'ETAT d'une ENTITE peut changer par l'effet de PROCESSUS.

La CLASSE SPECIFICATION D'ENSEMBLE D'ENTITES permet de spécifier un ensemble d'ENTITES, de façon concise, c'est-à-dire sans les désigner explicitement. Ses ATTRIBUTS sont une CLASSE d'ENTITES, un PREDICAT de sélection, un *effectif* dont la VALEUR est soit un entier `n` soit le symbole *all*, une METHODE *spécifie-effectif* et un *mode*.
Le PREDICAT de sélection spécifie les contraintes que doivent respecter l'ETAT des éléments de l'ensemble d'ENTITES, et éventuellement leur nature (c'est-à-dire une *particularisation* de la CLASSE d'ENTITES).
Si la VALEUR `n` n'est pas directement affectée à la PROPRIETE *effectif*, elle est calculée par l'INVOCATION de la METHODE *spécifie-effectif* (cela permet notamment de déterminer `n` en fonction de l'ETAT d'ENTITES).
La CLASSE est dotée d'une METHODE *expansion*, qui génère un ensemble d'ENTITES qui respectent le PREDICAT de sélection, et de taille `n`.
Si le *mode* est `return_instances`, les `n` ENTITES sont choisies arbitrairement parmi les INSTANCES existantes de la CLASSE d'ENTITES qui respectent le PREDICAT de sélection. Si la VALEUR de *effectif* est *all*, la METHODE *expansion* renvoie toutes les INSTANCES existantes qui respectent le PREDICAT de sélection.
Si le *mode* est `instantiate`, les ENTITES sont le résultat de `n` INSTANTIATIONS de la CLASSE d'ENTITES. La VALEUR *all* est incompatible avec ce *mode*, sauf lorsque la CLASSE est une *particularisation* de HOMOGENEOUS AGGREGATED RESOURCE (voir l'interprétation particulière du *all* dans l'article sur cette ENTITE).
Le mécanisme d'EXPANSION consiste simplement en une INVOCATION de la METHODE *expansion*.

La CLASSE SPECIFICATION DE DOMAINE DE VALEURS permet de spécifier un ensemble de VALEURS, de façon concise, c'est-à-dire sans les désigner explicitement. Ses ATTRIBUTS sont une référence à une PROPRIETE pour laquelle on spécifie un domaine de valeurs, la définition d'un ensemble de valeurs référence et un opérateur qui spécifie si le domaine est l'ensemble de référence ou bien son complémentaire. La METHODE *is-in-domain*, portant sur une ENTITE, détermine si la VALEUR de la PROPRIETE est, pour cette ENTITE, dans le domaine spécifié ou non.

La CLASSE SPECIFICATION D'INVOCATION REFLEXE permet de doter une ENTITE ou une de ses PROPRIETE d'un mécanisme d'INVOCATION REFLEXE.
Si l'INVOCATION REFLEXE doit intervenir quand la liste des COMPOSANTS ou des ELEMENTS d'une ENTITE est modifiée, les ATTRIBUTS de la présente CLASSE sont une SPECIFICATION D'ENSEMBLE D'ENTITES, un PREDICAT sur l'ETAT des ENTITES et une PROCEDURE *action-réflexe*. Si l'INVOCATION REFLEXE doit intervenir quand la VALEUR d'une PROPRIETE est modifiée, les

[2] En fait, d'une INSTANCE d'ENTITE. Le mot « ENTITE » sera employé pour l'expression « INSTANCE de la CLASSE ENTITE » chaque fois que le contexte permet de lever une éventuelle ambigüité.

ATTRIBUTS sont le nom de cette PROPRIETE et, encore là, une SPECIFICATION D'ENSEMBLE D'ENTITES, un PREDICAT sur l'ETAT des ENTITES et une PROCEDURE *action-réflexe*.
Cette CLASSE possède une METHODE *installation*, qui opère d'abord une EXPANSION de la SPECIFICATION D'ENSEMBLE D'ENTITES, si elle est dotée d'un PREDICAT de sélection. Pour chaque ENTITE renvoyée, soit la liste des COMPOSANTS ou des ELEMENTS, soit la PROPRIETE, est dotée d'une FACETTE `active_value_procedure`, dont le programme consiste à invoquer le PREDICAT sur l'ETAT des ENTITES puis, s'il est vérifié, la PROCEDURE *action-réflexe*. Cette PROCEDURE spécifie une conséquence quelconque de la modification de la liste des COMPOSANTS ou des ELEMENTS de l'ENTITE ou de la VALEUR de la PROPRIETE. L'INVOCATION REFLEXE est alors dite `activée`.
Si la SPECIFICATION D'ENSEMBLE D'ENTITES n'est pas dotée d'un PREDICAT de sélection, la METHODE *installation* ne fait que doter sa CLASSE d'ENTITES d'une référence à la présente SPECIFICATION D'INVOCATION REFLEXE. Ainsi, lors de toute INSTANCIATION de la CLASSE D'ENTITES, l'ENTITE créée est traitée comme ci-dessus dans le cas où le PREDICAT de sélection est présent.
La METHODE *désinstallation* rend l'INVOCATION REFLEXE `désactivée`.
L'INSTALLATION est le mécanisme qui réalise l'INSTANCIATION d'une SPECIFICATION D'INVOCATION REFLEXE et l'INVOCATION INTENTIONNELLE de sa METHODE *installation*. La DESINSTALLATION est le mécanisme qui réalise l'INVOCATION INTENTIONNELLE de sa METHODE *désinstallation.*

La CLASSE PROCESSUS permet de spécifier la manière avec laquelle évolue l'ETAT d'une ENTITE en réponse à l'OCCURRENCE d'un EVENEMENT. Cette évolution peut provoquer la PROGRAMMATION d'autres EVENEMENTS à des INSTANTS ultérieurs, notamment par INVOCATION REFLEXE installée sur l'ETAT de l'ENTITE. Les ATTRIBUTS d'un PROCESSUS sont l'ENTITE, sur laquelle il est dit « porter » et qui est dite son « objet », et une description de l'évolution de son ETAT. Cette description diffère selon qu'il s'agit d'un PROCESSUS PONCTUEL ou d'un PROCESSUS CONTINU.

Un PROCESSUS PONCTUEL est un PROCESSUS tel que la description de l'évolution de l'ETAT de l'ENTITE sur laquelle il porte est réalisée dans une seule PROCEDURE, VALEUR de la METHODE *procédure-exécution*. Le changement d'ETAT de l'ENTITE spécifié dans *procédure-exécution* intervient complètement à l'INSTANT même de son INVOCATION INTENTIONNELLE, conséquence de l'OCCURRENCE d'un EVENEMENT.

Un PROCESSUS CONTINU est un PROCESSUS tel que la description de l'évolution de l'ETAT de l'ENTITE sur laquelle il porte est réalisée dans trois PROCEDURES, VALEURS des METHODES *procédure-initialisation*, *procédure-poursuite* et *procédure-arrêt*.
Une INVOCATION de *procédure-initialisation* contraint l'ENTITE à posséder un ETAT compatible avec la première INVOCATION de *procédure-poursuite*.
L'évolution de l'ETAT est réalisée ensuite par des INVOCATIONS successives de *procédure-poursuite*. Le changement d'ETAT spécifié dans *procédure-poursuite* intervient à l'INSTANT de son INVOCATION. Il est opéré sur la base de l'ETAT à l'INSTANT de l'INVOCATION précédente, c'est-à-dire antérieur d'un délai VALEUR de l'ATTRIBUT *pas*. Deux INVOCATIONS successives de *procédure-poursuite* sont séparées par un intervalle dont la longueur est le *pas*. Le *pas* d'un PROCESSUS CONTINU est donc le plus grand intervalle de temps tel que les ETATS entre ses bornes ne soient jamais d'aucun intérêt pour le questionnement abordé dans le domaine.
L'arrêt de l'évolution est réalisé par une INVOCATION de *procédure-arrêt*.
Les METHODES *procédure-initialisation* et *procédure-arrêt* sont invoquées une et une seule fois. La METHODE *procédure-poursuite* est invoquée une fois juste après l'INVOCATION de *procédure-initialisation*, puis à chaque *pas* du PROCESSUS CONTINU jusqu'à l'OCCURRENCE d'EVENEMENTS incidents à traiter. Dans cette situation, on réalise la PROGRAMMATION, juste après les EVENEMENTS à traiter, d'un EVENEMENT provoquant une nouvelle série d'INVOCATIONS de *procédure-poursuite*. Il y a un seul EVENEMENT provoquant une série d'INVOCATIONS de *procédure-poursuite* si aucune OCCURRENCE d'EVENEMENT n'intervient avant l'INVOCATION de *procédure-arrêt*.

La CLASSE EVENEMENT permet de spécifier des directives de contrôle sur des PROCESSUS. Les directives sont appliquées à un certain INSTANT. Les ATTRIBUTS d'un EVENEMENT sont :
- les INSTANTS au plus tôt et au plus tard d'application des directives, autrement dit d'occurrence,
- l'INSTANT d'application des directives, qui est soit directement spécifié soit calculé aléatoirement entre les deux INSTANTS ci-dessus, par la METHODE ci-dessous
- une METHODE dont la VALEUR est une FONCTION qui calcule l'INSTANT d'occurrence, s'il n'est pas directement spécifié. La VALEUR par défaut calcule l'INSTANT sur la base de la loi de probabilité uniforme.
- la *probabilité* d'occurrence,
- un *degré-de-priorité*, entre *0* et *1*, utilisable pour ordonner des EVENEMENTS programmés au même INSTANT, et
- un ensemble de couples composés d'une INSTANCE de PROCESSUS et du nom d'une METHODE de PROCESSUS, parmi *procédure-exécution*, *procédure-initialisation*, *procédure-poursuite* et *procédure-arrêt*.

L'ordonnancement dans le temps et la synchronisation des directives passent par les mécanismes de PROGRAMMATION et d'OCCURRENCE.

Les EVENEMENTS possèdent une METHODE *post-consequence* dont la VALEUR est une référence à une PROCEDURE. Cette PROCEDURE est déclenchée à la fin du mécanisme d'OCCURRENCE. Son contenu est libre.
Les EVENEMENTS possèdent une METHODE *procédure-autogénération* dont la VALEUR est une référence à une PROCEDURE. Cette PROCEDURE est déclenchée à la fin du mécanisme d'OCCURRENCE seulement si l'EVENEMENT a été déclaré autogénérable. Elle réalise la PROGRAMMATION d'un autre EVENEMENT à un INSTANT postérieur. La *procédure-autogénération* a par défaut une VALEUR telle que c'est un EVENEMENT de la même CLASSE qui est généré. Le nouvel EVENEMENT est programmé sur la base des VALEURS des PROPRIETES complémentaires suivantes :

- les INSTANTS au plus tôt et au plus tard d'occurrence,
- une METHODE dont la VALEUR est une FONCTION qui calcule l'INSTANT d'occurrence. La VALEUR par défaut calcule l'INSTANT sur la base de la loi de probabilité uniforme.
- la *probabilité* d'occurrence.

Le mécanisme d'AUTOGENERATION est celui qui fait l'INVOCATION INTENTIONNELLE de la méthode *procédure-autogénération* d'un EVENEMENT.

Le mécanisme de PROGRAMMATION est celui par lequel un EVENEMENT est placé dans un AGENDA, en fonction croissante du point sur l'axe du temps auquel doit intervenir son OCCURRENCE. Une INSTANCE d'INSTANT est alors créée avec ce point comme VALEUR du *time-point*. Si d'autres EVENEMENTS sont réputés intervenir en d'autres points du segment représenté par cet INSTANT, on leur associe des INSTANTS partageant la même VALEUR de *time-point*. Dans ce cas, la position absolue des points est ignorée. Si on dispose de la connaissance des positions relatives, on l'exprime par les *degrés-de-priorité*, qui servent alors à ordonner les EVENEMENTS dans l'AGENDA.

Le mécanisme d'OCCURRENCE est celui par lequel un EVENEMENT est enlevé d'un AGENDA, à la condition nécessaire qu'il s'y trouve en tête. Il provoque l'INVOCATION INTENTIONNELLE immédiate des METHODES dont les noms définissent l'EVENEMENT (et qui correspondent aux directives), sur les PROCESSUS auxquels elles sont associées dans la définition. Si l'EVENEMENT a été déclaré autogénérable, l'AUTOGENERATION est invoquée comme dernière étape de l'OCCURRENCE.

La CLASSE AGENDA possède un ATTRIBUT dont la VALEUR est un ensemble d'EVENEMENTS classés par ordre croissant des INSTANTS auxquels ils sont programmés, et des *degrés-de-priorité* en cas d'égalité des INSTANTS. L'AGENDA d'une ENTITE désigne l'ensemble des EVENEMENTS intervenant sur les PROCESSUS qui portent sur l'ENTITE, sur ses COMPOSANTS ou sur ses ELEMENTS.

La CLASSE INSTANT permet de représenter un segment de l'axe du temps, c'est-à-dire un ensemble de points successifs infiniment proches deux à deux. A chacune de ses INSTANCES, appelée aussi un INSTANT, correspond d'une part un ensemble d'EVENEMENTS dont l'OCCURRENCE est programmée dans ce segment, et d'autre part un des ETATS des ENTITES, observable en ce segment.
La VALEUR de l'ATTRIBUT *time-point* est un des points choisi arbitrairement de ce segment. Par commodité, elle est appelée la VALEUR de l'INSTANT. Le TERME INSTANT sera aussi utilisé dans le texte pour désigner cette VALEUR. Deux INSTANTS sont égaux si leurs VALEURS sont égales.

Le mécanisme de SIMULATION porte sur une ENTITE à laquelle on a associé un AGENDA, comme VALEUR d'une PROPRIETE *agenda-événements*. L'ENTITE qui joue ce rôle est dénommée l' « entité simulée ». Le mécanisme consiste en la PROGRAMMATION d'au moins un EVENEMENT dans l'AGENDA de l'entité simulée. Puis on déclenche l' OCCURRENCE de l'EVENEMENT de tête, tant qu'il en existe un (*cf.* section 5.1). Cette OCCURRENCE peut avoir pour conséquence d'insérer de nouveaux EVENEMENTS dans l'AGENDA (notamment par le mécanisme d'AUTOGENERATION).

3.2 Extension aux systèmes

Un SYSTEME est une ENTITE qui possède deux PROPRIETES dont les VALEURS sont un AGENDA D'EVENEMENTS et un GESTIONNAIRE D'INFORMATION. En tant qu'ENTITE, un SYSTEME peut être constitué de plusieurs SYSTEMES (sous-SYSTEMES), qui en sont les COMPOSANTS ou les ELEMENTS.
Certains PROCESSUS portant sur un SYSTEME sont des PROCESSUS D'ENTREE-SORTIE.
L'AGENDA d'un SYSTEME comprend, par ordre croissant des INSTANTS d'OCCURRENCE, les EVENEMENTS contrôlant les PROCESSUS portant sur le SYSTEME ou sur des ENTITES qui le constituent. A tout INSTANT, l'EVENEMENT considéré est, dans l'AGENDA du SYSTEME et ceux des SYSTEMES composants, celui dont l'INSTANT d'OCCURRENCE est le plus proche.

Le TERME ENVIRONNEMENT désigne, pour un SYSTEME, la partie de l'univers extérieure au SYSTEME, et qui interagit avec lui par des PROCESSUS D'ENTREE et de SORTIE, et les EVENEMENTS de son AGENDA.

Un PROCESSUS D'ENTREE a pour effet d'inclure de la matière ou de l'énergie[3] dans le SYSTEME à partir de l'ENVIRONNEMENT (mise en place d'un COMPOSANT ou ajout d'un ELEMENT, ou bien modification, interprétable comme un ajout, de la VALEUR d'une PROPRIETE).

Un PROCESSUS DE SORTIE a pour effet d'exclure de la matière ou de l'énergie, du SYSTEME vers l'ENVIRONNEMENT (retrait d'un COMPOSANT ou d'un ELEMENT, ou bien modification, interprétable comme un retrait, de la VALEUR d'une PROPRIETE).

Un PROCESSUS DE TRANSFERT a pour effet d'exclure de la matière ou de l'énergie d'un SYSTEME et de l'inclure dans un autre SYSTEME.

Le GESTIONNAIRE D'INFORMATION est une ENTITE composée d'une TABLE D'EXPORTATION, d'une TABLE D'IMPORTATION, d'une MEMOIRE et d'un ensemble de références aux ALARMES qu'elle peut exploiter.

La TABLE D'EXPORTATION est une ENTITE dont les ELEMENTS sont tous les ASPECTS VISIBLES du SYSTEME. Elle possède une METHODE *exportation* portant sur un ASPECT VISIBLE, et dont l'INVOCATION INTENTIONNELLE par un autre SYSTEME (dans le mécanisme d'EXPORTATION) vérifie que l'ASPECT VISIBLE est demandable par ce SYSTEME, vérifie que son PREDICAT *condition-exportation* est satisfait, et renvoie enfin le résultat de l'EVALUATION de la FONCTION (sur l'ETAT) dont elle est dotée.

Un ASPECT VISIBLE d'un SYSTEME définit et modélise la manière de rendre visible un aspect de l'ETAT réel d'un SYSTEME, notamment via un capteur qui peut être imparfait. C'est une ENTITE caractérisée par :

- une PROPRIETE dont la VALEUR est la liste des autres SYSTEMES qui y ont accès ;
- une FONCTION sur l'ETAT du SYSTEME. La demande peut être soumise *via* un INDICATEUR. La VALEUR de la FONCTION est aussi appelée la VALEUR de l'ASPECT VISIBLE ;
- un PREDICAT *condition-exportation* sur l'ETAT du SYSTEME, qui doit renvoyer *true* pour que l'EXPORTATION de l'ASPECT VISIBLE soit possible.

La TABLE D'IMPORTATION est une ENTITE dont les ELEMENTS sont tous les INDICATEURS utilisés par le SYSTEME. Elle possède une METHODE *importation* portant sur un INDICATEUR, qui invoque d'abord le mécanisme d'EXPORTATION sur les ASPECTS VISIBLES en jeu dans l'INDICATEUR, puis invoque, sur les VALEURS renvoyées, la FONCTION dont la référence est la VALEUR de la METHODE *fonction-importation* de l'INDICATEUR. C'est le mécanisme d'IMPORTATION qui invoque la METHODE *importation*.

Un INDICATEUR d'un SYSTEME modélise la conception et la perception qu'il a d'un aspect de l'ETAT d'autres SYSTEMES, ou à une partie de sa propre MEMOIRE. C'est une ENTITE caractérisée par :

- une PROPRIETE dont la VALEUR est une liste d'ASPECTS VISIBLES d'autres SYSTEMES ;
- une FONCTION *fonction-importation* portant sur les VALEURS de ces ASPECTS VISIBLES et des FAITS enregistrés dans la MEMOIRE du SYSTEME. La VALEUR de la FONCTION est aussi appelée la VALEUR de l'INDICATEUR.

Une SPECIFICATION D'ALARME est une SPECIFICATION D'INVOCATION REFLEXE dont le mécanisme d'INSTALLATION installe des ALARMES.

Un mécanisme d'ALARME est une INVOCATION REFLEXE qui réalise[4] la PROGRAMMATION d'un EVENEMENT immédiat, dans l'AGENDA d'un SYSTEME[5].

La MEMOIRE est un ensemble de FAITS passés, enregistrés par le SYSTEME pour leur intérêt potentiel dans le futur.

Un FAIT rappelle, à la demande du SYSTEME, à quel INSTANT s'est produit, sur une ENTITE, un changement d'ETAT (décrit par les VALEURS avant et après l'occurrence du FAIT).

3.3 Structuration de l'espace

---

[3] Matière et Energie ne sont pas des TERMES de l'ontologie parce qu'ils peuvent être représentés, dans leurs rôles dans les PROCESSUS D'ENTREE-SORTIE, par ceux d'ENTITE et de PROPRIETE, sans pouvoir préciser, à ce stade de l'analyse, la nature d'une éventuelle particularisation. Ainsi les PROCESSUS D'ENTREE-SORTIE opèrent suivant deux modes : augmentation/diminution de la VALEUR d'une PROPRIETE représentant une quantité de matière ou d'énergie, ou bien addition/retrait d'une ENTITE de nature matérielle ou énergétique.

[4] On rappelle que ceci est réalisé par la PROCEDURE attachée à la FACETTE `active_value_procédure` par le mécanisme d'INSTALLATION, c'est-à-dire par la PROCEDURE *action-réflexe* de la SPECIFICATION D'ALARME.

[5] Cela représente l'envoi d'un « message », immédiatement pris en compte par le SYSTEME.

Une LOCATION est une ENTITE qui représente une portion d'espace.
C'est soit une PRIMITIVE LOCATION, soit une façon d'agréger des LOCATIONS (qui en sont alors les ELEMENTS) au moyen d'une RELATION de *positionnement* (*partition*, *inclusion*, *juxtaposition, superposition, recouvrement partiel*,…).
Une LOCATION est définie par la PROPRIETE *opérateur-sur-sous-locations*, qui symbolise la RELATION de *positionnement*. A chaque RELATION de *positionnement* correspond une *particularisation* de la CLASSE LOCATION.
Des *particularisations* de LOCATION peuvent être définies en les dotant des PROPRIETES géométriques ou topographiques (*surface-absolue*, *surface-relative*, *points-périmètre*, …) qui spécifient la portion d'espace. La *surface-relative* d'une INSTANCE `l` est le rapport de la *surface-absolue* (éventuellement non exprimée) de `l` à la *surface-absolue* de l'INSTANCE de LOCATION dont `l` est un ELEMENT.
Une LOCATION possède enfin une PROPRIETE *entités-résidentes*, dont la VALEUR est la liste, éventuellement vide, des ENTITES qui occupent la portion d'espace.
La METHODE *get-resident-entities*, argumentée par la référence à une CLASSE d'ENTITES `C`, renvoie, pour une INSTANCE de LOCATION `l`, la liste des INSTANCES de `C` qui sont soit une des *entités-résidentes* de `l`, soit une *entité-résidente* d'un ELEMENT (direct ou indirect) de `l`, soit un ELEMENT (direct ou indirect) d'une de ces *entités-résidentes*. Cette METHODE opère de façon spécifique pour chaque RELATION de *positionnement*. Pour une *partition* par exemple, c'est l'union des listes renvoyées par l'INVOCATION de *get-resident-entities* sur les ELEMENTS.
La METHODE *pre-split*, argumentée par une *particularisation* de LOCATION, et appliquée à une INSTANCE de LOCATION `l`, renvoie une INSTANCE de la *particularisation*, en donnant à celle-ci deux ELEMENTS `l'` et `l''` de la CLASSE de `l` et qui respectent ensemble les contraintes spatiales que respectait `l` avant l'INVOCATION de *pre-split*. La METHODE *pre-split* doit être dotée d'un contenu approprié aux propriétés souhaitées pour `l'` et `l''`.
La METHODE *pre-dispatch-resident-entities* est appliquée à une INSTANCE de LOCATION `l` et argumentée par une liste de *n* INSTANCES de LOCATIONS. Elle renvoie une liste de listes λ_i d'ENTITES, i=1,*n*, dans lesquelles sont réparties les références à toutes les *entités-résidentes* de l. Pour donner un sens particulier aux listes λ_i, la METHODE *pre-dispatch-resident-entities* doit être dotée d'un contenu spécifique. Par exemple :
. on peut coder *pre-dispatch-resident-entities* de telle sorte que, appliquée à `l` et argumentée avec la liste des LOCATIONS `l'` et `l''` construites par *pre-split* appliquée à `l`, les listes λ_i renvoyées puissent devenir VALEURS des *entités-résidentes* de `l'` et `l''`.
. on peut coder *pre-dispatch-resident-entities* de telle sorte que, appliquée à `l` et argumentée avec les ELEMENTS de `l`, les listes λ_i renvoyées puissent devenir VALEURS des *entités-résidentes* des ELEMENTS de `l`.
La CLASSE LOCATION n'est pas destinée à être instanciée, contrairement à toutes ses *particularisations*.

Une PRIMITIVE LOCATION est une LOCATION sans ELEMENTS. Elle ne correspond donc pas à une RELATION de *positionnement*.
La PROPRIETE *opérateur-sur-sous-locations* a la VALEUR `prim-loc`.
La METHODE *get-resident-entities*, argumentée par la CLASSE `C`, renvoie, pour une INSTANCE de PRIMITIVE LOCATION `pl`, la liste des INSTANCES de `C` qui sont une de ses *entités-résidentes*, ou ELEMENT à une profondeur quelconque de ces *entités-résidentes*.

Une PARTITION est une LOCATION en réalité constituée de tous ses ELEMENTS et seulement d'eux, sans qu'il existe une RELATION de *positionnement topographique* sur les ELEMENTS.
La PROPRIETE *opérateur-sur-sous-locations* a la VALEUR `partition`.

Une JUXTAPOSITION est une PARTITION telle que deux ELEMENTS ont une partie de frontière commune avec un et un seul autre, et tous les autres ont une partie de frontière commune avec exactement deux autres. Les ELEMENTS sont listés selon un ordre topographique pertinent.
La PROPRIETE *opérateur-sur-sous-locations* a la VALEUR `juxtaposition`.

Une INCLUSION est une LOCATION dont les ELEMENTS e_0, ...e_n sont tels que pour tout i > 0 la portion d'espace représentée par e_i est complètement incluse dans la portion d'espace représentée par e_{i-1}.

Une SUPERPOSITION est une PARTITION dont les ELEMENTS résultent de la superposition de deux autres PARTITIONS, dites « partitions parentes ». Chaque ELEMENT de la SUPERPOSITION est une portion d'espace commune entre un seul ELEMENT d'une partition parente et un seul ELEMENT de l'autre.
Les deux partitions parentes sont les VALEURS de deux PROPRIETES : *parent1* et *parent2*.
Pour construire la SUPERPOSITION, on considère un à un tous les couples constitués d'un ELEMENT de *parent1* et d'un ELEMENT de *parent2* ; si les deux ELEMENTS ont une portion d'espace commune, on ajoute à la liste des ELEMENTS de la SUPERPOSITION une LOCATION représentant la portion d'espace commune.
La SUPERPOSITION possède en outre deux PROPRIETES *liste-filiation1*et *liste-filiation2*. Pour chacune, la VALEUR est une liste de LOCATIONS dont la taille égale celle de la liste des ELEMENTS. La VALEUR de rang *k* dans la *liste-filiation1* est l'ELEMENT de la PARTITION *parent1* qui inclue l'ELEMENT de rang k de la *superposition*. *Idem* pour la *liste-filiation2*.

Un RECOUVREMENT PARTIEL est une LOCATION dont les ELEMENTS e_0, ...e_n sont tels que pour tout i < n e_i et e_{i+1} ont une portion d'espace commune. Cette portion d'espace est représentée par une LOCATION qui devient la VALEUR de la PROPRIETE *intersection*.

Le mécanisme de SCISSION d'une LOCATION consiste à invoquer sa METHODE *pre-split* puis à la remplacer, dans le CONTROLLED SYSTEM, par la LOCATION renvoyée.

3.4 Ensembles structurés d'individus

Un PEUPLEMENT est une ENTITE qui représente un ensemble d'individus[6], structuré en POPULATIONS qui en sont les ELEMENTS, et circonscrit dans un espace[7] qui lui donne une unité de lieu[8].
Un PEUPLEMENT possède comme PROPRIETE :

- l'*espace*, qui est, l'ENTITE dans laquelle est circonscrit le PEUPLEMENT, et qui en définit l'ENVIRONNEMENT par complémentarité.

Les VALEURS des PROPRIETES complémentaires que peut détenir une *particularisation* de PEUPLEMENT peuvent être le résultat de l'EVALUATION d'une FONCTION sur l'ETAT des POPULATIONS.

Un PEUPLEMENT MULTIPLE est une ENTITE dont les ELEMENTS sont des PEUPLEMENTS.

Une POPULATION est une ENTITE qui représente un groupe d'individus tous du même type, perçu comme une seule entité d'un point de vue fonctionnel. Ces individus sont des INSTANCES de la même CLASSE et sont supposés posséder des caractéristiques communes, que cela se traduise ou non par la même VALEUR de certaines de leurs PROPRIETES.
Le groupe d'individus est spécifié par la PROPRIETE suivante :

- la *classe-individus*, qui est la CLASSE d'ENTITES dont tous les individus sont des INSTANCES,

Une POPULATION possède en outre les PROPRIETES suivantes :

- *entité-spatiale*, qui, si elle est valuée, représente l'INSTANCE de LOCATION à laquelle est attachée la POPULATION. Sa PROPRIETE de *surface* est alors la taille de l'espace occupé par la POPULATION ;
- la *densité*, à VALEUR constante, qui est le nombre d'individus par unité de surface.

La VALEUR de certaines PROPRIETES complémentaires d'une POPULATION peut être le résultat de l'EVALUATION d'une FONCTION sur l'ETAT de ses ENTITES individus et/ou sur l'ETAT de l'éventuelle *entité-spatiale*.
Il est important de noter que bien qu'une POPULATION possède une PROPRIETE *entité-spatiale*, elle n'est pas nécessairement dotée d'une unité de lieu. Cette unité de lieu existe si la POPULATION est incluse dans un PEUPLEMENT ou si l'*entité-spatiale* est une INSTANCE directe ou indirecte d'une LOCATION qui présente une unité de lieu[9].
Une POPULATION est soit une POPULATION SIMPLE soit une POPULATION MULTIPLE. La CLASSE POPULATION n'est pas destinée à être instanciée, contrairement à ses *particularisations* terminales.

Une POPULATION MULTIPLE est une POPULATION dont les ELEMENTS sont des POPULATIONS.
L'*effectif* d'une POPULATION MULTIPLE peut être calculé à partir des *effectifs* de ses ELEMENTS.
La METHODE *pre-split*, appliquée à une POPULATION MULTIPLE pm, renvoie une POPULATION MULTIPLE dont les ELEMENTS sont deux autres INSTANCES de la CLASSE de pm : pm' et pm'', de telle sorte que chaque POPULATION dans pm figure une et une seule fois dans pm' ou pm''. La METHODE *pre-split* doit être dotée d'un contenu approprié à la manière de répartir les POPULATIONS de pm.

Une POPULATION SIMPLE est une POPULATION qui n'a pas d'ELEMENTS.
Ses PROPRIETES sont :

- l'*ensemble-représentatif*, qui a la forme d'une liste d'INSTANCES de la *classe-individus*, éventuellement réduite à un seul élément,
- l'*effectif*, dont la VALEUR entière est le nombre d'exemplaires de l'*ensemble-représentatif* dans la POPULATION réelle. Lorsque l'*effectif* est *1*, l'*ensemble-représentatif* est censé être la POPULATION entière, sinon il n'en est qu'un échantillon. La PROPRIETE *entité-spatiale*, plus précisément la PROPRIETE *surface* de celle-ci, permet de calculer l'*effectif*, s'il n'est pas directement spécifié.

---

6 Un individu est un « corps organisé vivant d'une existence propre et qui ne saurait être divisé sans être détruit. » (Le Petit Robert)

7 Selon la *particularisation* de PEUPLEMENT, il peut s'agir d'une surface ou d'un volume.

8 On dit qu'un PEUPLEMENT présente une unité de lieu lorsque tous ses individus sont considérés identiques du point de vue de leur localisation, absolue ou relative par rapport à un point de référence commun.

9 On dit qu'une LOCATION présente une unité de lieu lorsque tous ses points sont considérés identiques du point de vue de leur localisation, absolue ou relative par rapport à un point de référence commun.

Une POPULATION SIMPLE est soit une POPULATION SIMPLE HOMOGENE soit une POPULATION SIMPLE HETEROGENE. Elle n'est pas destinée à être instanciée, contrairement à ses deux *particularisations*.
La METHODE *pre-split*, appliquée à une POPULATION SIMPLE p, renvoie une POPULATION MULTIPLE dont les ELEMENTS sont deux autres INSTANCES de la CLASSE de p : p' et p'', de telle sorte que chaque individu dans p figure une et une seule fois dans p' ou p''. La METHODE *pre-split* doit être dotée d'un contenu approprié à la manière de répartir les individus de p. On peut en effet soit répartir les INSTANCES de l'*ensemble-représentatif* et donner aux *effectifs* de p' et p'' la VALEUR de l'*effectif* de p, soit donner aux *ensemble-représentatifs* de p' et p'' la VALEUR de l'*ensemble-représentatif* de p et donner aux *effectifs* de p' et p'' deux VALEURS dont la somme vaut l'*effectif* de p, soit mixer les deux procédures.

Une POPULATION SIMPLE HOMOGENE est une POPULATION SIMPLE telle que :
- l'*ensemble-représentatif* ne contient qu'une ENTITE, appelée l'« individu représentatif » de la POPULATION,
- l'*effectif* est donc le nombre entier d'exemplaires de l'individu représentatif dans la POPULATION.

Une POPULATION SIMPLE HETEROGENE est une POPULATION SIMPLE dont les individus ne sont pas identiques. Elle est donc telle que :
- l'*ensemble-représentatif* contient un ensemble d'ENTITES toutes supposées différentes.

Une POPULATION SIMPLE HETEROGENE possède en propre les deux PROPRIETES suivantes :
- la *liste-de-représentativité*, dont chaque élément correspond à une ENTITE de l'*ensemble-représentatif* et représente le nombre d'exemplaires de l'élément dans l'*ensemble-représentatif*.
- la *liste-de-représentativité-cumulée*, dont chaque élément correspond à une ENTITE de l'*ensemble-représentatif* , est appelé sa 'représentativité cumulée', et représente le cumul des nombres d'exemplaires des éléments, jusqu'à cette ENTITE comprise, dans l'*ensemble-représentatif*.

Ainsi, la PROPRIETE *effectif* est bien le nombre entier d'exemplaires de l'*ensemble-représentatif* dans la POPULATION, mais une fois qu'on l'a virtuellement complété selon la *liste-de-représentativité*.

Le mécanisme de SCISSION d'une POPULATION consiste à invoquer sa METHODE *pre-split* puis à la remplacer, dans le CONTROLLED SYSTEM, par la POPULATION MULTIPLE renvoyée.

4. Ontologie des systèmes pilotés

4.1 Système de production

Un PRODUCTION SYSTEM[10] est un SYSTEME dont les trois COMPOSANTS sont :

- un CONTROLLED SYSTEM,
- un OPERATING SYSTEM,
- un MANAGER.

Son GESTIONNAIRE D'INFORMATION est vide. Il ne possède pas de PROPRIETES spécifiques

Un CONTROLLED SYSTEM est un dispositif (naturel et/ou artificiel) siège des processus de production Dans un contexte qui lève l'ambiguïté avec le PRODUCTION SYSTEM, on l'appelle le « système piloté », au sens strict. C'est un SYSTEME dont l'AGENDA est nourri par l'OPERATING SYSTEM et l'ENVIRONNEMENT du PRODUCTION SYSTEM. Sa TABLE D'IMPORTATION est vide. Il est généralement doté d'ALARMES, et peut posséder une MEMOIRE[11]. Le dispositif piloté, spécifique à l'application, doit devenir la valeur de la PROPRIETE *entité-attachée* du CONTROLLED SYSTEM.

Un MANAGER est, au sein du PRODUCTION SYSTEM, un SYSTEME en charge du pilotage du CONTROLLED SYSTEM et de l'OPERATING SYSTEM. Il détient à cet effet une STRATEGY (qui en est l'unique COMPOSANT). On convient de considérer le pilotage comme résultant d'un ensemble d'activités (au sens commun) de pilotage[12].
Le MANAGER n'a pas de PROCESSUS D'ENTREE, de PROCESSUS DE SORTIE ni de PROCESSUS DE TRANSFERT. Les ASPECTS VISIBLES du MANAGER sont accessibles à l'OPERATING SYSTEM et portent sur les ACTIVITY SPECIFICATIONS à prendre en compte et sur les PREFERENCE RULES de sa STRATEGY. Il possède une MEMOIRE.
Il est l'objet de PROCESSUS qui mettent en œuvre les mécanismes CHECKING REACTIVE TRAJECTORY et UPDATING SITUATION. Ceux-ci ont les fonctions suivantes :

- mettre à jour la STRATEGY dans des situations reconnues à l'avance pour l'exiger (décrites dans la REACTIVE TRAJECTORY et les SPECIFICATIONS D'ALARMES),
- lire à certains INSTANTS cette STRATEGY,
- placer dans les ASPECTS VISIBLES du MANAGER une liste d'ACTIVITY SPECIFICATIONS correspondant à la préconisation des NOMINAL PLANS pour l'INSTANT courant, et
- placer dans l'AGENDA de l'OPERATING SYSTEM un EVENEMENT déclenchant les PROCESSUS de génération d'INSTRUCTIONS à partir de la liste d'activités préconisées.

Les INSTANTS de lecture de la STRATEGY sont déterminés par des EVENEMENTS dont le rythme d'OCCURRENCE est spécifié par la STRATEGY (ils sont à ce titre contrôlés) ou par des EVENEMENTS immédiats répondant à des ALARMES installées sur les deux autres SYSTEMES composant le PRODUCTION SYSTEM.
La METHODE *split-primitive-activity-spec* code une manière particulière de mettre à jour la STRATEGY, en scindant une activité en deux. La METHODE *split-operated-object-spec*, appliquée à une activité, partage en deux l'objet de l'activité. Ces deux METHODES seront décrites conjointement avec le mécanisme de SCISSION (*cf.* § 4.4).

Un OPERATING SYSTEM est un SYSTEME qui détermine et exécute les actions sur le CONTROLLED SYSTEM à partir des activités préconisées par le MANAGER. Son unique COMPOSANT est :

- une RESOURCE POOL DISJUNCTION, qui représente l'ensemble des CLASSES de RESOURCES et leurs INSTANCES pouvant être référencées dans la STRATEGY, et dont le cadre d'utilisation est spécifié par la STRATEGY.

Ce SYSTEME n'est pas l'objet de PROCESSUS D'ENTREE ni de PROCESSUS DE SORTIE.
Il est l'objet de PROCESSUS qui mettent en œuvre les mécanismes MAKING INSTRUCTION LIST et ACTING INSTRUCTION LIST. Ceux-ci mettent en œuvre les préconisations courantes de la STRATEGY du MANAGER sous la forme d'OPERATIONS qui affectent le CONTROLLED SYSTEM via des EVENEMENTS placés dans l'AGENDA de ce dernier, compte tenu de l'ETAT courant du PRODUCTION SYSTEM (en particulier de la disponibilité des RESOURCES).
Les EVENEMENTS qui contrôlent les PROCESSUS sont d'une part l'affichage par le MANAGER des préconisations de la STRATEGY et d'autre part certains EVENEMENTS portant sur les RESOURCES. Ces derniers EVENEMENTS sont d'origine externe au PRODUCTION SYSTEM (par exemple, indisponibilité d'une partie de la RESOURCE main d'œuvre pour des raisons exogènes) ou interne (par exemple, la fin d'exécution d'une OPERATION libère tout ou partie des RESOURCES qui lui étaient affectées).

---

[10] Le postulat de l'existence d'une supervision globale de la production implique que le domaine considéré (voir l'introduction) est constitué d'un et un seul PRODUCTION SYSTEM et de son ENVIRONNEMENT. Cette supervision est assumée par le MANAGER du PRODUCTION SYSTEM.

[11] Contrairement au MANAGER et à l'OPERATING SYSTEM, la constitution du CONTROLLED SYSTEM est entièrement définie par le modélisateur en fonction de la réalité du système piloté Il en est de même pour ses ASPECTS VISIBLES. Les ENTITES du système piloté doivent simplement être encapsulées (comme COMPOSANTS ou ELEMENTS) dans une ENTITE unique de la classe CONTROLLED SYSTEM.

[12] On ne fait pas de distinction entre les mots pilotage et conduite. On emploie avec le même sens le mot contrôle comme une traduction maintenant assez courante de l'anglais « control ».

La CLASSE INCONSISTENCY CONDITION permet d'exprimer une contrainte sur la réalisation de la STRATEGY du MANAGER. Cette contrainte a la forme d'une situation inacceptable ou irréalisable, quant à l'utilisation des RESOURCES pour la mise en œuvre des activités de pilotage. Elle n'est pas destinée à être instanciée, contrairement à ses deux *particularisations* que sont les CLASSES RESOURCE SHARING VIOLATION CONDITION et ACTIVITY INCONSISTENCY CONDITION. C'est une ENTITE sans COMPOSANTS ni ELEMENTS, définie par deux PROPRIETES :

- *classe-entité-contrainte*, dont la VALEUR est une *particularisation* d'ENTITE : toutes les INSTANCES de cette CLASSE sont alors sujettes à la contrainte ; selon la *particularisation*, c'est une RESOURCE ou une PRIMITIVE ACTIVITY SPECIFICATION ;
- un OCCURRENCE CROSS DOMAIN, interprété comme une situation inacceptable d'utilisation des RESOURCES.

Elle possède une METHODE *holding-condition* (un PREDICAT), qui décrit quelle condition sur l'ETAT du PRODUCTION SYSTEM doit être vérifiée pour que la contrainte soit prise en compte.

4.2 Ressources

4.2.1 Typologie

Une RESOURCE est une ENTITE matérielle requise par une INSTRUCTION pour qu'elle soit exécutable.
Une RESOURCE possède une PROPRIETE booléenne *état-de-disponibilité*, dont la valeur (`disponible` ou `non_disponible`) dépend d'AVAILABILITY CONSTRAINTS attachées à la RESOURCE. Une RESOURCE n'est utilisable que si son *état-de-disponibilité* est `disponible`. La RESOURCE est alors dite `disponible`.
Une RESOURCE possède une PROPRIETE *engagements*, dont la VALEUR est, à l'INSTANT courant, la liste des activités auxquelles elle est allouée. Si cette liste n'est pas vide, la RESOURCE est dite `engagée`.
Une RESOURCE possède une PROPRIETE *permamente*, dont la VALEUR est *vrai* par défaut. On peut donner la valeur *faux* pour des ressources qui ne font pas physiquement partie du système. C'est notamment le cas d'aggrégations de ressources créées de manière opportuniste et temporaire. Le moteur récupère lui-même la mémoire allouée à ces ressources quand elles ne sont plus en jeu dans une activité.
On attache enfin à la RESOURCE, *via* une PROPRIETE, une liste de RESOURCE SHARING VIOLATION CONDITIONS pour exprimer à quel point on peut partager cette RESOURCE au profit de plusieurs OPERATIONS ou de plusieurs autres RESOURCES simultanément. Cette liste rassemble les RESOURCE SHARING VIOLATION CONDITIONS telles que la *classe-entité-contrainte* soit une CLASSE dont la RESOURCE est une INSTANCE directe ou indirecte.
Une RESOURCE est une SINGLE RESOURCE ou une AGGREGATED RESOURCE.
Selon le mode de détermination de l'*état-de-disponibilité*, une RESOURCE est soit une DISCRETE STATE RESOURCE soit une CAPACITATED RESOURCE. Du point de vue de son usage, une RESOURCE est soit une REUSABLE RESOURCE soit une CONSUMABLE RESOURCE. Les deux points de vue sont complémentaires : il existe des DISCRETE-STATE REUSABLE RESOURCES, des CAPACITATED CONSUMABLE RESOURCES, etc.[13]

Une SINGLE RESOURCE (ou « ressource simple ») est une RESOURCE qui n'est pas l'assemblage d'autres RESOURCES. En d'autres termes ses PROPRIETES lui sont intrinsèques. Une ressource simple possède deux PROPRIETES dont les VALEURS sont, pour l'une une TIME AVAILABILITY CONSTRAINT, pour l'autre une liste de STATE AVAILABILITY CONSTRAINTS. Ces dernières doivent être satisfaites conjointement pour que la ressource soit disponible.

Une AGGREGATED RESOURCE est une RESOURCE qui est l'assemblage d'autres AGGREGATED RESOURCES ou SINGLE RESOURCES, qu'on appelle les « ressources éléments ». C'est une HOMOGENEOUS AGGREGATED RESOURCE ou une HETEROGENEOUS AGGREGATED RESOURCE. L'*état-de-disponibilité* d'une AGGREGATED RESOURCE est fonction de ses propres PROPRIETES et des PROPRIETES de ses ressources éléments. Lorsque des ressources éléments sont des ressources à capacité, la *capacité* et le *degré-de-disponibilité* d'une AGGREGATED RESOURCE sont chacun fonction de la *capacité* et du *degré-de-disponibilité* des ressources éléments. Ces fonctions sont spécifiques de chaque spécialisation de AGGREGATED RESOURCE[14].

Une HOMOGENEOUS AGGREGATED RESOURCE est une AGGREGATED RESOURCE dont les ELEMENTS sont soit des INSTANCES directes ou indirectes d'une même CLASSE de SINGLE RESOURCE, soit des INSTANCES de plusieurs *particularisations* d'HOMOGENEOUS AGGREGATED RESOURCE à condition qu'il existe une CLASSE dont héritent toutes ces *particularisations*. Elle ne possède pas de COMPOSANTS.
Une telle RESOURCE peut être spécifiée de deux manières différentes :

---

[13] Quelques *particularisations* standard de RESOURCES : un TOOL, un SINGLE WORKER, un OPERATED OBJECT sont des DISCRETE STATE REUSABLE RESOURCES, un DISPOSABLE-TOOL est une DISCRETE STATE CONSUMABLE RESOURCE, un STOCK est une CAPACITATED CONSUMABLE RESOURCE, un POTENTIAL est une CAPACITATED REUSABLE RESOURCE.

[14] Par exemples, si un STOCK est une CAPACITATED RESOURCE et si un ENSEMBLE DE STOCKS est une HOMOGENEOUS AGGREGATED RESOURCE, la *capacité* de l'ENSEMBLE DE STOCKS est la somme des *capacités* des STOCKS qu'il agrège ; si un PNEU et un MOTEUR sont des DISCRETE STATE RESOURCES et si une VOITURE est une HETEROGENEOUS AGGREGATED RESOURCE, la VOITURE est `disponible` si et seulement si ses quatre PNEUS et le MOTEUR sont `disponibles`.

- soit en indiquant explicitement la liste des `n` INSTANCES de la CLASSE des ELEMENTS ;
- soit en indiquant la CLASSE des ELEMENTS et leur *effectif* `n` ; dans ce cas seulement, l'INSTANCIATION établit elle-même la liste des `n` ELEMENTS, en les choisissant arbitrairement dans la liste des INSTANCES de la CLASSE des ELEMENTS, à quelque profondeur qu'elles se trouvent (c'est-à-dire que les INSTANCES choisies peuvent être des INSTANCES de *particularisations* de la CLASSE des ELEMENTS).

Lorsqu'une SPECIFICATION D'ENSEMBLE D'ENTITES porte sur une HOMOGENEOUS AGGREGATED RESOURCE, avec un *mode* `instantiate` et *all* pour *effectif*, l'EXPANSION génère la liste de toutes les INSTANCES de l'AGGREGATED RESOURCE qu'on peut construire à partir de l'ensemble des ressources élémentaires INSTANCES directes ou indirectes de la CLASSE[15].

Une HETEROGENEOUS AGGREGATED RESOURCE est une AGGREGATED RESOURCE qui possède des COMPOSANTS, INSTANCES de différentes CLASSES disjointes de RESOURCES[16], et pas d'ELEMENTS.
Une telle RESOURCE peut être spécifiée de deux manières différentes :

- soit en indiquant explicitement la liste des INSTANCES qui sont les COMPOSANTS ;
- soit en indiquant une liste de références à des CLASSES disjointes de RESOURCES ; dans ce cas seulement, l'INSTANCIATION établit elle-même la liste des COMPOSANTS en invoquant une INSTANCIATION de chaque CLASSE de RESOURCES.

Un RESOURCE SET est une ENTITE immatérielle qui permet de spécifier un ensemble de références à des RESOURCES. Ces références sont les ELEMENTS de l'ENTITE. Il est important de noter qu'un RESOURCE SET n'a pas les PROPRIETES d'une RESOURCE, contrairement à, par exemple, à une AGGREGATED RESOURCE.

Un RESOURCE POOL (ou pool de ressources) est un RESOURCE SET qui représente un ensemble de ressoures affectées au fonctionnement d'un PRODUCTION SYSTEM. Un pool de ressources est soit un SIMPLE RESOURCE POOL, soit une RESOURCE POOL DISJUNCTION.

Un RESOURCE POOL DISJUNCTION est une liste de RESOURCE POOLS tels qu'aucun élément de l'un ne se trouve dans un autre. Une telle entité est nécessairement placée en COMPOSANT d'un OPERATING SYSTEM.

Une DISCRETE-STATE RESOURCE est une SINGLE RESOURCE caractérisée par son ETAT qualitatif, et dont l'*état-de-disponibilité* est fonction de cet ETAT et des STATE-RELATED AVAILABILITY CONSTRAINTS attachées à la RESOURCE. Par exemple, une INSTANCE de la CLASSE *pneu* est une RESOURCE `disponible` si son ETAT de *pression* a une VALEUR comprise entre *1* et *2* bars, `non_disponible` sinon : ici, l'ETAT qualitatif a deux réalisations possibles. Une INSTANCE *voie* est `disponible` si la VALEUR de *feu-tricolore* est `vert` ou `orange`, `non_disponible` si la VALEUR est `rouge` : ici, l'ETAT qualitatif a trois réalisations possibles.

Une CAPACITATED RESOURCE est une SINGLE RESOURCE caractérisée par un vecteur de valeurs exprimant une *capacité* multidimensionnelle, et dont le *degré-de-disponibilité* est, à un INSTANT donné, le vecteur des valeurs courantes sur ces dimensions. L'*état-de-disponibilité* de la RESOURCE dépend du *degré-de-disponibilité* et des CAPACITY-RELATED AVAILABILITY CONSTRAINTS attachées à la RESOURCE. Par exemple, une INSTANCE de la CLASSE *multi-stock* contenant 10 ENTITES de CLASSE *P1* et 5 ENTITES de CLASSE *P2* est une RESOURCE `non_disponible` pour une INSTRUCTION qui requiert l'engagement de 6 ENTITES de CLASSE *P2*.

Une CONSUMABLE RESOURCE est une SINGLE RESOURCE exploitable une fois et une seule. En d'autres termes, chaque unité de *capacité* n'est engageable qu'une et une seule fois (CAPACITATED) ou bien l'engagement provoque une indisponibilité[17] définitive (DISCRETE-STATE). On dit que l'INSTRUCTION consomme la RESOURCE.

Un STOCK est une CAPACITATED CONSUMABLE RESOURCE munie d'une CAPACITE déterminée par différence entre la valeur courante que peut prendre une de ses PROPRIETES par les MECANISMES d'IMMOBILISATION et de RECHARGEMENT et ses VALEURS maximale et minimale.

---

15 Soit, par exemple, `A` une *particularisation* d'HOMOGENEOUS AGGREGATED RESOURCE, caractérisée par la CLASSE `H` et l'*effectif* 2. `H` possède deux particularisations `H1` et `H2`. Les INSTANCES de `H1` sont {h1, h3 }. Les INSTANCES de `H2` sont {h2, h4 }. L'EXPANSION de la SPECIFICATION D'ENSEMBLE D'ENTITES { `A`, *true*, *all*, `instantiate` } renvoie la liste { (h1 h2) (h1 h3) (h1 h4) (h2 h3) (h2 h4) (h3 h4) }. L'EXPANSION de la SPECIFICATION D'ENSEMBLE D'ENTITES { `A`, *true*, *1*, `instantiate` } peut renvoyer la liste { (h1 h2) } ou bien la liste { (h1 h3) }, etc. On verra que cette dernière alternative n'est pas neutre du point de vue de l'allocation de ressources.

16 C'est-à-dire qu'on ne peut pas indiquer, par exemple, une CLASSE A et une CLASSE B qui est une *particularisation* de A.

17 On nomme disponibilité et indisponibilité les situations d'une RESOURCE pour laquelle la VALEUR de la PROPRIETE *état-de-disponibilité* est respectivement `disponible` et `non_disponible`.

Une REUSABLE RESOURCE est une SINGLE RESOURCE qui redevient exploitable après qu'elle a été exploitée pour une INSTRUCTION. En d'autres termes, chaque unité de *capacité* est réengageable (CAPACITATED) ou bien l'engagement ne provoque pas une indisponibilité définitive (DISCRETE-STATE). On dit que l'INSTRUCTION utilise la RESOURCE.

4.2.2 Capacité des ressources

La capacité est une quantité, dont la valeur est relative à une certaine unité de mesure, qu'une CAPACITATED RESOURCE peut mettre à disposition d'une ou plusieurs activités pour leur réalisation. La capacité n'est effectivement utilisée que si la ou les activités en question la requièrent explicitement dans leur spécification. De façon générale, la capacité d'une CAPACITATED RESOURCE est en fait multidimensionnelle : la ressource met à disposition plusieurs quantités, chacune relative à une unité de mesure différente. Par exemple, tel type d'avion a une capacité d'accueil de 100 passagers et une capacité de stockage de 5 tonnes de bagages.
On distingue deux points de vue pour la capacité d'une ressource : la capacité qu'elle peut fournir et la capacité qu'il lui est demandé de fournir. Selon le premier point de vue, la VALEUR de la capacité est intrinsèque, autrement dit elle ne dépend que de la spécification de la RESOURCE indépendamment de toute éventuelle réquisition par une activité. Du second point de vue, la VALEUR de la capacité est déterminée par les activités dans lesquelles la RESOURCE est mise en jeu. C'est pourquoi une CAPACITATED RESOURCE possède les deux PROPRIETES de *capacité-fournie* et de *capacité-demandée*, dont les VALEURS sont deux INSTANCES de MULTIPLE RESOURCE CAPACITY.

Une MULTIPLE RESOURCE CAPACITY est une ENTITE qui définit une capacité multidimensionnelle. Ses ELEMENTS sont des INSTANCES de RESOURCE CAPACITY.

Une RESOURCE CAPACITY est une entité qui spécifie une capacité unidimensionnelle. Elle est nécessairement ELEMENT d'une INSTANCE de MULTIPLE RESOURCE CAPACITY.
Dans le cas général, les PROPRIETES d'une capacité sont :
- la référence à une PROPRIETE de la RESOURCE à laquelle la capacité sera rattachée. Comme cas particulier, si la capacité concerne la quantité d'ELEMENTS que la RESOURCE peut se voir ajouter ou enlever, cette référence est remplacée par le symbole `elements` ;
- une quantité sur la dimension de la PROPRIETE ci-dessus définie, interprétée comme la capacité fournie ou demandée. Selon le type de la PROPRIETE, cette quantité peut et doit être entière ou réelle.

Dans le cas particulier des STOCKS, les PROPRIETES d'une capacité sont :
- la référence à une PROPRIETE du STOCK, comme dans le cas général ;
- trois PROPRIETES aux sens de niveau inférieur, supérieur et courant du STOCK dans son rôle de RESOURCE ;
- deux capacités : la *capacité-en-entrée*, dont la VALEUR est la différence entre le *niveau-supérieur* et le *niveau-courant*, et la *capacité-en-sortie*, dont la VALEUR est la différence entre le *niveau-courant* et le *niveau-inférieur*.

4.2.3 Mécanismes

La REQUISITION est le mécanisme par lequel une RESOURCE `non_engagée` devient `engagée`, ou bien par lequel une RESOURCE `engagée` devient `engagée` pour la mise en œuvre d'une nouvelle activité. La REQUISITION d'une REUSABLE RESOURCE peut être suivie par une LIBERATION.

La LIBERATION est le mécanisme par lequel une RESOURCE `engagée` devient `non_engagée`, ou bien par lequel une RESOURCE reste `engagée`, mais plus pour toutes les activités pour lesquelles elle l'était.

Le RECHARGEMENT est le mécanisme par lequel le *degré-de-disponibilité* d'une CAPACITATED CONSUMABLE RESOURCE passe d'une certaine VALEUR à une VALEUR plus grande.

L'IMMOBILISATION est le mécanisme par lequel l'*état-de-disponibilité* d'une RESOURCE passe de la VALEUR `disponible` à la VALEUR `non_disponible`, non parce que la RESOURCE est requise par une INSTRUCTION, mais en conséquence d'un EVENEMENT dans l'ENVIRONNEMENT du PRODUCTION SYSTEM.

La MOBILISATION est le mécanisme par lequel l'*état-de-disponibilité* d'une RESOURCE passe de la VALEUR `non_disponible` à la VALEUR `disponible`, non pas comme conséquence de la fin de réalisation d'une INSTRUCTION, mais en conséquence d'un EVENEMENT dans l'ENVIRONNEMENT du PRODUCTION SYSTEM.

4.2.4 Contraintes

La CLASSE AVAILABILITY CONSTRAINT permet de spécifier des conditions qui régissent l'usage ou la consommation d'une RESOURCE. Une RESOURCE qui satisfait toutes les AVAILABILITY CONSTRAINTS qui lui sont attachées

(comme une VALEUR, sous forme d'une liste, d'une PROPRIETE) est dite `disponible`. C'est une TIME-RELATED AVAILABILITY CONSTRAINT, une STATE-RELATED AVAILABILITY CONSTRAINT ou une CAPACITY-RELATED AVAILABILITY CONSTRAINT.

Une TIME-RELATED AVAILABILITY CONSTRAINT est une AVAILABILITY CONSTRAINT qui doit être respectée par l'INSTANT courant pour que l'*état-de-disponibilité* des RESOURCES auxquelles elle est attachée ait la VALEUR `disponible à cet INSTANT`. Elle est définie par un ensemble d'intervalles entre deux INSTANTS. Typiquement, elle est satisfaite si et seulement si l'INSTANT courant se situe dans un des intervalles.

Une STATE-RELATED AVAILABILITY CONSTRAINT est une AVAILABILITY CONSTRAINT qui doit être respectée par l'ETAT d'une DISCRETE-STATE RESOURCE pour que son *état-de-disponibilité* ait la VALEUR `disponible`. Elle est définie par une conjonction de SPECIFICATIONS DE DOMAINES DE VALEURS et/ou un PREDICAT *state-related-predicate*. Elle est satisfaite si et seulement si cet éventuel PREDICAT est satisfait et si la RESOURCE satisfait la METHODE *is-in-domain* appliquée à chacune des SPECIFICATIONS DE DOMAINES DE VALEURS.

Une CAPACITY-RELATED AVAILABILITY CONSTRAINT est une AVAILABILITY CONSTRAINT qui doit être respectée par le *degré-de-disponibilité* d'une CAPACITATED RESOURCE pour que son *état-de-disponibilité* ait la VALEUR `disponible`. Elle est satisfaite si et seulement si le *degré-de-disponibilité* réel est dans un certain ensemble de VALEURS. Typiquement, elle n'est pas satisfaite si l'INSTRUCTION pour laquelle on teste la disponibilité d'une RESOURCE requiert son engagement (PROPRIETE *capacité-demandée* de la RESOURCE) au-delà du *degré-de-disponibilité* dans une quelconque de ses dimensions (PROPRIETE *capacité-fournie*).

`La CLASSE OCCURRENCE DOMAIN est utilisée pour décrire un domaine de situations d'engagement d'une RESOURCE ou d'une OPERATION. Il est défini par trois ATTRIBUTS dont les VALEURS sont :`

- `un identificateur de CLASSE d'ENTITE,`
- `un symbole de relation binaire (>, <, ...),`
- une valeur entière qui est la limite du domaine de situations d'engagement.

Soit `id`, `op` et `x` les VALEURS de ces trois ATTRIBUTS, respectivement. Soit *y* le nombre d'ENTITES INSTANCES directes ou indirectes de la CLASSE identifiée par `id` utilisées dans toutes les INSTRUCTIONS candidates à une mise en œuvre simultanée. L'OCCURRENCE DOMAIN est dit `observé` si et seulement si *y* satisfait la relation `op`(*y*, `x`).

`La CLASSE OCCURRENCE CROSS DOMAIN est utilisé` dans l'expression de contraintes sur les activités et les RESOURCES `pour décrire un ensemble de situations d'engagement conjoint de plusieurs RESOURCES et/ou plusieurs OPERATIONS. Elle possède un ATTRIBUT dont la VALEUR est un ensemble d'OCCURRENCE DOMAINS. L'OCCURRENCE CROSS DOMAIN est dit observé (par la situation réelle d'engagement) si et seulement si tous les OCCURRENCE DOMAINS sont dits observés.`

La CLASSE RESOURCE SHARING VIOLATION CONDITION est une INCONSISTENCY CONDITION qui permet d'exprimer une situation inacceptable ou irréalisable quant au partage d'une RESOURCE au profit de plusieurs OPERATIONS simultanées, ou quant au co-engagement simultané de la RESOURCE avec plusieurs autres RESOURCES[18]. La VALEUR de la PROPRIETE *classe-entité-contrainte* est une *particularisation* de RESOURCE, sujette à la limitation du niveau de partage.
L'OCCURRENCE CROSS DOMAIN est interprété comme une situation d'engagement inacceptable.
La METHODE *holding-condition* décrit quelle condition sur l'ETAT du PRODUCTION SYSTEM doit être vérifiée pour que la limitation soit prise en compte.
Pour exprimer, par exemple, qu'un PERFORMER ne peut jamais être mobilisé simultanément sur plusieurs LOCATIONS, on écrira :

- SINGLE PERFORMER
- ((LOCATION > *1*))
- *holding-condition* : `allways_true`

Pour exprimer qu'un PERFORMER ne peut pas exécuter à la fois une OPERATION `ot1` et une OPERATION `ot2` sur la même LOCATION, on écrira :

- SINGLE PERFORMER
- ((`ot1` > *0*) (`ot2` > *0*))

[18] Deux RESOURCES sont co-engagées (on dit aussi qu'elles collaborent) lorsqu'elles participent à la mise en œuvre d'une activité ou à chaque élément d'un ensemble d'activités. Si r1 est co-engagée avec r2 et r3, alors r2 est co-engagée avec r1, r3 est co-engagée avec r1, mais r2 n'est pas nécessairement co-engagée avec r3. Deux RESOURCES r et s ne sont pas co-engagées si r participe à la mise en œuvre d'un ensemble d'activités A et s à la mise en œuvre d'un ensemble d'activités B disjoint de A.

- *holding-condition* : getOperatedObject(ot1) == getOperatedObject(ot2)[19]

Noter qu'une CAPACITY-RELATED AVAILABILITY CONSTRAINT exprime que plusieurs OPERATIONS ne peuvent simultanément abaisser le *degré-de-disponibilité* d'une CAPACITATED RESOURCE de telle sorte que son *état-de-disponibilité* devienne non_disponible.

4.2.5 Rôles

Les RESOURCES d'une activité tiennent l'un des trois rôles suivants :

- une RESOURCE est objet de l'activité si c'est une ENTITE sur laquelle porte l'effet de l'OPERATION mise en jeu par l'activité ;
- une RESOURCE est support de l'activité si c'est une ENTITE dont la nécessité est liée à la nature de l'OPERATION en jeu ;
- une RESOURCE est sujet de l'activité si c'est une ENTITE qui réalise l'OPERATION en fonction de sa compétence et de sa puissance.

4.2.5.1 Les ressources objets

La CLASSE OPERATED OBJECT est une *particularisation* de DISCRETE-STATE REUSABLE RESOURCE qui permet de faire jouer à une ENTITE quelconque du CONTROLLED SYSTEM le rôle d'objet d'une INSTRUCTION.
A cette fin, cette CLASSE possède une PROPRIETE *entité-ressource*, dont la VALEUR est une référence à une ENTITE du CONTROLLED SYSTEM ou de l'OPERATING SYSTEM qui joue le rôle de l'objet de l'activité.
Pour faire jouer à une ENTITE le rôle d'objet d'une activité, il faut donc réaliser une INSTANCIATION d'OPERATED OBJECT, puis valuer la PROPRIETE *entité-ressource* de l'INSTANCE créée.
La METHODE *get-resource-class* appliquée cette INSTANCE renvoie la CLASSE de l'*entité-ressource*.
L'*entité-ressource* peut être une ENTITE élémentaire ou une ENTITE composée. Dans ce dernier cas, l'objet de l'INSTRUCTION est un ensemble d'ENTITES sur lesquelles se réalise tour à tour l'effet de l'OPERATION.
Puisqu'il s'agit d'une DISCRETE-STATE RESOURCE, la disponibilité d'une RESOURCE avec le rôle d'objet est fonction de son ETAT qualitatif. L'*état-de-disponibilité* d'un OPERATED OBJECT est disponible si et seulement si l''*état-de-disponibilité* de son *entité-ressource* est disponible et les AVAILABILITY CONSTRAINTS appliquées à l'*entité-ressource* sont respectées. Si l'*entité-ressource* n'est pas une INSTANCE de RESOURCE, son *état-de-disponibilité* est considéré disponible.
La METHODE *get-involved-entities* appliquée à une INSTANCE d'OPERATED OBJECT, et argumentée avec une CLASSE d'ENTITES C, renvoie une liste d'INSTANCES de la CLASSE C incluses dans l'*entité-ressource*. Le contenu de la METHODE dépend de la CLASSE de l'*entité-ressource*.

Bien que le rôle d'objet puisse être tenu par une quelconque ENTITE du CONTROLLED SYSTEM, on identifie trois sortes d'ENTITES appelées plus naturellement à le jouer. A chacune on fait correspondre une CLASSE définie comme une *particularisation* d'OPERATED OBJECT. Il s'agit de l'espace, des individus et enfin des RESOURCES de diverses natures lorsque c'est sur elles que porte l'effet d'une OPERATION. Lorsque l'objet de l'activité est une autre sorte d'ENTITE, il est l'*entité-ressource* d'une INSTANCE directe d'OPERATED OBJECT.

4.2.5.1.1 L'espace, objet d'activité

Une OPERATED LOCATION est un OPERATED OBJECT dont la VALEUR de la PROPRIETE *entité-ressource* est une référence à une LOCATION présentant une unité d'activité.
La METHODE *get-involved-entities* appliquée à une OPERATED LOCATION l avec l'argument C renvoie soit la liste {l} si l est INSTANCE de C, soit une liste de LOCATIONS de CLASSE C qui sont ELEMENTS de l à une profondeur quelconque.

4.2.5.1.2 Les ensembles d'individus, objets d'activité

Une OPERATED POPULATION est un OPERATED OBJECT dont la VALEUR de la PROPRIETE *entité-ressource* est une référence à une POPULATION.
La METHODE *get-involved-entities* appliquée à une OPERATED POPULATION p avec l'argument C se comporte différemment selon que C est ou non une *particularisation* de POPULATION. Dans le premier cas, elle renvoie soit la liste {p} si p est INSTANCE de C, soit une liste de POPULATIONS de CLASSE C qui sont ELEMENTS de p à une profondeur quelconque. Si C n'est pas une *particularisation* de POPULATION, *get-involved-entities* renvoie la liste des INSTANCES

[19] getOperatedObject(ot) est censé renvoyer l'objet de l'activité ; c'est ce que peut renvoyer l'INVOCATION de *current-operated-object* (*cf.* § 4.4) sur l'OPERATED OBJECT SPECIFICATION de l'INSTRUCTION qui met en œuvre l'OPERATION ot.

de C qui sont éléments des *ensembles-représentatifs* des POPULATIONS SIMPLES elles-mêmes ELEMENTS de `p` à une profondeur quelconque.

4.2.5.1.3 Supports et sujets dans le rôle d'objets d'activité

Une OPERATED RESOURCE est un OPERATED OBJECT dont la VALEUR de la PROPRIETE *entité-ressource* est une référence à une OPERATION RESOURCE ou un PERFORMER, qui joue dans ce cas le rôle de l'objet de l'activité.
La METHODE *get-involved-entities* appliquée à une OPERATED RESOURCE `r` avec l'argument `C` renvoie soit la liste `{r}` si `r` est INSTANCE de `C`, soit une liste d'ENTITES de CLASSE `C` qui sont ELEMENTS de `r` à une profondeur quelconque.

4.2.5.2 Les ressources supports

Une OPERATION RESOURCE est une RESOURCE propre d'une OPERATION. En d'autres termes, sa réquisition est spécifiée dans la définition, non pas d'une INSTRUCTION (via une PRIMITIVE ACTIVITY SPECIFICATION), mais d'une OPERATION.

4.2.5.3 Les ressources sujets

Le TERME PERFORMER désigne toute RESOURCE caractérisée par une *puissance*. Cette RESOURCE peut être un SINGLE PERFORMER, un HOMOGENEOUS AGGREGATED PERFORMER ou un HETEROGENEOUS AGGREGATED PERFORMER. Lorsqu'un PERFORMER tient le rôle du sujet d'une INSTRUCTION, la VALEUR de *puissance* multiplie la *vitesse* de l'OPERATION qui définit la nature de l'INSTRUCTION.

Un SINGLE PERFORMER est une DISCRETE-STATE REUSABLE RESOURCE dont les ELEMENTS ou COMPOSANTS, s'il y en a, ne sont pas des PERFORMERS. Elle est caractérisée par une *puissance*.

Un SINGLE WORKER est une *particularisation* de SINGLE PERFORMER. Elle représente un travailleur humain.

Un HOMOGENEOUS AGGREGATED PERFORMER est une HOMOGENEOUS AGGREGATED RESOURCE dont les ELEMENTS sont soit des INSTANCES d'une seule CLASSE[20] de SINGLE PERFORMER, soit des INSTANCES de plusieurs *particularisations* d'HOMOGENEOUS AGGREGATED PERFORMER à condition qu'il existe une CLASSE dont héritent toutes ces *particularisations*. Elle est caractérisée par cette unique CLASSE (dans les deux cas) et un *effectif*.
La *puissance* est, par défaut, la somme des *puissances* des ELEMENTS.

Une HOMOGENEOUS LABOR TEAM est une *particularisation* de HOMOGENEOUS AGGREGATED PERFORMER dont les ELEMENTS sont des SINGLE WORKERS.

Un HETEROGENEOUS AGGREGATED PERFORMER est une HETEROGENEOUS AGGREGATED RESOURCE dotée de COMPOSANTS qui sont des PERFORMERS.
La *puissance* doit être explicitée : elle n'est pas, par défaut, la somme des *puissances* des COMPOSANTS.

Une HETEROGENEOUS LABOR TEAM est une *particularisation* de HETEROGENEOUS AGGREGATED PERFORMER dont les COMPOSANTS sont des SINGLE WORKERS, des HOMOGENEOUS LABOR TEAMS ou des HETEROGENEOUS LABOR TEAMS.

4.2.6 Spécification de ressource propre d'opération

La CLASSE OPERATION RESOURCE SPECIFICATION permet de spécifier une ou pusieurs RESOURCE dont la disponibilité est requise par une OPERATION. Elle est définie par :

- un ATTRIBUT dont la VALEUR est une SPECIFICATION D'ENSEMBLE D'ENTITES dont la CLASSE d'ENTITES est une *particularisation* de RESOURCES référencée dans la RESOURCE POOL DISJUNCTION de l'OPERATING SYSTEM,
- une METHODE dont la VALEUR est une référence à une FONCTION de sélection, laquelle peut être *any*, *all*, *max*, *disj* ou *sets* ; les FONCTIONS *any* et *sets* possèdent un argument à valeur entière `n`,
- un ATTRIBUT *mode*. La VALEUR de *mode* est `proportional` si l'ensemble de RESOURCES spécifié par la SPECIFICATION D'ENSEMBLE D'ENTITES est requis pour chaque unité de *puissance* du PERFORMER alloué à la réalisation de l'OPERATION, et `non-proportional` si l'ensemble de RESOURCES est requis tel quel, quelle que soit la *puissance* du PERFORMER,
- une PROPRIETE *continuation*,

[20] On rappelle que, par définition, les ELEMENTS sont des INSTANCES de la même CLASSE.

- une PROPRIETE *consommation-unitaire-par-pas* valuée, seulement si la RESOURCE spécifiée est une CAPACITATED CONSUMABLE RESOURCE, avec la quantité (multidimensionnelle) de la *capacité* qui est consommée par l'OPERATION par unité de *puissance* du PERFORMER et par *pas* de l'OPERATION.

Le mécanisme de DEVELOPPEMENT de l'OPERATION RESOURCE SPECIFICATION invoque le mécanisme d'EXPANSION de la SPECIFICATION D'ENSEMBLE D'ENTITES puis la FONCTION de sélection sur l'ensemble issu de l'EXPANSION. Le résultat est une liste d'ensembles alternatifs de RESOURCES pour l'affectation à la mise en œuvre de l'OPERATION (voir l'article OPERATED OBJECT SPECIFICATION).

Le DEVELOPPEMENT réalise ces deux invocations d'une manière et autant de fois que déduit de la VALEUR du *mode* et de la *puissance* du PERFORMER[21].

Dans le *mode* `proportional`, si la FONCTION de sélection est *any*(`n`) ou *sets*(`n`), le DEVELOPPEMENT invoque la FONCTION de sélection avec un argument dont la valeur est `n` * *puissance*[22].

Toujours dans le *mode* `proportional`, si la FONCTION de sélection est *disj*, le DEVELOPPEMENT réalise ces deux invocations autant de fois que la VALEUR de la *puissance* du PERFORMER, et il opère ensuite le produit cartésien des listes résultant de ces invocations.

Les FONCTIONS de sélection *all* et *max* sont incompatibles avec le *mode* `proportional`.

Dans le *mode* `non_proportional`, la FONCTION de sélection, quelle qu'elle soit, n'est invoquée qu'une fois, avec ses éventuels arguments inchangés.

La PROPRIETE *continuation* spécifie, dans le cas où une OPERATION est reprise après avoir été interrompue pendant un délai, que les RESOURCES requises à la reprise sont ou non les mêmes que celles qui étaient en usage en l'INSTANT de l'interruption. Seulement si *continuation* est `new`, on opère un nouveau DEVELOPPEMENT de cette OPERATION RESOURCE SPECIFICATION de l'OPERATION ; selon les autres VALEURS de *continuation*, la spécification requiert strictement les mêmes RESOURCES (`exactly-same`), ou bien, parmi celles qui étaient en usage à l'interruption, seulement celles qui figurent encore dans le résultat de l'EXPANSION (`same-previous`).

4.3 Stratégie

Une STRATEGY est une ENTITE qui représente l'ensemble des dispositions prises par un MANAGER pour piloter un PRODUCTION SYSTEM, compte tenu d'un objectif et d'une connaissance (partielle) de son propre ENVIRONNEMENT. Ses COMPOSANTS sont :

- un ensemble d'au moins un ACTIVITIES-RESOURCES BLOCK,
- une REACTIVE TRAJECTORY,
- un ensemble de références à des PREFERENCE RULES.

Dans l'ensemble des ACTIVITIES-RESOURCES BLOCKS, tous les SIMPLE RESOURCE POOLS, qui en sont des COMPOSANTS, doivent être différents. A cette fin, ils sont pris, chacun une seule fois, parmi les éléments de la RESOURCE POOL DISJUNCTION qui est un des COMPOSANTS de l'OPERATING SYSTEM.

Deux PROPRIETES particulières de la STRATEGY sont :

- le *rythme-de-lecture*, qui est le rythme d'OCCURRENCE des EVENEMENTS provoquant l'exploitation des ACTIVITIES-RESOURCES BLOCKS par le MANAGER (mécanisme UPDATING SITUATION) et par l'OPERATING SYSTEM (MAKING INSTRUCTION LIST),
- le *rythme-de-révision*, qui est le rythme d'OCCURRENCE des EVENEMENTS provoquant l'exploitation par le MANAGER de la REACTIVE TRAJECTORY (mécanisme CHECKING REACTIVE TRAJECTORY).

Les VALEURS de *rythme-de-lecture* et *rythme-de-révision* sont exploitées par la PROCEDURE référencée dans la METHODE *procédure-autogénération* des EVENEMENTS en question. Noter que, pour les deux types d'EVENEMENTS, des OCCURRENCES complémentaires peuvent être invoquées par d'autres mécanismes que l'AUTOGENERATION.

Un ACTIVITIES-RESOURCES BLOCK est une ENTITE qui décrit un ensemble d'activités de contrôle exploitant un sous-ensemble des RESOURCES de l'OPERATING SYSTEM. L'exploitation des RESOURCES est consistante, c'est-à-dire qu'elle respecte des contraintes sur leur exploitation simultanée, et exclusive, c'est-à-dire que les RESOURCES ne sont pas partagées avec les activités d'un autre ACTIVITIES-RESOURCES BLOCK. Ses COMPOSANTS sont :

---

21 On verra en section 5 que cela requiert que les ressources sujets soient déterminées, pour un jeu d'activités donné, avant les ressources supports.

22 Soit par exemple une OPERATION requérant une RESOURCE selon la spécification suivante :

- SPECIFICATION D'ENSEMBLE D'ENTITES : { CLASSE D'ENTITES : A, PREDICAT de sélection : *true*, *mode* : `return_instances`, *effectif* : *all* },
- FONCTION de sélection : *sets*(*1*),
- *mode* : `proportionnal`.

Soit encore A1 (avec les INSTANCEs a11, a12) et A2 (avec les INSTANCEs a21, a22) deux *particularisations* terminales de A. Considérons enfin que la PERFORMER SPECIFICATION implique que la *puissance* allouée à l'OPERATION soit *2*.

Le DEVELOPPEMENT réalise l'invocation de l'EXPANSION de la SPECIFICATION D'ENSEMBLES D'ENTITES et de la FONCTION de sélection, avec l'argument 2 (puisque *1* INSTANCE de A est requise pour chacune des *2* unités de *puissance*). L'invocation renvoie la liste ((a11 a12) (a11 a21) (a21 a22)).

- un NOMINAL PLAN,
- un SIMPLE RESOURCE POOL, qui est l'ensemble des INSTANCES de RESOURCES exploitées, dénommé `l'inventory`,
- `un ensemble de` ACTIVITIES RESOURCES INCONSISTENT COMMITMENTS `.`

Un NOMINAL PLAN spécifie un ensemble d'activités de contrôle et pose des contraintes sur leur coordination. C'est une ACTIVITY SPECIFICATION racine d'une hiérarchie d'autres ACTIVITY SPECIFICATIONS. Autrement dit, c'est une ACTIVITY SPECIFICATION qui n'est pas elle-même partie prenante (en tant qu'ELEMENT) dans la définition d'une autre ACTIVITY SPECIFICATION.

Le mécanisme UPDATING SITUATION met à jour la PROPRIETE *situation* de chaque ACTIVITY SPECIFICATION des NOMINAL PLANS associés aux ACTIVITIES-RESOURCES BLOCKS, PROPRIETE qui détermine s'il n'y a pas encore lieu, s'il y a maintenant lieu ou s'il n'y a plus lieu de la considérer pour le contrôle du CONTROLLED SYSTEM.
Cette mise à jour repose sur deux éléments : d'une part l'examen des conditions d'ouverture et de fermeture déclarées dans les ACTIVITY SPECIFICATIONS, et d'autre part le mécanisme de PROPAGATION, lequel fixe et met en œuvre des contraintes sur la *situation* d'une ACTIVITY SPECIFICATION en fonction de la *situation* de ses ELEMENTS ou de celle de l'ACTIVITY SPECIFICATION dont elle est ELEMENT.

Le mécanisme MAKING INSTRUCTION LIST détermine les activités de contrôle à mettre en œuvre immédiatement sur le CONTROLLED SYSTEM.
On génère d'abord un ensemble de jeux de PRIMITIVE ACTIVITY SPECIFICATIONS alternatifs, par une INVOCATION de la METHODE *expand-to-disjunction* sur l'ACTIVITY SPECIFICATION qui définit le NOMINAL PLAN, en ne retenant que les ACTIVITY SPECIFICATIONS qui ont la VALEUR `open` pour leur PROPRIETE *situation*. Ensuite, seuls les jeux sur lesquels le mécanisme PRE-ALLOCATING RESOURCES parvient à son terme sont conservés et appelés PACKS OF EXECUTABLE INSTRUCTIONS. Un seul est choisi, par le mécanisme SELECTING INSTRUCTION LIST, qui met en œuvre les PREFERENCE RULES[23]. On réunit ensuite dans un seul PACK les PACKS issus des différents ACTIVITIES-RESOURCES BLOCK, puisque par construction une INSTRUCTION dans l'un ne peut pas être privée de ses RESOURCES par la considération d'un autre (à cause de l'exclusivité des RESOURCES). Enfin est lancé le mécanisme ACTING INSTRUCTION LIST.
Le mécanisme MAKING INSTRUCTION LIST est mis en œuvre dans trois sortes de situations : (i) à chaque INSTANT déterminé par le *rythme-de-lecture* de la STRATEGY, (ii) quand la remise en œuvre du PACK OF EXECUTABLE INSTRUCTIONS en cours ne peut se poursuivre, du fait de l'IMMOBILISATION d'une RESOURCE, et (iii) quand un autre PACK OF EXECUTABLE INSTRUCTIONS pourrait être établi et préféré, du fait de la MOBILISATION ou la LIBERATION d'une RESOURCE (notamment chaque fois qu'une OPERATION se termine). L'AGENDA du SYSTEME peut contenir plusieurs INVOCATIONS de ce mécanisme à des INSTANTS différents.

Le mécanisme PRE-ALLOCATING RESOURCES réalise la DETERMINATION conjointe d'un ensemble de PRIMITIVE ACTIVITY SPECIFICATIONS en respectant :
- les ACTIVITY INCONSISTENCY CONDITIONS attachées aux différentes PRIMITIVE ACTIVITY SPECIFICATIONS,
- les RESOURCE SHARING VIOLATION CONDITIONS attachées aux différentes RESOURCES en jeu,
- les ACTIVITIES RESOURCES INCONSISTENT COMMITMENTS attachés à l'ACTIVITIES-RESOURCES BLOCK duquel est issu l'ensemble de PRIMITIVE ACTIVITY SPECIFICATIONS.

Si une de ces contraintes n'est pas respectée, le jeu initial est décomposé en un ensemble de jeux, dans chacun desquels on a enlevé une PRIMITIVE ACTIVITY SPECIFICATION différente. On tente à nouveau une DETERMINATION sur chacun des jeux, et ce mécanisme se poursuit jusqu'à satisfaction de toutes les contraintes dans au moins un jeu. On a alors un ensemble d'un ou plusieurs jeux d'INSTRUCTIONS. Dans chaque jeu, ce nombre est maximal au sens où la réintroduction dans un jeu d'une INSTRUCTION non retenue est soit impossible du fait de l'indisponibilité d'une RESOURCE, soit entraînerait le non respect d'au moins une des contraintes.
Il est invoqué dans le mécanisme MAKING INSTRUCTION LIST.

Un ACTIVITIES RESOURCES INCONSISTENT COMMITMENT est une ENTITE qui décrit une situation impossible ou inacceptable quant à l'engagement simultané de plusieurs RESOURCES, ou bien la mise en œuvre simultanée de plusieurs OPERATIONS, dans un jeu d'INSTRUCTIONS généré à partir d'un ACTIVITIES-RESOURCES BLOCK. Ses ELEMENTS sont :
- un ensemble d'INCONSISTENCY CONDITIONS, c'est-à-dire des RESOURCE SHARING VIOLATION CONDITIONS et/ou de ACTIVITY INCONSISTENCY CONDITIONS.

Elle possède en propre une METHODE *holding-condition* (de type PREDICAT), qui décrit quelle condition sur l'ETAT du PRODUCTION SYSTEM doit être vérifiée pour que les conditions ci-dessus soient valides pour détecter une situation impossible ou inacceptable.

[23] On parle de pré-allocation parce qu'à ce stade on peut envisager d'allouer une RESOURCE à plusieurs PRIMITIVE ACTIVITY SPECIFICATIONS, ou bien parce qu'on peut envisager de mettre en œuvre une PRIMITIVE ACTIVITY SPECIFICATION avec plusieurs alternatives de RESOURCES. L'allocation effective des RESOURCES a lieu lorsqu'a été choisi le PACK OF EXECUTABLE INSTRUCTIONS à mettre en œuvre.

L'ACTIVITIES RESOURCES INCONSISTENT COMMITMENT est dit `vérifié` par le jeu d'INSTRUCTIONS si et seulement si la *holding-condition* renvoie `true` et si toutes les RESOURCE SHARING VIOLATION CONDITIONS portant sur chaque RESOURCE engagée dans le jeu sont `observées` et si toutes les ACTIVITY INCONSISTENCY CONDITIONS portant sur chaque INSTRUCTION du jeu sont `observées`.
Pour exprimer, par exemple, qu'on n'emploie pas une OPERATION RESOURCE de CLASSE `R`, en même temps qu'un SINGLE WORKER de CLASSE `W` utilise plusieurs OPERATION RESOURCES de CLASSE `T` dans une ou plusieurs activités, on écrira :

{ (RSVC, AIC), `allways_true` }

où RSVC est une RESOURCE SHARING VIOLATION CONDITION ainsi écrite :

- `W`
- ((`T` > 1))
- `allways_true`

et AIC est une ACTIVITY INCONSISTENCY CONDITION ainsi écrite :

- `activity specification`
- ((`R` > *0*))
- `allways_true`

Les RESOURCES `W`, `T`, `R` peuvent être ou ne pas être engagées dans la même activité.

Une REACTIVE TRAJECTORY est le COMPOSANT de la STRATEGY qui détermine les changements de celle-ci qui paraissent opportuns au MANAGER dans certaines circonstances prédéfinies. Ses ELEMENTS sont :

- des CONDITIONAL ADJUSTMENTS.

Un CONDITIONAL ADJUSTMENT spécifie qu'à l'INSTANT où il est détecté qu'un repère (TRIGGERING LANDMARK) a été atteint, il faut examiner si la situation courante satisfait un STATE PREDICATE et que, dans l'affirmative, il convient de mettre à jour la STRATEGY selon les directives d'une STRATEGY ADAPTATION. Cette *particularisation* d'ENTITE possède à cet effet, et respectivement, les trois METHODES suivantes :

- *triggering-landmark*, dont la VALEUR est un CALENDAR PREDICATE ou un STATE PREDICATE. C'est un repère temporel prédéfini et choisi pour l'intérêt qu'il comporte à provoquer un éventuel changement dans la STRATEGY.
- *state-predicate*, dont la VALEUR est un STATE PREDICATE.
- *strategy-adaptation*, dont la VALEUR est une STRATEGY ADAPTATION. C'est une PROCEDURE qui réalise un ensemble de modifications, chacune sur n'importe quel COMPOSANT ou sur n'importe quelle PROPRIETE de la STRATEGY.

Le TERME CALENDAR PREDICATE désigne un PREDICAT sur la VALEUR de l'INSTANT courant, dont la référence est la VALEUR de la METHODE *triggering-landmark* d'un CONDITIONAL ADJUSTMENT.

Le TERME STATE PREDICATE désigne un PREDICAT sur l'ETAT du PRODUCTION SYSTEM ou sur la VALEUR d'un INDICATEUR. Le PREDICAT est satisfait à certains INSTANTS et non satisfait à d'autres, et peut à ce titre représenter un repère temporel. Sa référence peut être la VALEUR de la METHODE *triggering-landmark* ou de la METHODE *state-predicate* d'un CONDITIONAL ADJUSTMENT.

Le mécanisme CHECKING REACTIVE TRAJECTORY est mis en œuvre à chaque INSTANT déterminé par le *rythme-de-révision* de la STRATEGY. Il réalise, pour chaque élément de la REACTIVE TRAJECTORY, l'INVOCATION INTENTIONNELLE des PREDICATS *triggering-landmark* et *state-predicate*, et, si les deux ont renvoyé *true*, l'INVOCATION INTENTIONNELLE de la PROCEDURE *strategy-adaptation*.

Un PACK OF EXECUTABLE INSTRUCTIONS est une ENTITE qui représente un ENSEMBLE d'INSTRUCTIONS spécifiées à l'intérieur d'un même ACTIVITIES-RESOURCES BLOCK. Par construction (dans le mécanisme PRE-ALLOCATING RESOURCES), cet ENSEMBLE satisfait l'ensemble des ACTIVITIES RESOURCES INCONSISTENT COMMITMENTS du BLOCK, l'ensemble des RESOURCE SHARING VIOLATION CONDITIONS des RESOURCES en jeu, et l'ensemble des ACTIVITY INCONSISTENCY CONDITIONS sur les INSTRUCTIONS. De plus, les OPERATIONS incluses dans les INSTRUCTIONS doivent avoir leur *condition-de-faisabilité* satisfaite.
Une PROPRIETE d'un PACK OF EXECUTABLE INSTRUCTIONS est la référence à ACTIVITIES-RESOURCES BLOCK dans lequel il a été généré.
Un PACK OF EXECUTABLE INSTRUCTIONS (désigné aussi par le mot paquet) est maximal au sens où l'adjonction de toute autre INSTRUCTION qui était candidate à l'exécution se heurte à l'indisponibilité d'une RESOURCE ou viole une contrainte.
A un INSTANT donné, on est en mesure d'établir un seul, ou aucun ou plusieurs PACKS OF EXECUTABLE INSTRUCTIONS. La remarque sur la maximalité implique que les paquets sont mutuellement exclusifs et constituent donc des alternatives d'action. Le choix entre ces alternatives est opéré par le mécanisme SELECTING INSTRUCTION LIST sur la

base des PREFERENCE RULES. Dès lors, toutes les exécutions des INSTRUCTIONS du paquet choisi peuvent démarrer et/ou se poursuivre.

Une PREFERENCE RULE (désignée aussi par l'expression règle de préférence) est une FONCTION qui exprime une relation de supériorité entre deux PACKS OF EXECUTABLE INSTRUCTIONS. N'importe quelle PROPRIETE des paquets ou de leurs ELEMENTS peut être le critère de la relation. La FONCTION renvoie le paquet qui est supérieur à l'autre.
Une STRATEGY ne doit pas faire référence à un ensemble de PREFERENCE RULES qui pourraient être contradictoires. Les conflits éventuels doivent être résolus par une rédaction appropriée des éléments de l'ensemble.

Le mécanisme SELECTING INSTRUCTION LIST extrait un paquet d'un ensemble de PACKS OF EXECUTABLE INSTRUCTIONS, sur la base des PREFERENCE RULES attachées à la STRATEGY du MANAGER.
Si la STRATEGY ne fait référence qu'à une PREFERENCE RULE, cette règle est appliquée aux deux premiers paquets puis, itérativement tant qu'il reste des paquets, au paquet renvoyé par l'application précédente et au paquet suivant. Si deux paquets ne peuvent être départagés, le premier est arbitrairement renvoyé. Le résultat de la dernière application est le paquet choisi.
Si la STRATEGY fait référence à plusieurs PREFERENCE RULES, celles-ci doivent être référencées dans la STRATEGY par ordre décroissant de priorité. On applique la même procédure que ci-dessus, sauf qu'on tente de départager deux paquets par la première règle, et, seulement si cette règle a échoué, par la suivante, et ainsi de suite jusqu'à la dernière règle. Si la dernière règle a échoué le premier des deux paquets est arbitrairement renvoyé.
Remarquer qu'en général il suffit de trouver le meilleur paquet sans établir un ordre sur les autres paquets.

4.4 Activités et instructions

Le mot « activité » sera employé dans cette section au sens d'INSTANCE de la CLASSE ACTIVITY SPECIFICATION lorsque le contexte permet de lever une éventuelle ambiguïté.

Une ACTIVITY SPECIFICATION spécifie une activité de contrôle, simple ou complexe. C'est une ENTITE qui est :

- soit une « activité agrégée » (dont les ELEMENTS sont aussi appelés les « activités composantes »), qui exprime une contrainte conjonctive, disjonctive ou temporelle (précédence, synchronisation, parallélisme) sur les composantes ;
- soit l'expression d'une contrainte sur une unique ACTIVITY SPECIFICATION, pour en expliciter le caractère itératif ou optionnel ;
- soit enfin une « activité primitive », sans composantes, et qui spécifie l'OPERATION mise en œuvre sur le CONTROLLED SYSTEM.

Toutes les ACTIVITY SECIFICATIONS sont définies par les huit PROPRIETES communes suivantes : *opérateur-sur-sous-activités, début-au-plus-tôt, début-au-plus-tard, fin-au-plus-tôt, fin-au-plus-tard*, *date-ouverture, date-fermeture, situation*, et par les quatre METHODES *expand-to-disjunction* et *fenêtre-début* (FONCTIONS), *condition-ouverture* et *condition-fermeture* (PREDICATS).
Seulement pour certaines *particularisations* d'activités agrégées, des PROPRIETES additionnelles spécifient des délais (temporels) entre les activités composantes.
Les *début-au-plus-tôt, début-au-plus-tard, fin-au-plus-tôt* et *fin-au-plus-tard* d'une activité A sont la combinaison, spécifique de chaque *particularisation* d'ACTIVITY SPECIFICATION (voir annexe 8) des spécifications propres de A avec les VALEURS de ces PROPRIETES pour les activités composantes de A et les activités dont A est une composante. Le cas échéant pour certaines *particularisation*, les VALEURS sont mises à jour lors d'ouvertures ou de fermetures d'activités, en tenant compte des VALEURS des PROPRIETES spécifiant des délais, mentionnées plus haut.
La FONCTION *fenêtre-début* renvoie les *début-au-plus-tôt* et *début-au-plus-tard* ainsi calculés.
Chaque VALEUR de *opérateur-sur-sous-activités* donne lieu à une *particularisation* de ACTIVITY SPECIFICATION. Le mécanisme d'INSTANCIATION n'est pas invoqué sur cette CLASSE directement, mais sur ses *particularisations*.
La FONCTION *expand-to-disjunction* renvoie, à partir d'une INSTANCE de la présente CLASSE (plus exactement d'une de ses *particularisations*) un ensemble de jeux alternatifs de PRIMITIVE ACTIVITY SPECIFICATIONS. Elle est invoquée dans le mécanisme MAKING INSTRUCTION LIST, et son contenu est spécifique de chaque *particularisation* d'ACTIVITY SPECIFICATION (cf. section 5.5.1).
La PROPRIETE *situation* peut prendre une des VALEURS `sleeping`, `waiting`, `open`, `closed`, `cancelled`. Selon les *opérateur-sur-sous-activités* les *situations* se propagent différemment entre l'ACTIVITY SPECIFICATION (activité) et ses ELEMENTS d'une part et entre les ELEMENTS d'autre part (mécanisme de PROPAGATION).
La VALEUR `sleeping` est celle donnée au moment de l'INSTANCIATION de l'activité, signifiant que les conditions d'ouverture de l'activité n'ont pas encore à être examinées.
Les conditions d'ouverture et de fermeture d'une activité portent d'une part sur la position de l'INSTANT courant par rapport aux dates de début (*début-au-plus-tôt, début-au-plus-tard*) et de fin (*fin-au-plus-tôt, fin-au-plus-tard*) de

l'activité et d'autre part sur des conditions spécifiques (les PREDICATS *condition-ouverture* et *condition-fermeture*) portant sur certains ETATS ou INDICATEURS. Pour les PRIMITIVE ACTIVITY SPECIFICATIONS, la fermeture dépend en outre du *degré-de-progression* de l'OPERATION encapsulée dans l'activité.
La *situation* passe à la VALEUR `waiting` dès que les conditions d'ouverture de l'activité doivent être examinées. Par exemple, dès qu'une activité se termine, on peut examiner les conditions d'ouverture de celle qui la suit dans une séquence.
La *situation* d'une activité passe à la VALEUR `open` lorsque les conditions d'ouverture de l'activité sont satisfaites. A cet INSTANT, la PROPRIETE *date-ouverture* est valuée avec l'INSTANT courant. Seules les OPERATIONS entrant dans la définition d'une activité `open` sont susceptibles d'être exécutées.
La *situation* passe de la VALEUR `open` à la VALEUR `closed` lorsque les conditions de fermeture de l'activité sont satisfaites. A cet INSTANT, la PROPRIETE *date-fermeture* est valuée avec l'INSTANT courant. S'il s'agit d'une PRIMITIVE ACTIVITY SPECIFICATION, la fermeture est provoquée lorsque l'OPERATION en jeu est `terminated`. Lorsque la fermeture d'une PRIMITIVE ACTIVITY SPECIFICATION est demandée par propagation d'un changement de *situation* par ailleurs, elle est acceptée si le *degré-de-progression* de l'OPERATION est supérieur au *seuil-fermeture*. Sinon, la fermeture est refusée.
La *situation* d'une activité passe à la VALEUR `cancelled` lorsqu'il n'y a plus lieu de gérer la *situation* de l'activité (c'est-à-dire d'opérer un quelconque changement de *situation*). Typiquement, cela se produit lors d'un choix entre deux ou plusieurs activités alternatives : la *situation* des activités non choisies passe à `cancelled`.
Seulement pour certaines *particularisations* d'activités agrégées, la METHODE prédéfinie additionnelle *modify-in-open-situation* permet de modifier une activité ouverte. On peut, typiquement, ajouter ou enlever des composantes à l'activité.

Une PRIMITIVE ACTIVITY SPECIFICATION (ou spécification d'activité « primitive ») exprime les propriétés souhaitées de l'objet (OPERATED OBJECT), du sujet (PERFORMER) et de la nature (OPERATION) d'une activité à réaliser. C'est une *particularisation* D'ACTIVITY SPECIFICATION qui ne possède pas d'ELEMENTS de CLASSE ACTIVITY SPECIFICATION (la PROPRIETE *opérateur-sur-sous-activités* est sans VALEUR). Elle possède trois ATTRIBUTS, dont les VALEURS sont des INSTANCES de:

- OPERATED OBJECT SPECIFICATION,
- OPERATION SPECIFICATION,
- PERFORMER SPECIFICATION.

On dote normalement une PRIMITIVE ACTIVITY SPECIFICATION soit de VALEURS significatives pour les PROPRIETES *début-au-plus-tôt, début-au-plus-tard,* soit d'un PREDICAT *condition-ouverture*, soit des deux. Il en est de même pour les éléments de la condition de fermeture. Par défaut, la VALEUR de *début-au-plus-tôt* est *0*, celle de *début-au-plus-tard* est infiniment grande. Les fenêtres d'ouverture et de fermeture peuvent aussi être inférées à partir des activités dont celle-ci est composante (voir annexe 8).
L'INVOCATION de la METHODE *fenêtre-début* renvoie le couple { *début-au-plus-tôt, début-au-plus-tard* }.
Le mécanisme de DETERMINATION est celui par lequel on exploite la spécification en termes de propriétés, pour allouer à l'activité, au moment souhaité de sa mise en œuvre, des RESOURCES ayant ces propriétés. Il réalise les DEVELOPPEMENTS de l'OPERATED OBJECT SPECIFICATION, de la PERFORMER SPECIFICATION et de l'OPERATION SPECIFICATION (lequel provoque le DEVELOPPEMENT de ses OPERATION RESOURCE SPECIFICATIONS), et transforme la PRIMITIVE ACTIVITY SPECIFICATION en INSTRUCTION.
Toute activité primitive doit être dotée d'une OPERATION SPECIFICATION. Toute activité primitive telle que l'OPERATION en jeu est une UNIT-GRANULATED-OPERATION ou une QUANTITY-GRANULATED-OPERATION doit être dotée d'une OPERATED OBJECT SPECIFICATION. Lorsque l'OPERATED OBJECT SPECIFICATION est absente (seulement possible pour des TIME-GRANULATED-OPERATION), l'activité est supposée avoir un effet qui ne porte pas sur le système piloté. Lorsque la PERFORMER SPECIFICATION est absente, l'effet de l'activité est supposé pouvoir être réalisée sans l'intervention d'une telle RESOURCE.
La DETERMINATION est réalisée dans le mécanisme PRE-ALLOCATING RESOURCES. Ce dernier réalise en effet des DETERMINATIONS compatibles pour un ensemble de PRIMITIVE ACTIVITY SPECIFICATIONS. Ces déterminations sont faites dans l'ordre des activités dans l'ensemble. Cet ordre peut être contrôlé par le modélisateur (*cf.* § 5.5.1.3), par exemple en faisant jouer les priorités d'allocation des activités primitives de l'ensemble, valeurs de la PROPRIETE *.priorité-d-allocation*.
Noter que cette DETERMINATION peut établir que l'activité doit porter sur un ensemble d'objets, ou bien mobiliser un ensemble d'OPERATION RESOURCES, ou bien être réalisée par un ensemble de PERFORMERS. Les trois ensembles de RESOURCES sont collectivement et mutuellement engagés. En d'autres termes, la *capacité* totale des PERFORMERS est utilisée pour opérer sur une ENTITE unique virtuelle dont les ELEMENTS seraient les OPERATED OBJECTS, et les RESOURCES PROPRES ne sont pas supposées réparties à l'usage de différents PERFORMERS ou au bénéfice de différents OPERATED OBJECTS.

Noter encore que les ENTITES issues du DEVELOPPEMENT de l'OPERATED OBJECT SPECIFICATION sont successivement en position d'objet de l'activité, alors que les ENTITES issues des DEVELOPPEMENTS de la PERFORMER SPECIFICATION et des OPERATION RESOURCE SPECIFICATIONS de l'OPERATION SPECIFICATION sont concomitamment mis au service de l'activité.
Une PRIMITIVE ACTIVITY SPECIFICATION est caractérisée par un ensemble d'ACTIVITY INCONSISTENCY CONDITIONS, qui rendent inconsistante soit la spécification elle-même *a priori* (c'est-à-dire toute INSTRUCTION qui pourrait résulter de la DETERMINATION), soit une INSTRUCTION particulière résultant *a posteriori* de la DETERMINATION.

La mise en œuvre d'une PRIMITIVE ACTIVITY SPECIFICATION échoue[24] si sa *condition-ouverture* n'a pas pu être vérifiée entre son *début-au-plus-tôt* et son *début-au-plus-tard*, ou si sa *condition-fermeture* n'a pas pu être vérifiée entre sa *fin-au-plus-tôt* et sa *fin-au-plus-tard*.

Le TERME INSTRUCTION désigne une PRIMITIVE ACTIVITY SPECIFICATION dont les COMPOSANTS sont complètement déterminés : une OPERATION à réaliser par un ensemble de PERFORMERS sur un ensemble d'OPERATED OBJECTS en utilisant des OPERATION RESOURCES identifiées.

La CLASSE OPERATED OBJECT SPECIFICATION permet de spécifier l'ensemble des objets sur lequel porte la PRIMITIVE ACTIVITY SPECIFICATION. Elle est définie par :

- un ATTRIBUT dont la VALEUR est une SPECIFICATION D'ENSEMBLE D'ENTITES (elle-même définie, pour mémoire, par une *particularisation* d'ENTITE, un PREDICAT de sélection, un *mode* et un *effectif* appropriés),
- une METHODE dont la VALEUR est une référence à une FONCTION de sélection, laquelle peut être *any*, *all*, *max*, *disj* ou *sets* ; les FONCTIONS *any* et *sets* possèdent un argument à valeur entière `n`.
- une PROPRIETE *continuation*,
- un ATTRIBUT *liste-opérée* dont la VALEUR est la liste des OPERATED OBJECTS sur lesquels est mise en œuvre l'INSTRUCTION et résultant du mécanisme MAKING INSTRUCTION LIST (*cf.* § 5.5.2.1.3).
- une METHODE *state-availability-predicate* dont le corps encapsule les STATE AVAILABILITY CONSTRAINTS sur les OPERATED OBJECTS qui seront générés par le mécanisme d'EXPANSION. Ce corps, spécifié au niveau de L'OPERATED OBJECT SPECIFICATION sera en effet attaché à la METHODE *state-availability-predicate* des OPERATED OBJECTS. On n'a pas à donner de VALEUR à cette METHODE si les *entités-ressources* sont des INSTANCES de RESOURCE puisqu'on peut alors attacher à ces dernières le PREDICAT en question (cf. § 4.2.4).

Soit $\mathcal{L}_0$: {obj1, obj2} la liste d'ENTITES résultat de l'EXPANSION de la SPECIFICATION D'ENSEMBLE D'ENTITES. Soit $\mathcal{L}$: {oo1, oo2}la liste d'OPERATED OBJECTS tels que obj1 et obj2 en soient les *entités-ressources*, respectivement.
La FONCTION de sélection *any*(n) permet d'allouer les n premiers OPERATED OBJECTS de $\mathcal{L}$ trouvés disponibles. Quand la PRIMITIVE ACTIVITY SPECIFICATION requiert d'exécuter une OPERATION sur obj1 ou (exclusivement mais indifféremment) sur obj2, on applique *any*(*1*) à $\mathcal{L}$.
La FONCTION *max*() permet d'allouer tous les OPERATED OBJECTS de $\mathcal{L}$ trouvés disponibles. Pour exécuter une OPERATION sur obj1 si c'est possible, puis sur obj2 si c'est possible, ou bien seulement sur obj2, si c'est possible, on applique *max*() à $\mathcal{L}$.
Le choix de la FONCTION *all*() entraîne l'allocation de tous les OPERATED OBJECTS de $\mathcal{L}$, ou un échec si un OPERATED OBJECT est trouvé indisponible ou non allouable. Pour exécuter une OPERATION sur obj1 puis sur obj2, on applique *all*() à $\mathcal{L}$.
Le choix de la FONCTION *disj*() entraîne que chaque OPERATED OBJECT de $\mathcal{L}$ est placé comme seul élément d'une liste, s'il est trouvé disponible, puisque les listes ainsi construites sont des alternatives pour l'allocation d'OPERATED OBJECTS à la PRIMITIVE ACTIVITY SPECIFICATION. *disj*() appliquée à $\mathcal{L}$ établit la disjonction ((obj1) (obj2)).
La FONCTION *sets*(n) établit une liste disjonctive d'ensembles de n OPERATED OBJECTS trouvés disponibles dans $\mathcal{L}$. Cette liste contient autant d'ensembles qu'il y a de manières de répartir la quantité n sur les *k* différentes *particularisations* terminales de la CLASSE D'ENTITES. Pour chaque répartition possible J_i : $\{n_{i,1}, ..., n_{i,k}\}$, avec $\Sigma_{j=1,k}$

[24] Cela entraîne l'arrêt du mécanisme de SIMULATION (voir section 5).

$n_{i,j} = n$, et pour chaque j = 1,..., k, on recherche un ensemble de $n_{i,j}$ INSTANCES dont l'allocation est possible[25]. Le premier trouvé convient, parce qu'affecter celui-ci ou un autre est neutre de tous points de vue. Noter que cette propriété n'est vraie que si tous les PREDICATS de sélection des SPECIFICATIONS D'ENSEMBLE D'ENTITES portant sur la CLASSE d'ENTITES renvoient toujours *true*, c'est-à-dire s'il n'est introduite aucune distinction entre les INSTANCES de la CLASSE D'ENTITES. Les éléments de la liste établie en appliquant *sets*(n) à $\mathcal{L}$ sont des alternatives pour l'allocation d'OPERATED OBJECTS à la PRIMITIVE ACTIVITY SPECIFICATION.
Il faut noter que les spécifications *any*(n), *max*() et *all*() renvoient en fait une liste, comme *disj*() et *sets*(n), mais ne contenant qu'un seul ensemble d'OPERATED OBJECTS. Il n'y a pas d'alternatives, à proprement parler.
Remarquer encore que si la CLASSE D'ENTITES définissant la SPECIFICATION D'ENSEMBLE D'ENTITES est une *particularisation* terminale, *any*(n) et *sets*(n) entraînent l'établissement de la même liste.
La METHODE *current-operated-object*, invoquée au cours du mécanisme ACTING INSTRUCTION LIST, renvoie l'OPERATED OBJECT sur lequel l'OPERATION est en cours de mise en œuvre ou, à défaut, le prochain sur lequel elle doit être mise en œuvre. On verra que c'est le premier, dans l'ordre de construction de la *liste-opérée*, à posséder la VALEUR *faux* pour la PROPRIETE *opéré-totalement* (*cf.* § 5.6.2).
La PROPRIETE *continuation* spécifie, dans le cas où une OPERATION est reprise après avoir été interrompue pendant un délai, qu'elle porte ou non sur les mêmes ENTITES que celles qui en étaient déjà l'objet en l'INSTANT de l'interruption. Seulement si *continuation* est `new`, on opère un nouveau DEVELOPPEMENT de l'OPERATED OBJECT SPECIFICATION ; si la VALEUR est `same-previous`, la spécification requiert d'opérer sur les ENTITES qui étaient présentes dans le résultat de la précédente EXPANSION (avant l'interruption) et qui sont encore dans le résultat de l'EXPANSION en l'INSTANT de la reprise ; si la VALEUR est `same-first`, la spécification requiert d'opérer sur les ENTITES qui étaient présentes dans le résultat de la première EXPANSION (au début de la mise en œuvre de l'activité) et qui sont encore dans le résultat de l'EXPANSION en l'INSTANT de la reprise.

La CLASSE OPERATION SPECIFICATION permet de spécifier l'OPERATION dont l'exécution est requise par la PRIMITIVE ACTIVITY SPECIFICATION. Elle est essentiellement définie par un ATTRIBUT dont la VALEUR est une référence à une *particularisation* d'OPERATION. L'INSTANCE d'OPERATION est alors déterminée par une INSTANCIATION de la *particularisation* d'OPERATION.
Le DEVELOPPEMENT d'une OPERATION SPECIFICATION consiste en cette INSTANCIATION et en les DEVELOPPEMENTS des OPERATION RESOURCE SPECIFICATIONS de l'OPERATION. Chacun de ces DEVELOPPEMENTS renvoie une liste d'ensembles alternatifs de RESOURCES. Ils sont suivis par la constitution d'une liste unique d'ensembles alternatifs de RESOURCES, établie comme le produit cartésien des listes issues des différents DEVELOPPEMENTS.
Les OPERATION RESOURCE SPECIFICATIONS peuvent être déclarés au niveau de l'OPERATION SPECIFICATION, c'est-à-dire en dehors de la définition de l'OPERATION. Leurs DEVELOPPEMENTS sont opérés de la même façon et au même moment.

La CLASSE PERFORMER SPECIFICATION permet de spécifier l'ensemble de PERFORMERS qu'il faut affecter collectivement au service de la PRIMITIVE ACTIVITY SPECIFICATION. Elle est définie par :

- un ATTRIBUT dont la VALEUR est une SPECIFICATION D'ENSEMBLE D'ENTITES. Celle-ci est définie :
  - une CLASSE D'ENTITES *particularisation* de SINGLE PERFORMER, d'HOMOGENEOUS AGGREGATED PERFORMER ou d'HETEROGENEOUS AGGREGATED PERFORMER,
  - (pour mémoire) un PREDICAT de sélection, un *mode* et un *effectif* appropriés.
- une METHODE dont la VALEUR est une référence à une FONCTION de sélection, laquelle peut être *any*, *all*, *max*, *disj* ou *sets* ; les FONCTIONS *any* et *sets* possèdent un argument à valeur entière n.
- une PROPRIETE *continuation*.

Soit $\mathcal{L}$: {perf1, perf2} la liste de PERFORMERS résultat de l'EXPANSION de la SPECIFICATION D'ENSEMBLE D'ENTITES.
La FONCTION de sélection *any*(n) permet d'allouer les n premiers PERFORMERS de $\mathcal{L}$ trouvés `disponibles`. Quand la PRIMITIVE ACTIVITY SPECIFICATION requiert de faire exécuter une OPERATION par perf1 ou (exclusivement mais indifféremment) par perf2, on applique *any*(*1*) à $\mathcal{L}$.

---

[25] Soit par exemple PM et PF deux *particularisations* de P. Les INSTANCES de PM sont {m1, m2, m3} et celles de PF sont {f1, f2}. Soit l'OPERATED OBJECT SPECIFICATION suivante, exprimant que l'INSTRUCTION doit être réalisée sur deux OPERATED OBJECTS de type P :

- SPECIFICATION D'ENSEMBLE D'ENTITES :
  - CLASSE D'ENTITES : P
  - PREDICAT de sélection : `allways_true`
  - *effectif* : *all*
  - *mode* : `return_instances`
- FONCTION de sélection : *sets*(*2*)

Alors, le DEVELOPPEMENT de l'OPERATED OBJECT SPECIFICATION renvoie : ({m1, m2}, {m1, f1}, {f1, f2}). On verra (*cf.* §5.5.2.4.3.2) que si l'ensemble {m1, m2}, par exemple, n'est pas satisfaisant du fait que l'une au moins des deux ressources n'est pas allouable, {m1, m2} est remplacé par {m1, m3} ou {m2, m3}.

La FONCTION *max*() permet d'allouer tous les PERFORMERS de $\mathcal{L}$ trouvés `disponibles`. Pour faire exécuter une OPERATION par perf1 si c'est possible, et par perf2 si c'est possible, ou bien seulement par perf2, si c'est possible, on applique *max*() à $\mathcal{L}$.
Le choix de la FONCTION *all*() entraîne l'allocation de tous les PERFORMERS de $\mathcal{L}$, ou un échec si un PERFORMER est trouvé `indisponible` ou non allouable. Pour faire exécuter une OPERATION par perf1 et par perf2, on applique *all*() à $\mathcal{L}$.
Le choix de la FONCTION *disj*() entraîne que chaque PERFORMER de $\mathcal{L}$ est placé comme seul élément d'une liste, s'il est trouvé `disponible`, puis que les listes ainsi construites sont des alternatives pour l'allocation de PERFORMERS à la PRIMITIVE ACTIVITY SPECIFICATION. *disj*() appliquée à $\mathcal{L}$ établit la disjonction ((perf1) (perf2)).
La FONCTION *sets(*`n`*)* est définie ici de la même manière que dans l'article sur l'OPERATED OBJECT SPECIFICATION. C'est dire que les éléments de la liste établie avec la spécification *sets(*`n`*)* sont des alternatives pour l'affectation de PERFORMERS au service de la PRIMITIVE ACTIVITY SPECIFICATION.
Le mécanisme de DEVELOPPEMENT de la PERFORMER SPECIFICATION invoque l'EXPANSION de la SPECIFICATION D'ENSEMBLE D'ENTITES, puis la FONCTION de sélection sur l'ensemble issu de l'EXPANSION.
Par convention, la *puissance* allouée à l'INSTRUCTION est la somme des *puissances* des PERFORMERS dans l'alternative choisie dans la liste renvoyée par la FONCTION de sélection. En l'absence de PERFORMER SPECIFICATION, une INSTRUCTION est censée être mise en œuvre par un PERFORMER virtuel dont la *puissance* est *1*.
La PROPRIETE *continuation* est utilisée de la même manière que dans les OPERATION RESOURCE SPECIFICATIONS.

La CLASSE ACTIVITY INCONSISTENCY CONDITION est une INCONSISTENCY CONDITION qui permet d'exprimer que l'engagement conjoint d'un ensemble de RESOURCES (ou bien l'engagement d'un ensemble de RESOURCES pour une certaine OPERATION) dans une PRIMITIVE ACTIVITY SPECIFICATION est impossible ou inacceptable.
La VALEUR de la PROPRIETE *classe-entité-contrainte* est une *particularisation* de PRIMITIVE ACTIVITY SPECIFICATION, dont toutes les INSTANCES sont sujettes à la contrainte.
L'OCCURRENCE CROSS DOMAIN est interprété comme les co-engagements (internes à la PRIMITIVE ACTIVITY SPECIFICATION) interdits pour cette *particularisation*.
La METHODE HOLDING CONDITION décrit dans quelles conditions les co-engagements sont interdits.
On peut attacher une ACTIVITY INCONSISTENCY CONDITION à une PRIMITIVE ACTIVITY SPECIFICATION ou à un ACTIVITIES RESOURCES BLOCK. Dans ce dernier cas, toutes les PRIMITIVE ACTIVITY SPECIFICATIONS de l'ACTIVITIES RESOURCES BLOCK devront respecter la contrainte. La situation d'inconsistance peut être détectée dès la déclaration de l'ACTIVITIES RESOURCES BLOCK ou de la PRIMITIVE ACTIVITY SPECIFICATION, ou bien seulement de façon dynamique pendant la DETERMINATION de la PRIMITIVE ACTIVITY SPECIFICATION.
Pour exprimer, par exemple, qu'un SINGLE WORKER de CLASSE `W` ne peut jamais utiliser (pour aucune activité et dans aucune condition) une OPERATION RESOURCE de CLASSE `R`, on écrira :

- PRIMITIVE ACTIVITY SPECIFICATION
- ((`W` > *0*) (`R` > *0*))
- `allways_true`

Le mécanisme de SCISSION d'une PRIMITIVE ACTIVITY SPECIFICATION `i` consiste à invoquer la METHODE *split-primitive-activity-spec* du MANAGER, argumentée par `i` et par une liste `L` de deux listes d'OPERATED OBJECTS. Elle remplace chaque occurrence de i dans tout NOMINAL PLAN de la STRATEGY par la même conjonction de deux PRIMITIVE ACTIVITY SPECIFICATIONS `i'` et `i''` (voir au § 4.5.5 l'article ACTIVITY CONJUNCTION SPECIFICATION), en leur donnant respectivement comme objet (c'est-à-dire comme VALEUR de la *liste-opérée* de l'OPERATED OBJECT SPECIFICATION) chacune des deux listes éléments de `L`. La liste `L` peut être renvoyée par une INVOCATION de la METHODE *split-operated-object-spec* du MANAGER, argumentée par `i` (*cf.* § 5.6.2.2).

4.5 Agrégation d'activités

On définit ici les *particularisations* (on dira aussi les types) d'ACTIVITY SPECIFICATION qui permettent d'exprimer les différentes contraintes d'agrégation entre activités. Une ACTIVITY SPECIFICATION d'un de ces types est appelée « activité agrégée ». C'est une ENTITE dont les ELEMENTS sont eux-mêmes des ACTIVITY SPECIFICATIONS et appelés les « composantes » de l'activité. Pour chaque *particularisation*, on donnera ici trois ensembles d'informations, en jeu dans l'évolution de la *situation*[26] :

- les PROPRIETES qu'elle détient en propre, et ses VALEURS spécifiques pour certaines PROPRIETES communes ;

[26] Chaque *particularisation* est aussi caractérisée par les contenus des FONCTIONS qui gèrent l'ajout, le retrait et le remplacement d'une composante, en veillant au maintien de la sémantique des contraintes d'agrégation (opérateurs) et à la cohérence des PROPRIETES (notamment des dates et de la *situation*). Par exemple, le retrait de la composante `b` dans l'activité `meeting(a, b, c)` remplace cette dernière par `before(a, c)` (voir les articles correspondants dans la présente section).

- les préconditions que doivent satisfaire ses composantes, à l'ouverture et à la fermeture de ses propres INSTANCES, ainsi que les préconditions qu'elle doit elle-même satisfaire lors de l'ouverture ou la fermeture d'une composante ;
- les postconditions qui définissent le mécanisme de PROPAGATION de *situation*, c'est-à-dire de changement de *situation* d'une activité en fonction de changements de *situation* de l'activité qui l'agrège, d'une des activités qu'elle agrège, ou bien d'une des activités agrégées avec elle dans une autre.

Il n'est pas obligatoire de doter une activité agrégée de VALEURS significatives pour les PROPRIETES *fin-au-plus-tôt , fin-au-plus-tard*, ni d'un PREDICAT *condition-fermeture*. A l'exception des ITERATION ACTIVITY SPECIFICATIONS (voir plus loin), il en est de même pour les éléments de la condition d'ouverture.

La mise en œuvre d'une ACTIVITY SPECIFICATION non primitive échoue[27] si la mise en œuvre d'une des PRIMITIVE ACTIVITY SPECIFICATIONS qu'elle agrège échoue, ou si une précondition à son ouverture ou sa fermeture ne peut plus être satisfaite[28].

Les passages aux *situations* `waiting`, `open` et `closed` sont soumis à des préconditions générales : la *situation* doit être, respectivement, `sleeping`, `waiting` ou pouvant le devenir immédiatement, et `open`. Le passage de la *situation* à `open` et `closed` est conditionné à la non présence d'un prédicat (d'ouverture ou de fermeture, respectivement) ou, s'il y a un prédicat, à sa satisfaction. Le passage à `closed` de la *situation* d'une PRIMITIVE ACTIVITY SPECIFICATION est en outre conditionné à la terminaison de l'OPERATION en jeu (situation `terminated`), plus exactement au fait que son *degré-de-progression* soit au moins égal à son *seuil-fermeture*. Si cette OPERATION est permanente (voir § 4.7.2), cette dernière condition (non satisfaite par construction) n'est pas testée.

Les activités agrégées, sauf l'itération d'activités, sont dotées d'une méthode *modify-in-open-situation*, exécutée automatiquement par le mécanisme UPDATING SITUATION) quand on lui a donné un corps, et si l'activité est `open`. Elle permet une évolution dynamique du NOMINAL PLAN.

4.5.1 Contraintes de précédence

Une BEFORE ACTIVITY SPECIFICATION permet d'exprimer que deux ou plusieurs activités doivent se succéder dans le temps sans aucun recouvrement. En d'autres termes, une composante ne peut être ouverte (passage à `open`) avant que la précédente ne soit fermée (passage à `closed`).

Cette « précédence » des INSTRUCTIONS implique la précédence de l'exécution des OPERATIONS qui les composent respectivement.

L'INVOCATION de *fenêtre-début* renvoie le couple {*début-au-plus-tôt, début-au-plus-tard*} ou, à défaut, l'INVOCATION de *fenêtre-début* sur la première activité composante.

► Propriétés :

La PROPRIETE *opérateur-sur-sous-activités* a la VALEUR `before`.

La PROPRIETE *délais-o-o* a pour éventuelle VALEUR une liste de *n-1* couples [*x*, *y*], où *n* est le nombre d'activités composantes. Les entiers *x* et *y* sont tels que l'ouverture d'une composante doit intervenir entre *d*+*x* et *d*+*y*, si *d* est la *date-ouverture* de l'activité précédente. Valeur par défaut : [*0*, ∝].

La PROPRIETE *délais-f-o* a pour éventuelle VALEUR une liste de *n-1* couples [*x*, *y*], où *n* est le nombre d'activités composantes. Les entiers *x* et *y* sont tels que l'ouverture d'une composante doit intervenir entre *d*+*x* et *d*+*y*, si *d* est la *date-fermeture* de l'activité précédente. Valeur par défaut : [*1*, ∝].

Noter que la valeur par défaut pour *délais-f-o* donne à une BEFORE ACTIVITY SPECIFICATION les propriétés d'une MEETING ACTIVITY SPECIFICATION (voir plus loin).

► Préconditions :

Pour qu'une activité de type `before` puisse devenir :
- `waiting`, il faut que sa première composante puisse devenir `waiting` immédiatement ;
- `open`, il faut que sa première composante puisse devenir `open` immédiatement ;
- `closed`, il faut que sa dernière composante puisse devenir `closed` immédiatement.

Pour que la première composante d'une activité de type `before` puisse devenir :
- `waiting`, il faut que celle-ci puisse le devenir immédiatement ;
- `open`, il faut que celle-ci puisse le devenir immédiatement.

Pour qu'une autre composante, après la première, puisse devenir :
- `waiting`, il faut que la précédente soit `closed` ou bien qu'elle soit `open` et qu'elle puisse devenir `closed` immédiatement.

► Postconditions :

Dès qu'une activité de type `before` devient :
- `waiting`, la première activité composante devient `waiting` ;
- `open`, la première activité composante devient `open` ;

---

27 Cela entraîne l'arrêt du mécanisme de SIMULATION (voir section 5).

28 Sauf s'il s'agit d'une OPTIONAL ACTIVITY SPECIFICATION (activité optionnelle).

- `closed`, la dernière activité composante devient `closed`.

Dès qu'une activité composante d'une activité de type `before` devient :

- `waiting` et si elle est la première composante, alors l'activité `before` devient `waiting`. Si elle n'est pas la première composante, alors la composante précédente devient `closed` si elle ne l'est pas déjà ;
- `open` et si elle est la première composante, alors l'activité `before` devient `open` ; si la composante précédente (le cas échéant) sous-tend une OPERATION permanente (voir § 4.7.2), cette composante devient `closed` et l'OPERATION `terminated`.
- `closed` et qu'elle est la dernière composante, alors l'activité `before` devient `closed`, `si autorisé`. Si elle n'est pas la dernière composante, l'activité qui la suit devient `waiting si elle le peut`.

Une MEETING ACTIVITY SPECIFICATION permet d'exprimer que deux ou plusieurs activités doivent se succéder dans le temps sans aucun délai entre la fermeture d'une composante (passage à `closed`) et l'ouverture de la suivante (passage à `open`). En d'autres termes, si une composante est fermée en *d*, la suivante est ouverte en *d+1*.

L'INVOCATION de *fenêtre-début* renvoie le couple {*début-au-plus-tôt, début-au-plus-tard*} ou, à défaut, l'INVOCATION de *fenêtre-début* sur la première activité composante.

► Propriétés :

La PROPRIETE *opérateur-sur-sous-activités* a la VALEUR `meeting`.

► Préconditions :

Pour qu'une activité de type `meeting` puisse devenir :

- `waiting`, il faut que sa première composante puisse devenir `waiting` immédiatement ;
- `open`, il faut que sa première composante puisse devenir `open` immédiatement ;
- `closed`, il faut que sa dernière composante puisse devenir `closed` immédiatement.

Pour que la première composante d'une activité de type `meeting` puisse devenir :

- `waiting`, il faut que celle-ci puisse le devenir immédiatement ;
- `open`, il faut que celle-ci puisse le devenir immédiatement.

Pour qu'une autre composante, après la première, puisse devenir :

- `waiting ou open`, il faut que la précédente soit `open` et qu'elle puisse devenir `closed` immédiatement.

Pour qu'une autre composante, avant la dernière, puisse devenir :

- `closed`, il faut que la suivante puisse devenir `open` immédiatement.

► Postconditions :

Dès qu'une activité de type `meeting` devient :

- `waiting`, la première activité composante devient `waiting` ;
- `open`, la première activité composante devient `open` ;
- `closed`, la dernière activité composante devient `closed`.

Dès qu'une activité composante d'une activité de type `meeting` devient :

- `waiting` et si elle est la première composante, alors l'activité `meeting` devient `waiting`. Si elle n'est pas la première composante l'activité qui la précède devient `closed`.
- `open` et si elle est la première composante, alors l'activité `meeting` devient `open`. Si elle n'est pas la première composante l'activité qui la précède devient `closed`.
- `closed` et qu'elle est la dernière composante, alors l'activité `meeting` devient `closed`, si autorisé. Si elle n'est pas la dernière composante, l'activité qui la suit devient `waiting` puis `open`.

Une STRONG MEETING ACTIVITY SPECIFICATION permet d'exprimer que deux ou plusieurs PRIMITIVE ACTIVITY SPECIFICATIONS doivent se succéder dans le temps sans aucun délai entre la fin de l'OPERATION liée à une composante (passage de `executing` à `terminated`) et le début de l'OPERATION liée à la suivante (passage de `dormant` à `executing`).

► Propriétés :

La PROPRIETE *opérateur-sur-sous-activités* a la VALEUR `strong_meeting`.

► Préconditions :

Pour qu'une activité de type `strong_meeting` puisse devenir :

- `waiting`, il faut que sa première composante soit puisse devenir `waiting` immédiatement ;
- `open`, il faut que sa première composante puisse devenir `open` immédiatement ;
- `closed`, il faut que sa dernière composante puisse devenir `closed` immédiatement.

Pour que la première composante d'une activité de type `strong_meeting` puisse devenir :

- `waiting`, il faut que celle-ci puisse le devenir immédiatement ;
- `open`, il faut que celle-ci puisse le devenir immédiatement.

Pour qu'une autre composante, après la première, puisse devenir :

- `waiting ou open`, il faut que la *condition-de-faisabilité* de l'OPERATION en jeu soit vérifiée, et que la composante précédente soit `open` et qu'elle puisse devenir `closed` immédiatement avec un *degré-de-progression* égal à *1*.

Pour qu'une autre composante, avant la dernière, puisse devenir :

- `closed`, il faut que la suivante puisse devenir `open` immédiatement et que la *condition-de-faisabilité* de l'OPERATION qui y est en jeu soit vérifiée.

► Postconditions :

Les règles de propagation de *situation* sont les mêmes que pour une MEETING ACTIVITY SPECIFICATION.
Une particularité est que dès qu'une composante d'une activité `strong_meeting` devient `closed` et qu'elle n'est pas la dernière, l'activité qui la suit devient certes `waiting` puis `open`, mais de plus l'exécution immédiate de l'OPERATION en jeu est requise (on verra en section 5.5 qu'elle est assortie de la notation *req*).

Une UNORDERED-BEFORE ACTIVITY SPECIFICATION permet d'exprimer que deux ou plusieurs activités doivent se succéder, sans recouvrement, dans un ordre quelconque. Cette séquence n'est pas exclusive, au sens où d'autres éléments du NOMINAL PLAN peuvent autoriser ou contraindre une autre activité à s'intercaler entre deux activités de cette séquence.
La spécification `before_any` (a, b) est équivalente à `or`(`before`(a, b), `before`(b, a)). Par contre, la spécification `before_any` (a, b, c) n'est pas équivalente à `or` (`before`(a, b, c), `before` (a, c, b), `before`(b, a, c),...). En effet, cette dernière expression impose, à l'INSTANT du choix de la première activité entre a, b et c, de choisir aussi l'ordre de réalisation des deux autres (puisque toutes les composantes d'une DISJUNCTION ACTIVITY SPECIFICATION sauf une passent à la *situation* `cancelled`).
Une récriture toujours valide de `before_any` (a, b, c) est `before`(`or`(a, b, c), `before_any`()), avec la convention que le choix de la première activité entre a, b et c provoque le transfert dans le second élément `before_any` les activités non choisies. L'opérateur `before_any` se comporte alors comme une séquence `before` de deux éléments, dont le second est reconstruit dynamiquement et récursivement.
Ainsi, dans le mécanisme MAKING INSTRUCTION LIST, la FONCTION *expand-to-disjunction* de cette *particularisation* donne aux composantes alternatives (plus exactement aux PRIMITIVE ACTIVITY SPECIFICATIONs qu'elles contiennent) le statut `open` : au plus une seule sera choisie pour sa mise en œuvre dans le mécanisme ACTING INSTRUCTION LIST. Dès lors les composantes `open` non choisies sont renvoyées `sleeping` dans le second élément de la séquence. Elles ne pourront devenir `waiting` qu'après que la composante choisie sera devenue `closed`. On assure ainsi la succession sans recouvrement des composantes.

► Propriétés :

La PROPRIETE *opérateur-sur-sous-activités* a la VALEUR `before_any`.

► Préconditions :

Les préconditions aux passages d'une activité de type `before_any`, ou d'une de ses composantes, aux *situations* `waiting`, `open` et `closed` sont conceptuellement les mêmes que pour une CONJUNCTION ACTIVITY SPECIFICATION.
Dans la récriture en un opérateur `before` de deux elements, ces préconditions sont les mêmes que pour une BEFORE ACTIVITY SPECIFICATION

► Postconditions :

Dès qu'une activité de type `before_any` devient :

- `waiting`, la première activité composante devient `waiting` ;
- `open`, la première activité composante devient `open` ;
- `closed`, les deux activités composantes deviennent `closed`.

Dès qu'une des composantes d'une activité de type `before_any` devient :

- `waiting`, celle-ci devient `waiting` ;
- `open`, celle-ci devient `open`, et les autres deviennent `sleeping` ;
- `closed`, les autres activités composantes passent de la situation `sleeping` à la situation `waiting`.

Dès que toutes les composantes d'une activité de type `before_any` sont devenues :

- `closed`, celle-ci devient `closed`.

Les PROPRIETES *délais-o-o* et *délais-f-o* ne peuvent pas être exploitées pour cet opérateur, puisque sa VALEUR est une liste de *n-1* couples [*x*, *y*], ordonnée comme les *n* activités composantes, non pré-ordonnées par définition.

Une UNORDERED-MEETING ACTIVITY SPECIFICATION permet d'exprimer que deux ou plusieurs activités doivent se succéder dans le temps sans aucun délai entre la fermeture d'une composante (passage à `closed`) et l'ouverture de la suivante (passage à `open`). En d'autres termes, si une composante est fermée en *d*, la suivante est ouverte en *d+1*.
Comme dans une UNORDERED-BEFORE ACTIVITY SPECIFICATION, l'ordre de mise en œuvre des activités composantes n'est pas pré-déterminé statiquement, mais construit dynamiquement et récursivement.
La PROPRIETE *opérateur-sur-sous-activités* a la VALEUR `meeting_any`.

4.5.2 Contraintes de recouvrement

Une OVERLAPPING ACTIVITY SPECIFICATION permet d'exprimer que deux ou plusieurs activités doivent se recouvrir partiellement dans le temps. En d'autres termes, dans le cas de deux activités, il y a un délai non nul entre l'ouverture de la première composante (passage à `open`) et l'ouverture de la seconde, un délai non nul entre l'ouverture de la seconde et la fermeture de la première (passage à `closed`), et enfin un délai non nul entre la fermeture de la première et la fermeture de la seconde.
L'INVOCATION de *fenêtre-début* renvoie le couple {*début-au-plus-tôt, début-au-plus-tard*} ou, à défaut, l'INVOCATION de *fenêtre-début* sur la première activité composante.

► Propriétés :

La PROPRIETE *opérateur-sur-sous-activités* a la VALEUR `overlapping`.
La PROPRIETE *délais-o-o* a pour éventuelle VALEUR une liste de *n-1* couples [*x*, *y*], où *n* est le nombre d'activités composantes. Les entiers *x* et *y* sont tels que l'ouverture d'une composante doit intervenir entre *d*+*x* et *d*+*y*, si *d* est la *date-ouverture* de l'activité précédente. Valeur par défaut : [*1*, ∝].
La PROPRIETE *délais-o-f* a pour éventuelle VALEUR une liste de *n-1* couples [*x*, *y*]. Les entiers *x* et *y* sont tels que la fermeture d'une composante doit intervenir entre *d*+*x* et *d*+*y*, si *d* est la *date-ouverture* de l'activité suivante. Valeur par défaut : [*1*, ∝].
La PROPRIETE *délais-f-f* a pour éventuelle VALEUR une liste de *n-1* couples [*x*, *y*]. Les entiers *x* et *y* sont tels que la fermeture d'une composante doit intervenir entre *d*+*x* et *d*+*y*, si *d* est la *date-fermeture* de l'activité précédente. Valeur par défaut : [*1*, ∝].
Noter qu'une valeur [*0*, ∝] pour *délais-o-o* ne donne pas à une OVERLAPPING ACTIVITY SPECIFICATION les propriétés d'une COSTARTING ACTIVITY SPECIFICATION. En effet, la valeur *0* est automatiquement transformée en *1*, pour forcer le délai minimum. Même remarque pour une valeur [*0*, ∝] qui serait affectée à *délais-f-f* (on n'obtient pas une COENDING ACTIVITY SPECIFICATION), et pour une valeur [*0*, ∝] qui serait affectée à *délais-o-f* (on n'obtient pas une MEETING ACTIVITY SPECIFICATION),

► Préconditions :

Pour qu'une activité de type `overlapping` puisse devenir :
- `waiting`, il faut que sa première composante puisse devenir `waiting` immédiatement ;
- `open`, il faut que sa première composante puisse devenir `open` immédiatement ;
- `closed`, il faut que sa dernière composante puisse devenir `closed` immédiatement.

Pour que la première composante d'une activité de type `overlapping` puisse devenir :
- `waiting`, il faut que celle-ci puisse le devenir immédiatement ;
- `open`, il faut que celle-ci puisse le devenir immédiatement.

Pour qu'une autre composante, après la première, puisse devenir :
- `waiting`, il faut que la précédente soit `open` ou qu'elle puisse le devenir immédiatement.
- `closed`, il faut que la précédente soit `closed`.

Pour qu'une autre composante que la dernière puisse devenir :
- `closed`, il faut que la suivante soit `open`[29].

► Postconditions :

Dès qu'une activité de type `overlapping` devient :
- `waiting`, la première activité composante devient `waiting` ;
- `open`, la première activité composante devient `open`, la seconde devient `waiting`, si elle le peut ;
- `closed`, la dernière activité composante devient `closed`.

Dès qu'une activité composante d'une activité de type `overlapping` devient :
- `waiting` et si elle est la première composante, alors l'activité `overlapping` devient `waiting` ;. Si elle n'est pas la première composante, alors la composante précédente devient `open` si elle est `waiting` ;
- `open` et si elle est la première composante, alors l'activité `overlapping` devient `open`. Si elle n'est pas la dernière composante, l'activité qui la suit devient `waiting si elle le peut ;`
- `closed` et qu'elle est la dernière composante, alors l'activité `overlapping` devient `closed`.

Une INCLUSION ACTIVITY SPECIFICATION permet d'exprimer que deux ou plusieurs activités doivent se recouvrir totalement dans le temps. En d'autres termes il y a un délai non nul entre l'ouverture d'une composante (passage à `open`) et l'ouverture de la suivante, et un délai non nul entre la fermeture de l'une et la fermeture de la précédente (passage à `closed`).

---

[29] Noter que pour qu'une composante puisse devenir `open`, on n'impose pas que la suivante puisse devenir `waiting`. En effet, la PROPAGATION réalise le passage à `waiting` si c'est posible. Si ce n'est pas immédiatement possible, c'est nécessairement du fait d'une autre contrainte : dès que celle-ci sera levée, le passage à `waiting` sera enfin effectué (voir le développement de l'exemple en section 5.4). La même remarque tient pour les INCLUSION ACTIVITY SPECIFICATIONS et les COENDING ACTIVITY SPECIFICATIONS.

L'INVOCATION de *fenêtre-début* renvoie le couple {*début-au-plus-tôt, début-au-plus-tard*} ou, à défaut, l'INVOCATION de *fenêtre-début* sur la première activité composante.

► Propriétés :

La PROPRIETE *opérateur-sur-sous-activités* a la VALEUR `inclusion`.

La PROPRIETE *délais-o-o* a pour éventuelle VALEUR une liste de *n-1* couples [*x*, *y*], où *n* est le nombre d'activités composantes. Les entiers *x* et *y* sont tels que l'ouverture d'une composante doit intervenir entre *d*+*x* et *d*+*y*, si *d* est la *date-ouverture* de l'activité précédente. Valeur par défaut : [*1*, ∝].

La PROPRIETE *délais-f-f* a pour éventuelle VALEUR une liste de *n-1* couples [*x*, *y*]. Les entiers *x* et *y* sont tels que la fermeture d'une composante doit intervenir entre *d*+*x* et *d*+*y*, si *d* est la *date-fermeture* de l'activité suivante. Valeur par défaut : [*1*, ∝].

Dans la suite des remarques faites à ce niveau pour les OVERLAPPING ACTIVITY SPECIFICATIONS, une INCLUSION ACTIVITY SPECIFICATION ne peut pas devenir une COSTARTING ACTIVITY SPECIFICATIOn, une COENDING ACTIVITY SPECIFICATION ou une EQUALITY ACTIVITY SPECIFICATION par le jeu de valeurs limites pour les délais.

► Préconditions :

Pour qu'une activité de type `inclusion` puisse devenir :

- `waiting`, il faut que sa première composante puisse devenir `waiting` immédiatement ;
- `open`, il faut que sa première composante puisse devenir `open` immédiatement ;
- `closed`, il faut que sa première composante puisse devenir `closed` immédiatement.

Pour que la première composante d'une activité de type `inclusion` puisse devenir :

- `waiting`, il faut que celle-ci puisse le devenir immédiatement ;
- `open`, il faut que celle-ci puisse le devenir immédiatement

Pour qu'une autre composante, après la première, puisse devenir :

- `waiting`, il faut que la précédente soit `open` ou qu'elle puisse le devenir immédiatement ;
- `closed`, il faut que la précédente soit `open`.

► Postconditions :

Dès qu'une activité de type `inclusion` devient :

- `waiting`, la première activité composante devient `waiting` ;
- `open`, la première activité composante devient `open`, la seconde devient `waiting` si elle le peut ;
- `closed`, la première activité composante devient `closed`.

Dès qu'une activité composante d'une activité de type `inclusion` devient :

- `waiting` et si elle est la première composante, alors l'activité `inclusion` devient `waiting`. Si elle n'est pas la première composante, alors la composante précédente devient `open` si elle est `waiting` ;
- `open` et si elle est la première composante, alors l'activité `inclusion` devient `open`. Si elle n'est pas la dernière composante l'activité qui la suit devient `waiting si elle le peut ;`
- `closed` et qu'elle est la première composante, alors l'activité `inclusion` devient `closed`.

Une COSTARTING ACTIVITY SPECIFICATION permet d'exprimer que deux ou plusieurs activités doivent débuter en même temps : leurs *date-ouverture* doivent être égales.

L'INVOCATION de *fenêtre-début* renvoie le couple {*début-au-plus-tôt, début-au-plus-tard*} ou, à défaut, un couple { `m`, `M` } où `m` est le plus grand x_i et `M` est le plus petit y_i parmi les couples { x_i, y_i } renvoyés par les invocations de *fenêtre-début* sur les activités composantes.

► Propriétés :

La PROPRIETE *opérateur-sur-sous-activités* a la VALEUR `costarting`.

► Préconditions :

Pour qu'une activité de type `costarting` puisse devenir :

- `waiting`, il faut que toutes ses composantes puissent devenir `waiting` immédiatement ;
- `open`, il faut que toutes ses composantes puissent devenir `open` immédiatement ;
- `closed`, il faut que toutes les composantes soient `closed` ou puissent le devenir immédiatement.

Pour qu'une composante d'une activité de type `costarting` puisse devenir :

- `waiting`, il faut que celle-ci puisse le devenir immédiatement ;
- `open`, il faut que celle-ci puisse le devenir immédiatement.

► Postconditions :

Dès qu'une activité de type `costarting` devient :

- `waiting`, toutes les activités composantes deviennent `waiting` ;
- `open`, toutes les activités composantes deviennent `open` ;
- `closed`, toutes les activités composantes qui sont `open` deviennent `closed`.

Dès qu'une activité composante d'une activité de type `costarting` devient :

- `waiting`, alors l'activité `costarting` devient `waiting` ;
- `open`, alors l'activité `costarting` devient `open` ;

- `closed`, alors l'activité `costarting` devient `closed` si toutes les autres composantes sont `closed` ou peuvent le devenir immédiatement.

Une STRONG COSTARTING ACTIVITY SPECIFICATION permet d'exprimer que les opérations liées à deux ou plusieurs PRIMITIVE ACTIVITY SPECIFICATIONS doivent débuter en même temps : le passage de `dormant` à `executing` a lieu au même INSTANT.

► Propriétés :

La PROPRIETE *opérateur-sur-sous-activités* a la VALEUR `strong_costarting`.

► Préconditions :

Pour qu'une activité de type `strong_costarting` puisse devenir :
- `waiting`, il faut que toutes ses composantes puissent devenir `waiting` immédiatement ;
- `open`, il faut que toutes ses composantes puissent devenir `open` immédiatement, et que les *condition-de-faisabilité* de toutes les OPERATIONS en jeu soient vérifiées ;
- `closed`, il faut que toutes les composantes soient `closed` ou puissent le devenir immédiatement.

Pour qu'une composante d'une activité de type `strong_costarting` puisse devenir :
- `waiting`, il faut que celle-ci puisse le devenir immédiatement ;
- `open`, il faut que l'activité de type `strong_costarting` puisse devenir `open` immédiatement, et que les *condition-de-faisabilité* de toutes les OPERATIONS en jeu soient vérifiés.

► Postconditions :

Les règles de propagation de *situation* sont les mêmes que pour une COSTARTING ACTIVITY SPECIFICATION.
Une particularité est que dès qu'une une activité `strong_costarting` devient `open`, les OPERATIONS en jeu devront débuter au même INSTANT (on verra en section 5.5 que les composantes seront liées par la notation *un-set*).

Une COENDING ACTIVITY SPECIFICATION permet d'exprimer que deux ou plusieurs activités doivent finir en même temps : leurs *date-fermeture* doivent être égales.
L'INVOCATION de *fenêtre-début* renvoie le couple {*début-au-plus-tôt, début-au-plus-tard*} ou, à défaut, l'INVOCATION de *fenêtre-début* sur la première activité composante.

► Propriétés :

La PROPRIETE *opérateur-sur-sous-activités* a la VALEUR `coending`.

► Préconditions :

Pour qu'une activité de type `coending` puisse devenir :
- `waiting`, il faut qu'il existe au moins une composante qui puisse devenir `waiting` immédiatement ;
- `open`, il faut qu'il existe au moins une composante qui puisse devenir `open` immédiatement ;
- `closed`, il faut que toutes ses composantes puissent devenir `closed` immédiatement.

Pour qu'une composante d'une activité de type `coending` puisse devenir :
- `waiting`, il faut que celle-ci puisse le devenir immédiatement ;
- `open`, il faut que celle-ci puisse le devenir immédiatement.

► Postconditions :

Dès qu'une activité de type `coending` devient :
- `waiting`, toutes les activités composantes qui peuvent devenir `waiting` immédiatement le deviennent ;
- `open`, toutes les activités composantes qui peuvent devenir `open` immédiatement le deviennent ;
- `closed`, toutes les activités composantes deviennent `closed`.

Dès qu'une activité composante d'une activité de type `coending` devient :
- `waiting`, alors l'activité `coending` devient `waiting` ;
- `open`, alors l'activité `coending` devient `open` ;
- `closed`, alors l'activité `coending` devient `closed`.

Une EQUALITY ACTIVITY SPECIFICATION permet de spécifier que deux ou plusieurs activités doivent débuter et finir en même temps : leurs *date-ouverture* doivent être égales ainsi que leurs *date-fermeture*.
L'INVOCATION de *fenêtre-début* renvoie le couple {*début-au-plus-tôt, début-au-plus-tard*} ou, à défaut, un couple { `m`, `M` } construit comme s'il s'agissait d'une COSTARTING ACTIVITY SPECIFICATION.

► Propriétés :

La PROPRIETE *opérateur-sur-sous-activités* a la VALEUR `equality`.

► Préconditions :

Pour qu'une activité de type `equality` puisse devenir :
- `waiting`, il faut que toutes ses composantes puissent devenir `waiting` immédiatement ;
- `open`, il faut que toutes ses composantes puissent devenir `open` immédiatement ;
- `closed`, il faut que toutes ses composantes puissent devenir `closed` immédiatement.

Pour qu'une composante d'une activité de type `equality` puisse devenir :
- `waiting`, il faut que celle-ci puisse le devenir immédiatement ;

- `open`, il faut que celle-ci puisse le devenir immédiatement

► Postconditions :

Dès qu'une activité de type `equality` devient :
- `waiting`, toutes les activités composantes deviennent `waiting` ;
- `open`, toutes les activités composantes deviennent `open` ;
- `closed`, toutes les activités composantes deviennent `closed`.

Dès qu'une activité composante d'une activité de type `equality` devient :
- `waiting`, alors l'activité `equality` devient `waiting` ;
- `open`, alors l'activité `equality` devient `open` ;
- `closed`, alors l'activité `equality` devient `closed`.

Une STRONG EQUALITY ACTIVITY SPECIFICATION permet de spécifier que les OPERATIONS liées à deux ou plusieurs PRIMITIVE ACTIVITY SPECIFICATIONS doivent débuter et finir en même temps : le passage de `dormant` à `executing` a lieu au même INSTANT, de même que le passage de `executing` à `terminated`.

► Propriétés :

La PROPRIETE *opérateur-sur-sous-activités* a la VALEUR `strong_equality`.

► Préconditions :

Pour qu'une activité de type `strong_equality` puisse devenir :
- `waiting`, il faut que toutes ses composantes puissent devenir `waiting` immédiatement ;
- `open`, il faut que toutes ses composantes puissent devenir `open` immédiatement, et que les *condition-de-faisabilité* de toutes les OPERATIONS en jeu soient vérifiées ;
- `closed`, il faut que toutes ses composantes puissent devenir `closed` immédiatement, et que les *degré-de-progression* de toutes les OPERATIONS en jeu soient égaux à *1*.

Pour qu'une composante d'une activité de type `strong_equality` puisse devenir :
- `waiting`, il faut que celle-ci puisse le devenir immédiatement, et que toutes les autres composantes soient `waiting` ou puissent le devenir immédiatement ;
- `open`, il faut que l'activité de type `strong_equality` puisse devenir `open` immédiatement, et que toutes les autres composantes soient `open` ou puissent le devenir immédiatement, et que les *condition-de-faisabilité* de toutes les OPERATIONS en jeu soient vérifiées ;
- `closed`, il faut que l'activité de type `strong_equality` puisse devenir `closed` immédiatement, et que toutes les autres composantes soient `closed` ou puissent le devenir immédiatement, et que les *degré-de-progression* de toutes les OPERATIONS en jeu soient égaux à *1*.

► Postconditions :

Les règles de propagation de *situation* sont les mêmes que pour une EQUALITY ACTIVITY SPECIFICATION.
Une particularité est que dès qu'une une activité `strong_equality` devient `open`, les OPERATIONS en jeu devront débuter au même INSTANT (on verra en section 5.5 que les composantes seront liées par la notation *un-set*).

4.5.3 Itération

Une ITERATION ACTIVITY SPECIFICATION (ou spécification d'activité « itérative ») permet d'exprimer qu'une activité (éventuellement non primitive) dite « itérée », doit être répétée à l'intérieur d'une période, comprise entre l'ouverture et la fermeture de l'activité itérative.

► Propriétés :

L'activité itérative possède obligatoirement des spécifications d'ouverture et de fermeture, soit sous la forme de VALEURS contraignantes pour les PROPRIETES *début-au-plus-tôt, début-au-plus-tard, fin-au-plus-tôt, fin-au-plus-tard*, [30]soit en donnant un contenu contraignant aux deux PREDICATS *condition-ouverture* et *condition-fermeture*, soit en donnant une valeur aux PROPRIETES *nombre-max-de-répétitions* et *nombre-min-de-répétitions*, ou bien enfin par les trois voies conjointement[31].
L'INVOCATION de *fenêtre-début* renvoie le couple { *début-au-plus-tôt, début-au-plus-tard* }.
La PROPRIETE *délais-o-o* a pour éventuelle VALEUR un couple [*x*, *y*]. Les entiers *x* et *y* sont les délais entre la *date-ouverture* d'une activité itérée et, respectivement, le *début-au-plus-tôt* et le *début-au-plus tard* de sa prochaine réalisation. Valeur par défaut : [*1*, ∝].

---

[30] Par défaut, la VALEUR de *début-au-plus-tôt* est *0*, celle de *début-au-plus-tard* est infiniment grande. Les fenêtres d'ouverture et de fermeture peuvent aussi être inférées à partir des activités dont celle-ci est composante (voir annexe 8).

[31] Ces deux contraintes ne sont pas imposées aux ACTIVITY SPECIFICATIONS en général, pour que celles-ci puissent éventuellement ne véhiculer qu'une contrainte symbolique (disjonction, conjonction, itération, optionalité, précédence,..). Elles sont imposées aux ITERATION ACTIVITY SPECIFICATIONS parce que l'inférence n'est pas assurée à partir de la composante, dont les dates ne sont normalement pas spécifiées parce que changeantes au fil des itérations.

La PROPRIETE *délais-f-o* a pour éventuelle VALEUR un couple [*x*, *y*]. Les entiers *x* et *y* sont les délais entre la *date-fermeture* d'une activité itérée et, respectivement, le *début-au-plus-tôt* et le *début-au-plus tard* de sa prochaine réalisation. Valeur par défaut : [*1*, ∝].
La PROPRIETE *opérateur-sur-sous-activités* a la VALEUR `iterate`.

► Préconditions :

Sauf les mécanismes décrits à la fin de cette section, il n'y a pas de préconditions (portant sur l'unique composante) aux passages aux situations `waiting`, `open` et `closed`.
Pour que l'unique composante d'une activité de type `iterate` puisse devenir :

- `waiting`, il faut que celle-ci soit `waiting` ou `open` ;
- `open`, il faut que celle-ci soit `open` ;
- `closed`, il faut que celle-ci soit `open`.

► Postconditions :

Dès qu'une activité de type `iterate` devient :

- `open`, l'unique activité composante devient `waiting` si elle le peut ;
- `closed`, la dernière activité composante devient `closed` si elle le peut ;

Dès que l'unique activité composante d'une activité de type `iterate` devient :

- `closed`, elle redevient immédiatement `waiting`, sauf si les conditions de fermeture de l'activité itérative sont satisfaites à cet INSTANT, auquel cas elle redevient `sleeping`.

Une activité itérée ne peut pas se trouver, ailleurs dans le NOMINAL PLAN, comme ELEMENT d'une activité autre qu'itérative, et ne peut être composante (à quelque profondeur que ce soit) que d'une seule activité itérative.
La répétition se traduit pour l'activité itérée par des transitions de *situation* de `waiting` à `open`, `open` à `closed` et, exclusivement pour cette CLASSE, de `closed` à `waiting` pour permettre une nouvelle ouverture. Ces transitions sont opérées tant que l'activité itérative reste `open`, et sont inhibées après que cette dernière est devenue `closed`, de par ses propres spécifications de fermeture.
La répétition est sous le contrôle de plusieurs éléments :

- les VALEURS des PROPRIETES *délais-o-o* et *délais-f-o*, si présentes, servent conjointement à calculer la nouvelle fenêtre d'ouverture de l'activité itérée, et seulement de celle-ci (*i.e.* à l'exclusion de ses composantes) ;
- on peut limiter le nombre de répétitions et, respectivement imposer un nombre minimum de répétitions, en donnant les valeurs désirées aux descripteurs *nombre-max-de-répétitions* et, respectivement, *nombre-min-de-répétitions* (spécifiques de cette CLASSE).
- la METHODE *update-reactivated-son*, si elle est présente, établit de manière plus globale les nouvelles conditions d'ouverture et de fermeture de l'activité itérée et de ses composantes. On peut ainsi spécifier ou préciser le délai entre une réalisation et l'ouverture de la suivante. Ce délai peut être fonction de la VALEUR de la PROPRIETE *délais-f-o* ou *délais-o-o*, ou bien fonction de l'ETAT du CONTROLLED SYSTEM.

Ces éléments sont pris en compte par le mécanisme d'ITERATION, mis en œuvre lors de chaque fermeture (passage à `closed`) de l'activité itérée ; il procède de la manière suivante :

-l'activivé itérée reçoit la VALEUR de *situation* `sleeping` et la gardera éventuellement ;
-si le nombre maximum d'itérations n'est pas encore atteint, alors :
  - si une METHODE *update-reactivated-son* est présente, elle est invoquée ;
  - sinon :
    - la valeur par défaut (équivalente à la non-spécification) est affectée aux fenêtres d'ouverture et de fermeture des composantes éventuelles de l'activité itérée ;
    - la fenêtre d'ouverture de l'activité itérée est réduite selon les VALEURS de *délais-f-o* et *délais-o-o* si elles sont spécifiées ;
  - si de plus le nombre minimum d'itérations n'est pas encore atteint, alors l'activivé itérée reçoit la VALEUR de *situation* `waiting` ;
  - sinon, la *situation* de l'activité itérée passe à `waiting` seulement si la fenêtre de fermeture n'est pas spécifiée pour l'activité itérative, ou si sa *fin-au-plus-tôt* est postérieure au *début-au-plus-tard* de l'activité itérée (en d'autres termes, si on ne peut pas conclure que l'ouverture de l'activité itérée peut être postérieure à la fermeture de l'activité itérative) ;
-sinon, si la *fin-au-plus-tôt* n'est pas encore atteinte, le plan est déclaré en échec

Lorsque la *situation* de l'activité itérée devient `sleeping` lors de sa fermeture (parce qu'une nouvelle itération n'est pas autorisée), l'activité itérative est elle-aussi fermée, même si la spécification de la fermeture n'est pas compatible, notamment si la *fin-au-plus-tôt* n'est pas encore atteinte.

Lorsque les conditions de fermeture de l'activité itérative sont satisfaites, deux situations peuvent se présenter :

- l'activité itérée est `waiting` (ou `sleeping`, mais ce cas est considéré dans le paragraphe précédent): l'activité itérative devient `closed` et l'activité itérée devient `closed` sans PROPAGATION aux composantes. La *date-fermeture* de l'activité itérée reste ce qu'elle était au moment de son dernier passage à `closed`.
- l'activité itérée est `open` : celle-ci est poursuivie jusqu'à son terme puis n'est pas réitérée ; l'activité itérative devient `closed` avec l'activité itérée. Dans ce cas, et à titre exceptionnel, la fermeture de l'activité itérative peut intervenir après sa *fin-au-plus-tard*.

4.5.4 Optionalité

Une OPTIONAL ACTIVITY SPECIFICATION (ou spécification d'activité option) permet d'exprimer qu'une activité, dite optionnelle et qui en est l'unique composante[32], peut être réalisée ou non, sans que cela constitue un défaut de contrôle du PRODUCTION SYSTEM. La non-réalisation correspond à une des trois situations suivantes :

- l'activité optionnelle n'a pas pu être ouverte dans l'intervalle renvoyé par l'INVOCATION de *fenêtre-début*, ou bien jusqu'à ce que le PREDICAT *max-beg-state-predicate* finisse par renvoyer `vrai`, ou bien la première fois qu'on constate que l'activité optionnelle ne peut pas être ouverte alors que la PROPRIETE *is-one-shot* de l'activité option est `vrai` ;
- il faudrait ouvrir l'activité optionnelle par PROPAGATION, mais sa condition d'ouverture n'est pas satisfaite à cet INSTANT : par exemple on doit pouvoir ouvrir une activité composante d'une activité `meeting` dès que la précédente devient `closed`.
- l'activité optionnelle a pu être ouverte dans l'intervalle renvoyé par l'INVOCATION de *fenêtre-début*, mais en l'INSTANT où la condition de fermeture est satisfaite, aucune OPERATION en jeu dans l'activité n'a connu un début de réalisation (*degré-de-progression = 0*). Cette situation peut provenir de l'indisponibilité constante des RESOURCES requises, ou bien de la non-satisfaction constante de la *condition-de-faisabilité* des OPERATIONS en jeu.

Une ACTIVITY SPECIFICATION composante à une profondeur quelconque d'une OPTIONAL ACTIVITY SPECIFICATION ne peut figurer ailleurs dans le NOMINAL PLAN que comme composante à une profondeur quelconque d'une OPTIONAL ACTIVITY SPECIFICATION. En d'autres termes une activité ne peut pas être déclarée ici optionnelle et là obligatoire.

L'INVOCATION de *fenêtre-début* renvoie le couple {*début-au-plus-tôt, début-au-plus-tard*} ou, à défaut, l'INVOCATION de *fenêtre-début* sur l'unique activité composante.

► Propriétés :

La PROPRIETE *opérateur-sur-sous-activités* a la VALEUR `optional`.

► Préconditions :

Pour qu'une activité de type `optional` puisse devenir :

- `waiting`, il faut que son unique composante puisse devenir `waiting` immédiatement ;
- `open`, il faut que son unique composante puisse devenir `open` immédiatement ;
- `closed`, il faut que son unique composante puisse devenir `closed` immédiatement.

Pour que l'unique composante d'une activité de type `optional` puisse devenir :

- `waiting`, il faut que celle-ci puisse le devenir immédiatement ;
- `open`, il faut que celle-ci puisse le devenir immédiatement ;
- `closed`, il faut que celle-ci puisse le devenir immédiatement.

► Postconditions :

Dès que l'activité option devient :

- `waiting`, l'activité optionnelle devient `waiting` ;
- `open`, l'activité optionnelle devient `open` ;
- `closed`, l'activité optionnelle devient `closed` ;

Dès que l'activité composante optionnelle devient :

- `waiting`, l'activité option devient `waiting` ;
- `open`, l'activité option devient `open` ;
- `closed`, l'activité option devient `closed`.

Dans les trois cas de non-réalisation d'une activité optionnelle, celle-ci passe à la *situation* `closed`, sans examen des préconditions. Cette VALEUR est propagée récursivement aux activités dont elle est composante (à une profondeur quelconque), mais elle ne l'est pas aux composantes de l'activité optionnelle. Puis chaque activité option dont elle est fille devient `cancelled` (VALEUR non propagée) si elle n'a pas été rendue `waiting` dans le cas où c'est une activité itérée. Ainsi, les conditions d'ouverture de l'activité optionnelle (et de ses composantes) ne seront plus examinées.

Cet opérateur peut typiquement représenter la répétition de la tentative d'exécution d'une OPERATION, incluse dans une INSTRUCTION désignée ici par `a`. On déclarera à cet effet `iterate(optional(a))`. Si `a` ne parvient pas à être exécutée, `a` est fermée, donc l'activité `optional` aussi. Une nouvelle activité du même type est alors

[32] Dans l'expression `optional(before(a, b))`, la seule activité optionnelle est `before`.

programmée. De même, la répétition de a avec la possibilité de ne pas l'exécuter une fois sur deux s'écrira iterate(before(optional(a1), a2))}, où a1 et a2 sont deux activités du même type que a. La fermeture de a1 est propagée jusqu'au before, ce qui provoque l'ouverture ultérieure de a2.

Une CONSTRAINED OPTIONAL ACTIVITY SPECIFICATION (ou spécification d'activité option contrainte) permet d'exprimer qu'une activité, dite optionnelle contrainte, peut être réalisée ou non. Cependant le nombre *n* de réalisations effectives au cours des n_2 dernières tentatives de réalisation doit être au moins égal à n_1 ($0 < n_1 < n_2$). Il y a défaut de contrôle du PRODUCTION SYSTEM dès que *n* devient inférieur à n_1.
Ce comportement est seulement exploitable lorsque l'activité optionnelle contrainte est une composante, à une profondeur quelconque d'une ITERATION ACTIVITY SPECIFICATION.

► Propriétés :

Cette *particularisation* d'ACTIVITY SPECIFICATION possède les PROPRIETES et les METHODES d'une OPTIONAL ACTIVITY SPECIFICATION. Elle possède en outre et en propre trois autres PROPRIETES :

- la *contrainte-sur-réalisations* est le couple d'entiers $\{n_1, n_2\}$;
- la *file-des-réalisations* est une file de booléens (first in, first out) de taille n_2 dont tous les éléments sont initialisés avec *true* ; l'élément le plus à droite est le témoin de la dernière réalisation ; celui le plus à gauche est le témoin de la réalisation de rang n_2-*1* avant la dernière ;
- le *nombre-de-réalisations* est le nombre de booléens *true* dans la *file-des-réalisations* (*n*).

La PROPRIETE *opérateur-sur-sous-activités* a la VALEUR constrained-optional.

► Préconditions :

Elles sont identiques à celles qui régissent une OPTIONAL ACTIVITY SPECIFICATION.

► Postconditions :

Les contraintes de PROPAGATION sont les mêmes que pour une OPTIONAL ACTIVITY SPECIFICATION.

Lorsque la *situation* de l'activité optionnelle contrainte passe à la VALEUR closed, on enlève le premier élément à gauche de la *file-des-réalisations* (celui le plus anciennement installé) et on ajoute un élément à droite. On est alors dans une des deux situations suivantes, quant au déterminant du passage à closed :

- on a tenté d'ouvrir l'activité optionnelle contrainte après sa fenêtre de début : l'élément ajouté à droite est alors *false* ;
- dans le cas contraire, l'élément ajouté à droite est *true* (les conditions d'ouverture avaient été normalement satisfaites, et l'activité est maintenant fermée).

Ainsi la VALEUR de *nombre-de-réalisations* est le nombre de réalisations effectives de l'activité optionnelle contrainte parmi ses n_2 dernières tentatives de réalisation. En l'INSTANT de sa fermeture, la non réalisation d'une activité optionnelle contrainte ne provoque l'échec du NOMINAL PLAN que si le *nombre-de-réalisations* est inférieur à n_1.
Noter qu'une CONSTRAINED OPTIONAL ACTIVITY SPECIFICATION dont n_1 est égal à *0* a le même sens qu'une OPTIONAL ACTIVITY SPECIFICATION. Une CONSTRAINED OPTIONAL ACTIVITY SPECIFICATION telle que $n_1 = n_2$ n'est pas une activité optionnelle.

4.5.5 Disjonction et conjonction

Une DISJUNCTION ACTIVITY SPECIFICATION permet d'exprimer un choix entre deux ou plusieurs activités alternatives. Lorsqu'une est choisie et voit son exécution débuter, les autres sont annulées (la *situation* passe à cancelled), c'est-à-dire que les OPERATIONS qu'elles induisent ne pourront plus être exécutées.
C'est le mécanisme MAKING INSTRUCTION LIST qui opère le choix entre les activités alternatives, en invoquant la METHODE *expand-to-disjunction* sur l'ACTIVITY SPECIFICATION qui définit le NOMINAL PLAN. Si l'ACTIVITY SPECIFICATION contient, à une profondeur quelconque, une ou plusieurs DISJUNCTION ACTIVITY SPECIFICATIONS, la METHODE *expand-to-disjunction* renvoie un ensemble de jeux de PRIMITIVE ACTIVITY SPECIFICATIONS tel que deux activités alternatives (composantes d'un or à une profondeur quelconque) ne se trouvent jamais dans le même jeu.
Noter que si les conditions d'ouverture des activités alternatives ne sont pas satisfaites au même INSTANT, la situation réelle de choix (par les PREFERENCE RULES) n'est pas rencontrée. Il en est de même si une indisponibilité de RESOURCES fait qu'une ou plusieurs activités alternatives ne se retrouvent pas dans l'ensemble de jeux d'INSTRUCTIONS soumis aux PREFERENCE RULES.

► Propriétés :

La PROPRIETE *opérateur-sur-sous-activités* a la VALEUR or.

► Préconditions :

Pour qu'une activité de type or puisse devenir :

- waiting, il faut qu'il existe au moins une composante qui puisse devenir waiting immédiatement ;
- open, il faut qu'il existe au moins une composante qui puisse devenir open immédiatement ;

- `closed`, il faut que chaque composante soit `cancelled`, ou `closed` ou puisse devenir `closed` immédiatement.

Pour qu'une composante d'une activité de type `or` puisse devenir :
- `waiting`, il faut que celle-ci puisse le devenir immédiatement ;
- `open`, il faut que celle-ci puisse le devenir immédiatement

► Postconditions :

Dès qu'une activité de type `or` devient :
- `waiting`, toutes les activités composantes qui peuvent devenir `waiting` immédiatement le deviennent ;
- `open`, toutes les activités composantes qui peuvent devenir `open` immédiatement le deviennent (on rappelle qu'une seule sera choisie et restera `open`, alors que les autres deviendront `cancelled`) ;
- `closed`, l'unique activité `open` devient `closed`.

Dès qu'une activité composante d'une activité de type `or` devient :
- `waiting`, alors l'activité `or` devient `waiting` ;
- `open`, alors l'activité `or` devient `open` ;
- `closed` (c'est la seule composante a avoir été `open`), alors l'activité `or` devient `closed`.

Une CONJUNCTION ACTIVITY SPECIFICATION permet d'exprimer que deux ou plusieurs PRIMITIVE ACTIVITY SPECIFICATIONS doivent être finies (en atteignant la *situation* `closed`). Plus généralement, cette CLASSE permet de spécifier que deux ou plusieurs contraintes doivent être conjointement satisfaites, autrement dit que deux ou plusieurs *opérateur-sur-sous-activités* doivent être conjointement pris en compte. Par exemple, pour exprimer que l'activité `a` doit précéder les deux activités `b` et `c` (ces deux dernières dans un ordre quelconque), on spécifiera `and(before(a, b), before(a, c))`.

Noter que cet opérateur conjonctif n'impose aucune contrainte temporelle sur ses composantes et qu'il n'y a pas d'intérêt à l'utiliser avec une seule composante.

► Propriétés :

La PROPRIETE *opérateur-sur-sous-activités* a la VALEUR `and`.

► Préconditions :

Pour qu'une activité de type `and` puisse devenir :
- `waiting`, il faut qu'il existe au moins une composante qui puisse devenir `waiting` immédiatement ;
- `open`, il faut qu'il existe au moins une composante qui puisse devenir `open` immédiatement ;
- `closed`, il faut que toutes les composantes soient `closed` ou puissent le devenir immédiatement.

Pour qu'une composante d'une activité de type `and` puisse devenir :
- `waiting`, il faut que celle-ci puisse le devenir immédiatement ;
- `open`, il faut que celle-ci puisse le devenir immédiatement.

► Postconditions :

Dès qu'une activité de type `and` devient :
- `waiting`, toutes les activités composantes qui peuvent devenir `waiting` immédiatement le deviennent ;
- `open`, toutes les activités composantes qui peuvent devenir `open` immédiatement le deviennent.
- `closed`, les activités composantes qui sont `open` deviennent `closed`.

Dès qu'une activité composante d'une activité de type `and` devient :
- `waiting`, alors l'activité `and` devient `waiting` ;
- `open`, alors l'activité `and` devient `open` ;
- `closed`, alors l'activité `and` devient `closed si toutes les autres composantes sont closed ou peuvent le devenir immédiatement.`

Une IMPLICIT CONJUNCTION ACTIVITY SPECIFICATION permet d'exprimer de façon concise qu'une même ACTIVITY SPECIFICATION doit être appliquée à un ensemble d'OPERATED OBJECTS (au sens de la conjonction).

► Propriétés :

Par rapport à la CLASSE ACTIVITY SPECIFICATION, celle-ci possède un ATTRIBUT spécifique complémentaire, dont la VALEUR est :
- une SPECIFICATION D'ENSEMBLE D'ENTITES, dont l'EXPANSION , complétée par la recherche d'OPERATED OBJECTS correspondants déjà instanciés ou leur INSTANCIATION, fournira l'ensemble d'OPERATED OBJECTS.

L'unique composante (ELEMENT) de cette *particularisation* est l'INSTANCE d'ACTIVITY SPECIFICATION à appliquer conjointement à l'ensemble issu de cette EXPANSION. Cette INSTANCE ne peut figurer qu'une seule fois (en tant que composante) dans le NOMINAL PLAN.

Les PRIMITIVE ACTIVITY SPECIFICATIONS incluses dans cette unique activité composante doivent contenir une OPERATED OBJECT SPECIFICATION non renseignée.

La PROPRIETE *opérateur-sur-sous-activités* a la VALEUR `implicit_and`.

Le mécanisme d'EXPLICITATION ne fait qu'invoquer la METHODE *expand-to-conjunction*, spécifique de la présente CLASSE. Elle invoque d'abord l'EXPANSION de la SPECIFICATION D'ENSEMBLE D'ENTITES. Pour chaque ENTITE de l'ensemble renvoyé, une copie de l'INSTANCE d'ACTIVITY SPECIFICATION composante est créée en y remplaçant l'OPERATED OBJECT SPECIFICATION (non

renseignée) par cette ENTITE. Une CONJUNCTION ACTIVITY SPECIFICATION est enfin créée, avec ces ACTIVITY SPECIFICATIONS ainsi modifiées comme composantes. Par exemple, l'expression `implicit_and`((`o1`, `o2`), `before`(< `?`, `t`, `p`>,< `?`, `u`, `q` >)) –où (`o1`, `o2`) est censé représenter le résultat de l'EXPANSION à un INSTANT donné de la SPECIFICATION D'ENSEMBLE D'ENTITES- est traduite en `and`(`before`(< `o1`, `t`, `p`>,< `o1`, `u`, `q` >), `before`(< `o2`, `t`, `p`>,< `o2`, `u`, `q` >)).
Noter que, puisque les conditions d'ouverture et de fermeture sont copiées, toutes les composantes d'une activité `implicit_and` seront ouvertes et fermées en même temps (sauf le cas particulier où les conditions d'ouverture et de fermeture des différentes composantes dépendent de l'ETAT des différentes ENTITES qui leur sont associées après l'EXPANSION de la SPECIFICATION D'ENSEMBLE D'ENTITES).
La METHODE *expand-to-disjunction* invoquée pour une activité de cette CLASSE est en fait appliquée à la CONJUNCTION ACTIVITY SPECIFICATION qui est le résultat de l'EXPLICITATION.

► Les préconditions et postconditions sont celles d'une CONJUNCTION ACTIVITY SPECIFICATION, appliquées à la CONJUNCTION ACTIVITY SPECIFICATION issue de l'EXPLICITATION.

4.6 Activités de transfert

Une TRANSFER ACTIVITY (ou activité de transfert) est une spécification d'activité primitive qui exprime, en plus des PROPRIETES souhaitées de l'objet opéré, lequel est considéré comme la source d'un transfert, les PROPRIETES souhaitées de l'objet qui en sera la destination. La nature de l'OPERATION n'est pas libre au modélisateur : c'est nécessairement une opération de transfert, instance d'une des deux classes prédéfinies UNIT-GRANULATED TRANSFER OPERATION et QUANTITY-GRANULATED TRANSFER OPERATION (voir § 4.7.3). La prédéfinition de ces opérations, et notamment de la procédure qui régit la progression du transfert de la source vers la destination, demande au modélisateur de spécifier ce qui est transféré, sous la forme d'un ATTRIBUT de l'activité. La VALEUR de cet ATTRIBUT est une instance de TRANSFER INFORMATION.
La destination est spécifiée par un ATTRUBUT complémentaire dont la VALEUR est une OPERATED OBJECT SPECIFICATION.

Une TRANSFER INFORMATION est attachée à une activité de transfert pour lui spécifier ce qui doit être transféré de la source vers la destination. Un transfert peut être conçu et modélisé de deux manières différentes. Dans un premier cas, ce qui est transféré est tout ou partie des ELEMENTS de la source, qui deviennent ELEMENTS de la destination. Si la source est un ensemble d'objets, la spécification d'objets opérés (OPERATED OBJECT SPECIFICATION) leur donne un ordre (voir § 4.4) et le retrait d'ELEMENTS les touchera tour à tour dans cet ordre, un nouvel objet n'étant touché que lorsque le précédent n'a plus d'ELEMENTS. Si la destination est un ensemble d'objets, lui aussi ordonné par sa spécification, c'est le premier qui incorpore tous les ELEMENTS transférés, sauf si c'est une RESSOURCE dotée d'une capacité limitée. Dans ce dernier cas les ELEMENTS de la source qui n'ont pas pu devenir ELEMENTS d'un objet de la destination deviennent ELEMENTS, autant que possible, de l'objet suivant. La spécification d'un tel transfert se fait par trois PROPRIETES de la TRANSFER INFORMATION : l'*identificateur* prend la VALEUR `elements`, l'*option* prend une des deux VALEURS `whole` ou `part`, enfin la *quantité*, seulement si l'*option* est `part`, indique le nombre d'ELEMENTS à transférer.
Dans un deuxième cas, la nature de ce qui est transféré est spécifiée en donnant comme VALEUR à la PROPRIETE *identificateur* la référence à une PROPRIETE commune des objets de la source et de la destination. Par exemple, si on transfère des passagers d'un avion vers le hall d'arrivée, ces deux ENTITES doivent posséder une PROPRIETE commune dont la VALEUR a le sens d'un nombre de passagers, et la référence à cette PROPRIETE est la VALEUR de l'*identificateur*. La PROPRIETE *quantité* indique le nombre de passagers à transférer. L'*option* n'est pas utilisée. En fait, il existe deux PROPRIETES avec le sens de *quantité* : l'une pour les PROPRIETES à VALEURS entières et une autre pour celles à VALEURS réelles (pour transférer, par exemple, une quantité de matière d'un réservoir à un autre).

4.7 Opérations

Une OPERATION est une ENTITE qui permet de spécifier la nature et les modalités d'un changement d'ETAT d'une partie du CONTROLLED SYSTEM, changement décidé intentionnellement par le MANAGER et opéré en utilisant des RESOURCES de l'OPERATING SYSTEM. Ce changement d'ETAT n'est donc pas provoqué par les PROCESSUS internes du CONTROLLED SYSTEM, ceux qui ont cours même quand il n'est soumis à aucune activité de contrôle. La mise en œuvre des OPERATIONS est faite par le mécanisme ACTING INSTRUCTION LIST.
Les PROPRIETES d'une OPERATION sont les suivantes : *classe-entité*, *pas*, *mode-de-progression*, *degré-de-progression*, *vitesse*, *situation*, *date-début*, *date-fin*, *durée-brute*, *durée*, *ressources-requises*, *ressource-usage*, *seuil-suspension* et *priorité-d-exécution*.
Cette CLASSE possède trois METHODES : *procédure-transition-état, condition-de-faisabilité* et *calcul-vitesse*.
La *procédure-transition-état* est une PROCEDURE qui porte sur une INSTANCE de la CLASSE *classe-entité*, qui joue alors le rôle d'objet de l'OPERATION. Elle réalise, pour cette INSTANCE qui en est l'argument, le passage de l'ETAT courant à un autre.

Le *pas* est l'intervalle entre deux INSTANTS successifs auxquels la modification d'ETAT provoquée par l'OPERATION peut être perçue par les autres ENTITES. Sa VALEUR est normalement fixée au plus grand intervalle tel qu'aucun des ETATS intermédiaires ne soient jamais d'intérêt pour ces autres ENTITES.
Selon le *mode-de-progression*, la modification d'ETAT entre les deux INSTANTS définissant le *pas* est réalisée par une seule ou bien plusieurs INVOCATIONS de la *procédure-transition-état*.
Le *degré-de-progression* a une VALEUR comprise entre *0* et *1*. La VALEUR *0* dénote que l'OPERATION n'a pas encore eu d'effet. La VALEUR est *1* quand et seulement quand l'OPERATION a eu son effet complet sur les OPERATED OBJECTS de l'INSTRUCTION[33].
La *vitesse* détermine l'augmentation du *degré-de-progression* en un *pas* pour une unité de *puissance* du PERFORMER affecté dans l'INSTRUCTION à la réalisation de l'OPERATION. A chaque *pas*, le *degré-de-progression* augmente donc en fonction de la *vitesse* de l'OPERATION et de la *puissance* du PEFORMER de l'INSTRUCTION. La VALEUR de *vitesse* est déclarée de manière statique ou bien calculée dynamiquement par invocation de la METHODE *calcul-vitesse*.
L'unité de mesure de la *vitesse* dépend du *mode-de-progression* :

- c'est un nombre d'ENTITES traitées, pour le mode `unit` (caractérisant la *particularisation* UNIT-GRANULATED OPERATION)
- c'est une quantité d'une certaine PROPRIETE numérique (par example une surface traitée), pour le mode `quantity` (*particularisation* QUANTITY-GRANULATED OPERATION) ; une référence à cette PROPRIETE numérique est la VALEUR de la PROPRIETE *descripteur-de-la-quantité* de l'opération
- c'est un pourcentage de réalisation de l'effet, pour le mode `time` (*particularisation* TIME-GRANULATED OPERATION)

La *condition-de-faisabilité* est un PREDICAT sur l'ETAT du CONTROLLED SYSTEM. En général, il porte plus précisément sur l'ETAT des *entités-ressources* des OPERATED OBJECTS de l'INSTRUCTION. Il doit renvoyer *true* pour que la *procédure-transition-état* puisse être invoquée (plus précisément pour qu'une INSTRUCTION incluant l'OPERATION puisse figurer dans un PACK OF EXECUTABLE INSTRUCTIONS). On peut aussi invoquer la *condition-de-faisabilité* de l'OPERATION dans la *condition-ouverture* de la PRIMITIVE ACTIVITY SPECIFICATION qu'elle définit.
La *situation* de l'OPERATION prend une des VALEURS suivantes : `dormant`, `executing`, `suspended`, `terminated`, `cancelled`. La *situation* est `dormant` lors de l'INSTANCIATION de l'OPERATION. La *situation* passe à `executing` à la première INVOCATION de la *procédure-transition-état*, c'est-à-dire dès lors que l'INSTRUCTION qui met en jeu l'OPERATION est incluse dans un PACK OF EXECUTABLE INSTRUCTIONS choisi par les PREFERENCE RULES. La *situation* de l'OPERATION peut passer à `suspended` par IMMOBILISATION d'une RESOURCE, ce qui justifie un nouvel établissement d'un PACK OF EXECUTABLE INSTRUCTIONS. Dans ce cas, toutes les OPERATIONS en cours sont `suspended`. Certaines d'entre elles pourront être reprises immédiatement (continuées) si elle figurent à nouveau dans le PACK choisi (leur *situation* redeviendra `executing`), et si leur *condition-de-faisabilité* est toujours vérifiée. Les autres resteront `suspended` jusqu'à ce que les conditions de leur reprise soient réunies. L'OPERATION est `terminated` lorsqu'elle est réalisée en totalité, ce qui se définit par un *degré-de-progression* à VALEUR *1*. Ceci entraîne que l'INSTRUCTION devient `closed`, si le passage est valide (*cf.* section 5.4.2.2). Une OPERATION est `cancelled` dès que l'INSTRUCTION en jeu est `cancelled`.
Les *date-début* et *date-fin* sont les INSTANTS d'apparition des *situations* `executing` et `terminated` respectivement. Elles déterminent la *durée-brute* de l'OPERATION. La *durée* est la *durée-brute* réduite des durées des intervalles pendant lesquels la *situation* est `suspended`[34].
La VALEUR de *ressources-requises* est une liste d'OPERATION RESOURCE SPECIFICATIONS. La PROPRIETE *ressource-usage* mesure à chaque INSTANT la diminution du *degré-de-disponibilité* des *ressources-requises* provoquée par la réalisation de l'OPERATION.
Le *seuil-suspension* a une VALEUR constante comprise entre *0* et *1*. Si et seulement si le *degré-de-progression* est supérieur au *seuil-suspension*, la continuation de l'OPERATION est requise lors d'un nouvel établissement d'un PACK OF EXECUTABLE INSTRUCTIONS (on verra en section 5.5 que la PRIMITIVE ACTIVITY SPECIFICATION concernée est assortie de la notation *req*).
Le *seuil-fermeture* a une VALEUR constante comprise entre *0* et *1*. Si et seulement si le *degré-de-progression* est supérieur au *seuil-fermeture*, la fermeture de l'activité encapsulant l'OPERATION est autorisée lors du mécanisme de propagation des changements de *situation*. Ce seuil est *1* pour les *particularisations* QUANTITY-GRANULATED OPERATION et TIME-GRANULATED OPERATION, parce que l'effet de leur *procédure-transition-état* est réalisé complètement au

---

[33] Plus précisément sur l'ensemble des OPERATED OBJECTS résultant de l'EXPANSION de l'OPERATED OBJECT SPECIFICATION au début de la mise en œuvre de l'activité (et non sur l'ensemble résultant de la dernière EXPANSION dans le cas d'une OPERATION interrompue et reprise plusieurs fois). Encore plus précisément, il s'agit des ENTITES valeurs de la PROPRIETE *entité-ressource* des OPERATED OBJECTS.

[34] Noter que la *durée* est une conséquence de l'INVOCATION du mécanisme ACTING INSTRUCTION LIST, compte tenu de la taille de l'OPERATED OBJECT, de la *vitesse* de l'OPERATION et de la *puissance* du PERFORMER. Aucun mécanisme spécifique n'est dédié à l'induction de la *puissance* ou de la *vitesse* en fonction d'une *durée* posée *a priori*. On peut viser cet objectif par une rédaction appropriée du NOMINAL PLAN (en introduisant par une DISJUNCTION ACTIVITY SPECIFICATION un choix non déterministe sur l'OPERATION et/ou le PERFORMER) et des PREFERENCE RULES (en préférant, par l'exploitation du mécanisme SELECTING INSTRUCTION LIST, un paquet dans lequel les PROPRIETES de l'OPERATION et du PERFORMER respectent mieux la *durée* visée).

premier *pas* de la progression (voir ci-dessous). Si le *degré-de-progression* est nul, le fonctionnement vis-à-vis de ce seuil n'est pas activé. En effet, fermer une activité sans avoir démarré l'exécution de l'OPERATION revient à ne pas entreprendre l'activité, sans conséquence sur le déroulement du plan ; cette possibilité est déjà donnée par l'OPTIONAL ACTIVITY SPECIFICATION.
La *priorité-d-exécution* permet d'ordonnancer les événements gérant la mise en œuvre d'OPERATIONS devant intervenir au même INSTANT. C'est l'OPERATION de VALEUR de *priorité-d-exécution* la plus faible qui est mise en œuvre d'abord. Par défaut de spécification de la *priorité-d-exécution*, le *degré-de-priorité* des EVENEMENTS portant sur une OPERATION sera celle définie dans la section 5.8.

4.7.1 Classes d'opérations, fondées sur le mode de progression de l'effet

Une UNIT-GRANULATED OPERATION est une OPERATION telle que, à chaque `pas` de la progression de l'effet, c'est un certain nombre entier (*n*) d'ENTITES des OPERATED OBJECTS de l'INSTRUCTION qui changent d'ETAT. Ces ENTITES sont les INSTANCES de la CLASSE *classe-entité*, à quelque profondeur qu'elles se trouvent comme ELEMENTS dans les *entités-ressources* des OPERATED OBJECTS[35] [36]. La *vitesse* est l'entier *n*. La *procédure-transition-état* porte sur une seule des ENTITES. Elle est donc invoquée *n* fois à chaque *pas*. Le *degré-de-progression* est, à chaque INSTANT, la proportion des ENTITES sur lesquelles la *procédure-transition-état* a déjà été invoquée[37].

Une QUANTITY-GRANULATED OPERATION est une OPERATION telle que, à chaque `pas` de la progression de l'effet, c'est une certaine quantité de la valeur d'une certaine PROPRIETE numérique (valeur absolue *s*) des *entités-ressources* des OPERATED OBJECTS de l'INSTRUCTION qui change d'ETAT. Les *entités-ressources* des OPERATED OBJECTS doivent être des INSTANCES de la CLASSE *classe-entité* et posséder la PROPRIETE en question. La *vitesse* est *s*. La *procédure-transition-état* porte sur une seule des *entités-ressources* des OPERATED OBJECTS. Elle est invoquée une et une seule fois pour chaque *entité-ressource* : au début de l'exécution de l'OPERATION pour le premier OPERATED OBJECT de l'ensemble et, pour chacun des suivants, dès que l'OPERATED OBJECT qui le précède dans l'ensemble a été traitée sur la totalité de la valeur de la PROPRIETE (par exemple sur la totalité de sa surface). Bien noter que, dès l'INSTANT où la *procédure-transition-état* a été invoquée pour une *entité-ressource* des OPERATED OBJECT, les PROPRIETES définissant son ETAT sont changées et qu'elles ne changeront plus jusqu'à la réalisation complète de l'OPERATION : seule la PROPRIETE *degré-de-progression* de l'OPERATION sera modifiée à chaque *pas*. Le *degré-de-progression* est, à chaque INSTANT, la proportion déjà traitée de la somme des valeurs de la PROPRIETE des *entité-ressources* des OPERATED OBJECTS.
Une OPERATION sera une QUANTITY-GRANULATED OPERATION seulement lorsque l'évolution de l'ETAT des ENTITES qu'elle opère est supposé ne pas dépendre de son caractère réalisée ou non, pour autant que la durée de réalisation de l'OPERATION ne dépasse pas un seuil qui lui est spécifique. Autrement dit, si on identifiait deux parties, une sur laquelle l'OPERATION est effectivement réalisée et une autre sur laquelle elle ne l'est pas, alors ces deux parties évolueraient, tous PROCESSUS pris en compte, de la même manière jusqu'à l'atteinte du seuil en question.

Une TIME-GRANULATED OPERATION est une OPERATION telle que, à chaque `pas` de la progression de l'effet, c'est un certain pourcentage (*q*) du changement d'ETAT provoqué par la *procédure-transition-état* qui est réalisé. La *procédure-transition-état* porte sur une seule des *entités-ressources* des OPERATED OBJECTS, lesquelles doivent être des INSTANCES de la CLASSE *classe-entité*. Elle est invoquée une et une seule fois pour chaque OPERATED OBJECT, au début de l'exécution de l'OPERATION pour le premier OPERATED OBJECT de l'ensemble et, pour chacun des suivants, dès que l'OPERATED OBJECT qui le précède dans l'ensemble a subi la réalisation complète de l'effet.. Bien noter que, dès l'INSTANT où la *procédure-transition-état* a été invoquée pour une *entité-ressource* des OPERATED OBJECT, les PROPRIETES définissant l'ETAT de cette *entité-ressource* sont changées et qu'elles ne changeront plus jusqu'à la réalisation complète de l'OPERATION : seule la PROPRIETE *degré-de-progression* sera modifiée à chaque *pas*. Le *degré-de-progression* est, à chaque INSTANT, la

---

[35] On rappelle que l'OPERATED OBJECT SPECIFICATION de la PRIMITIVE ACTIVITY SPECIFICATION génère (par son DEVELOPPEMENT) un ensemble d'OPERATED OBJECTS, sur lesquels l'OPERATION porte conjointement.

[36] Ces ENTITES peuvent être directement des INSTANCES de *classe-entité*, si les OPERATED OBJECTS de l'INSTRUCTION sont générés par EXPANSION d'une SPECIFICATION D'ENSEMBLE D'ENTITES.

[37] Noter que toute opération sur le système piloté peut en principe être modélisée par l'une ou l'autre de ces trois *particularisation*s d'OPERATIONS. Considérons par exemple, la récolte des fruits des plantes dans une serre de tomates.

Avec une UNIT-GRANULATED OPERATION, la *procédure-transition-état* modifie le nombre et le poids total des fruits sur une plante. La *vitesse* est un nombre de plantes récoltées par *pas*. A chaque *pas*, les plantes récoltées changent d'ETAT et leur nombre augmente. Les plantes non encore récoltées n'ont pas encore changé d'ETAT.

Avec une QUANTITY-GRANULATED OPERATION dont la PROPRIETE en jeu serait la surface, la *procédure-transition-état* porte sur la serre entière et le modélisateur peut choisir de lui faire modifier le nombre et le poids total des fruits sur chaque plante de la serre. La *vitesse* est une surface récoltée par *pas*. Toutes les plantes de la serre changent d'ETAT au début de l'exécution de l'OPERATION.

Avec une TIME-GRANULATED OPERATION, la *procédure-transition-état* porte aussi sur la serre entière et le modélisateur peut choisir de lui faire modifier le nombre et le poids total des fruits sur chaque plante de la serre. La *vitesse* est un pourcentage de réalisation de l'OPERATION par *pas*. Toutes les plantes de la serre changent d'ETAT au début de l'exécution de l'OPERATION.

valeur *q* multipliée par le nombre d'INVOCATIONS de la *procédure-transition-état* depuis le début de l'exécution sur le premier OPERATED OBJECT.

Le mécanisme ACTING INSTRUCTION LIST provoque la mise en œuvre en parallèle des OPERATIONS sur le CONTROLLED SYSTEM. Il est activé dans le mécanisme MAKING INSTRUCTION LIST, après que celui-ci a établi et choisi un PACK OF EXECUTABLE INSTRUCTIONS. L'AGENDA du SYSTEME ne peut contenir qu'une INVOCATION de ce mécanisme. A l'INSTANT de l'activation, on considère les OPERATIONS en jeu dans toutes les INSTRUCTIONS du PACK. Pour chaque OPERATION, on initie éventuellement puis on poursuit sans délai un PROCESSUS CONTINU spécifié à partir de la description de l'OPERATION, notamment la VALEUR de son *pas*, du contenu de *procédure-transition-état* et du *degré-de-progression*. Ces PROCESSUS sont entretenus, pour chacun d'eux, jusqu'à la réalisation complète de l'OPERATION (*situation* `terminated`). Ils sont tous arrêtés au moment d'OCCURRENCE d'un EVENEMENT déclenchant le mécanisme MAKING INSTRUCTION LIST (et par conséquent un nouvel ACTING INSTRUCTION LIST). Ceux correspondant à une OPERATION non complètement terminée (*situation* `suspended`) pourront être poursuivis (sans renouveler l'initialisation) à partir du *degré-de-progression* à l'INSTANT de la suspension.

4.7.2 Opérations permanentes et opérations ponctuelles

Une opération permanente est une opération dont l'INSTANT de terminaison n'est pas spécifié à l'aide de ses propres ATTRIBUTS (*date-début*, *vitesse* et *pas*) et des PROPRIETES de l'objet opéré (nombre d'unités, surface). Son effet est réalisé tant que ne vient pas l'arrêter un événement extérieur. Cet événement peut être le dépassement de la fenêtre de fin de l'activité primitive qui sous-tend l'OPERATION. Ou encore la fermeture de cette activité primitive par propagation à partir d'événements sur d'autres activités liées. Par exemple, Si *b* est une activité primitive qui sous-tend une opération permanente, et si le NOMINAL PLAN est *coend*(*a*, *b*), l'effet de l'opération permanente est arrêté dès que *a* est fermée.
La déclaration du caractère permanent d'une OPERATION se fait en donnant à la *vitesse* une VALEUR positive infiniment petite, de telle sorte que le *degré-de-progression* reste toujours proche de *0* [38]. Ceci assure que la VALEUR de *1* ne sera jamais atteinte, laquelle seule provoque par construction l'arrêt de la progression de l'effet.
Seules les QUANTITY-GRANULATED OPERATIONS et les TIME-GRANULATED OPERATIONS peuvent être déclarées permanentes. En effet, pour les UNIT-GRANULATED OPERATIONS, l'effet porte à chaque *pas* sur un certain nombre d'ENTITES, et l'OPERATION se termine nécessairement quand toutes les ENTITES, en nombre toujours fini, ont subi l'effet. Encore faut-il, pour les QUANTITY-GRANULATED OPERATIONS et les TIME-GRANULATED OPERATIONS, respecter la condition suivante : l'OPERATED OBJECT SPECIFICATION doit se développer en une liste d'un seul élément. S'il y en avait plusieurs, l'effet ne serait de toutes façons pas réalisé sur les éléments après le premier, puisque la quantité opérée (ou la proportion opérée) du premier n'atteint jamais *1*.
Pour les opérations permanentes, la *procédure-transition-état* est invoquée une et une seule fois à chaque *pas*.
Une alternative à cette représentation des opérations permanentes consiste à créer une activité primitive itérée. L'OPERATION en jeu serait une QUANTITY-GRANULATED OPERATION (resp. UNIT-GRANULATED OPERATION, TIME-GRANULATED OPERATION) dont la vitesse aurait pour VALEUR, par exemple, la surface totale de l'objet opéré (resp. le nombre total d'ENTITES de la *classe-entité*, *1*). La permanence de l'effet est alors assuré par le mécanisme d'ITERATION. Cette solution implique une procédure de calcul plus lourde.

Une opération ponctuelle est une OPERATION dont l'INSTANT de terminaison est le même que l'INSTANT de début d'exécution.
La déclaration du caractère ponctuel d'une OPERATION se fait en donnant à la *vitesse* une VALEUR qui assure que le *degré-de-progression* atteigne *1* dès la première invocation de la *procedure-transition-état*. Pour les QUANTITY-GRANULATED OPERATION (resp. UNIT-GRANULATED OPERATION, TIME-GRANULATED OPERATION) c'est la VALEUR de la PROPRIETE en jeu de l'objet opéré (resp. le nombre total d'ENTITES de la *classe-entité*, *1*).

4.7.3 Les opérations de transfert

Une opération de transfert est une OPERATION dont la *procédure-transition-état* gère le transfert d'un certain nombre d'ENTITES qui sont ELEMENTS de la source de l'activité vers sa destination, ou bien le transfert de tout ou partie de la VALEUR d'une PROPRIETE quantitative de la source de l'activité vers sa destination. Une opération de transfert ne peut être incluse que dans la spécification d'une activité de transfert (voir § 4.6).
Ce n'est pas le modélisateur qui détermine la manière dont est géré le transfert : ceci est prédéfini dans les *procédure-transition-état* caractéristiques des *particularisations* UNIT-GRANULATED TRANSFER OPERATION et QUANTITY-GRANULATED TRANSFER OPERATION.

---

[38] Donner la VALEUR *0* aurait aussi une conséquence permanente ... mais évidemment non désirée : la non réalisation de l'effet.

Par rapport aux OPERATIONS en général, une opération de transfert ne possède pas de PROPRIETES additionnelles spécifiques. La *vitesse* est le nombre d'ELEMENTS transférés ou la quantité transférée en un *pas*. Pour un transfert d'ELEMENTS (la PROPRIETE *identificateur* de la TRANSFER INFORMATION est alors `elements`), la *classe-entités* de l'opération doit être la CLASSE des ELEMENTS immédiats, et non la CLASSE des objets source ou destination. Pour un transfert spécifié par une PROPRIETE quantitative commune à la source et à la destination, la *classe-entité* doit être la CLASSE des objets de la source.
Une opération de transfert peut être ponctuelle mais ne peut pas être permanente (voir (§ 4.7.2).

5. Aspects fonctionnel et dynamique de la simulation d'un système de production

La simulation d'un système de production (PRODUCTION SYSTEM) consiste à reproduire l'évolution dans le temps de la partie de son état qui présente un intérêt dans la résolution d'un problème. Le problème que le simulateur contribue à solutionner est généralement une combinaison de la prédiction d'états futurs, du contrôle du système compte tenu d'un objectif de production, et du diagnostic d'écarts entre états attendus et observés. Cette section décrit les mécanismes généraux de cette simulation. La généralité s'entend au sens où ils s'appliquent sans changement pour des systèmes de production dans différents domaines. Autrement dit, chaque mécanisme opère sur des données dont la structure est toujours la même, mais dont les valeurs diffèrent non seulement d'un domaine à l'autre, mais encore d'une simulation à l'autre dans un domaine donné.
Les mécanismes peuvent être représentés par des PROCEDURES de calcul, chacune caractérisée par les structures de données sur lesquelles elle opère, et par un programme d'instructions à mettre en œuvre sur ces données. Le codage de ces PROCEDURES en un language de programmation permet alors d'automatiser le calcul pour des systèmes complexes, ou dont l'évolution est à reproduire sur une longue période ou à partir d'une large gamme de jeux de données.
Un mécanisme (et la PROCEDURE associée) peut sous-traiter une partie de son rôle à un autre. Deux mécanismes sous-traitants exclusifs d'un troisième sont dits agrégés par celui-ci. Un mécanisme, dit client, peut demander à un autre, dit fournisseur, de lui élaborer et lui renvoyer une donnée nécessaire à son propre fonctionnement.
Les PROCEDURES générales, applicables pour la simulation dans différents domaines, seront qualifiées de standard, par opposition à celles qui décrivent tel ou tel aspect fonctionnel de la réalité dans un domaine. On qualifiera aussi de standards les PROCESSUS et les EVENEMENTS qu'elles participent à spécifier.

5.1 Le mécanisme de simulation

Les PROCEDURES codant les mécanismes sont une partie de la connaissance sur le domaine objet de la présente ontologie. A ce titre, elles sont liées aux concepts de l'ontologie. Un lien naturel consiste à les installer comme METHODES des CLASSES représentant les concepts. Plus précisément, la VALEUR d'une METHODE est la référence à une PROCEDURE, ce qui peut être par exemple son nom ou bien l'adresse de l'implantation de son code en mémoire. Il existe nécessairement une PROCEDURE de démarrage, dénommée plus loin *run-simulation*, qui invoque directement et indirectement l'ensemble des mécanismes de la simulation. Elle est placée au niveau des concepts de base de l'ontologie, et non au niveau de l'extension aux SYSTEMES. En effet, une simulation peut s'opérer sur une réalité modélisée sans utiliser les concepts de cette extension (*cf.* introduction de la section 3). C'est la CLASSE SIMULATION qui détient cette PROCEDURE.

La CLASSE SIMULATION permet de spécifier les conditions dans lesquelles on va invoquer les mécanismes de la simulation. Certaines de ces conditions sont décrites dans des ATTRIBUTS :

- l'ATTRIBUT *entité-simulée* a pour VALEUR une référence sur une ENTITE, dénommée l' « entité simulée » ; elle est choisie de telle sorte que toutes les PROCEDURES en jeu dans la simulation puissent référencer à partir d'elle toutes les ENTITES sur lesquelles elles opèrent ; normalement, il existe une ENTITE qui inclut toutes les autres par les RELATIONS de *composition* et *ensembliste* ou qui entretient avec elles des RELATIONS fonctionnelles représentées par des METHODES : cette ENTITE est alors normalement la VALEUR de *entité-simulée*. Cette ENTITE doit posséder une PROPRIETE *agenda-événements*, à laquelle on donnera pour VALEUR une liste d'EVENEMENTS totalement ordonnés selon la règle codée dans la METHODE *before-in-agenda-events*.
- les VALEURS de deux autres ATTRIBUTS (*instant-début*, *instant-fin*) délimitent par des INSTANTS l'intervalle de temps pendant lequel l'évolution de l'entité simulée va être reproduite ;

La METHODE *before-in-agenda-events* est un PREDICAT qui opère sur deux EVENEMENTS et qui renvoie *true* si le premier doit être placé avant l'autre dans la liste VALEUR de l'ATTRIBUT *agenda-événem*ents de l'entité simulée. Ce PREDICAT doit renvoyer *true* si, mais pas seulement si, l'INSTANT d'OCCURRENCE du premier est strictement antérieur à l'INSTANT d'OCCURRENCE du second.
L'ATTRIBUT *horloge* a une VALEUR qui évolue de façon monotone entre le *time-point* de l'*instant-début* et le *time-point* de l'*instant-fin*.
La METHODE *run-simulation* provoque l'OCCURRENCE des EVENEMENTS programmés.

La simulation d'une réalité consiste à invoquer le mécanisme de SIMULATION. Il consiste :

S1) à créer les INSTANCES d'ENTITES et de PROCESSUS qui représentent les aspects structurel et fonctionnel de la réalité ; on place les ENTITES dans les ETATS qu'ils doivent avoir au début de la période simulée ; on choisit parmi elles l'entité simulée ; on réalise éventuellement l'INSTALLATION de SPECIFICATIONS D'INVOCATION REFLEXE.
S2) à créer une INSTANCE de SIMULATION, en valuant les attributs *entité-simulée*, *instant-début* et *instant-fin* ; cette INSTANCE est dénommée la « simulation courante » ; l'*horloge* est valué avec le *time-point* de l'*instant-début* ;

S3) à valuer la METHODE *before-in-agenda-events* avec la référence à un PREDICAT approprié ;
S4) à programmer dans l'*agenda-événements* de l'entité simulée ou celui d'un de ses COMPOSANTS ou ELEMENTS (mécanisme de PROGRAMMATION), les EVENEMENTS qui, directement ou indirectement, spécifient des directives de contrôle sur des PROCESSUS qui font évoluer l'ETAT des ENTITES ; ils représentent l'aspect dynamique de la réalité ;
S5) à invoquer la METHODE *run-simulation*.

Le mécanisme de PROGRAMMATION d'un EVENEMENT `e` dans l'AGENDA d'une ENTITE opère de la manière suivante. Si la liste `L` VALEUR de l'ATTRIBUT *agenda-événements* de l'ENTITE est vide, `e` y est placé. Sinon, le PREDICAT *before-in-agenda-events* de la simulation courante est invoqué sur un couple d'EVENEMENTS, composé de l'EVENEMENT `e` et, itérativement, le premier EVENEMENT non encore examiné (nommé ici `next`) de `L`. Si le PREDICAT renvoie *true*, `e` est placé juste avant `next` dans `L` et l'itération s'arrête. Sinon l'itération continue. Si tous les EVENEMENTS de `L` ont été examinés sans que *before-in-agenda-events* n'ait jamais renvoyé *true*, alors `e` est placé en fin de `L`. `L` devient la nouvelle VALEUR de l'ATTRIBUT *agenda-événements* de l'ENTITE.

La METHODE *run-simulation* itère l'action suivante. Soit `L` la liste VALEUR de l'ATTRIBUT *agenda-événements* de l'entité simulée[39]. Tant que la liste `L` n'est pas vide, le *time-point* de l'INSTANT auquel est programmé le premier EVENEMENT dans `L` devient la VALEUR de l'*horloge*, et la METHODE *déclenche-événements* de SIMULATION est invoquée.

La METHODE *déclenche-événements* de la simulation courante, avec l'*horloge* en argument, enlève successivement de l'AGENDA tous les EVENEMENTS programmés en un INSTANT dont le *time-point* égale l'*horloge*. Lors de chaque enlèvement et avant le suivant éventuel, le mécanisme d'OCCURRENCE de l'EVENEMENT enlevé est déclenché. Celui-ci peut programmer de nouveaux EVENEMENTS, possiblement en tête de l'AGENDA. Dans ce cas, c'est le nouvel EVENEMENT de tête qui fait l'objet de la prochaine OCCURRENCE.

Synchronisation des processus

Lorsque les EVENEMENTS programmés émettent des directives sur des PROCESSUS CONTINUS, le mécanisme de SIMULATION assure leur synchronisation.
La synchronisation d'un ensemble de PROCESSUS CONTINUS consiste à assurer qu'à tout INSTANT courant, et pour chaque PROCESSUS, la dernière mise à jour de l'ETAT des ENTITES sur lesquelles il porte date de moins de la VALEUR du *pas* du PROCESSUS.
Considérons par exemple les PROCESSUS *Pa* et *Pb*, de *pas 1* et *2*, respectivement, et initiés en l'INSTANT *0*. La synchronisation n'est pas assurée en l'INSTANT *3* si les ETATS gérés par *Pa* et *Pb* sont ceux établis en l'INSTANT *2* : l'ETAT géré par *Pa* doit être mis à jour en l'INSTANT *3*.
La synchronisation est assurée de la manière suivante. Lors de l'OCCURRENCE d'un EVENEMENT qui émet sur un PROCESSUS CONTINU une directive correspondant à sa METHODE *procédure-poursuite*, le mécanisme de SIMULATION commence à enchaîner des INVOCATIONS de *procédure-poursuite*. A chacune de ces INVOCATIONS, l'effet du PROCESSUS est réalisé ; puis l'*horloge* de la simulation courante progresse de la VALEUR du *pas* du PROCESSUS, sauf si cela aurait pour effet de réaliser la prochaine INVOCATION en un INSTANT `t` postérieur à l'INSTANT d'OCCURRENCE du premier EVENEMENT dans l'AGENDA. Dans ce cas, un nouvel EVENEMENT est programmé (avec une directive *procédure-poursuite* sur le même PROCESSUS) en l'INSTANT `t`. L'INSTANT `t` est postérieur de la VALEUR du *pas* du PROCESSUS à l'INSTANT de la dernière INVOCATION.

5.2 Simulation d'un système piloté

La simulation d'un système piloté ne présente qu'une spécificité par rapport au mécanisme général : l'entité simulée est une INSTANCE de PRODUCTION SYSTEM. La liste VALEUR de l'ATTRIBUT *agenda-événements* de l'entité simulée est constituée des EVENEMENTS des AGENDAS du MANAGER, de l'OPERATING SYSTEM et du CONTROLLED SYSTEM, exclusivement.

Sauf à considérer le cas particulier d'un PRODUCTION SYSTEM sur lequel on prévoit de ne faire aucune OPERATION, le mécanisme de SIMULATION impose de programmer dans l'*agenda-événements* (phase S4) au moins un EVENEMENT invoquant les mécanismes UPDATING SITUATION et MAKING INSTRUCTION LIST. Sauf si la REACTIVE TRAJECTORY du MANAGER est vide, on programme aussi un EVENEMENT invoquant le mécanisme CHECKING REACTIVE TRAJECTORY.

[39] A défaut d'avoir programmé des EVENEMENTS dans cet AGENDA, la liste L est constituée des EVENEMENTS programmés dans les AGENDAS des COMPOSANTS et ELEMENTS de l'entité simulée, à quelque profondeur que ce soit.

Les EVENEMENTS contrôlant les PROCESSUS de l'OPERATING SYSTEM (mécanismes MAKING INSTRUCTION LIST et ACTING INSTRUCTION LIST) sont normalement la conséquence de l'OCCURRENCE des EVENEMENTS programmés dans l'AGENDA du MANAGER.

Les EVENEMENTS contrôlant les PROCESSUS portant sur le CONTROLLED SYSTEM peuvent être programmés lors de la phase S4, ou bien comme conséquence de l'OCCURRENCE des EVENEMENTS programmés dans l'AGENDA du MANAGER ou de l'OPERATING SYSTEM (typiquement, être la conséquence d'une OPERATION sur le CONTROLLED SYSTEM).
Les PROCESSUS en jeu dans l'évolution du CONTROLLED SYSTEM sont décrits dans la spécialisation de l'ontologie sur le domaine d'application étudié[40]. *A contrario*, ils ne sont aucunement prédéfinis dans l'ontologie des systèmes de production en général. Il en est de même pour les EVENEMENTS qui leur appliquent les directives correspondant aux METHODES *procédure-exécution*, *procédure-initialisation*, *procédure-poursuite* et *procédure-arrêt* de ces PROCESSUS. Le mécanisme général de SIMULATION d'un système piloté se ramène donc à la description des mécanismes invoqués par les EVENEMENTS programmés dans les AGENDAS du MANAGER et de l'OPERATING SYSTEM. C'est l'objet des sous-sections suivantes.

Une fois décrits les différents mécanismes de la simulation d'un système piloté (sections 5.3 à 5.7), on précise (dans la section 5.8) leur ordonnancement relatif dans le cas particulier où les OCCURRENCES d'au moins deux EVENEMENTS sont programmées au même INSTANT.

Un schéma général du mécanisme de SIMULATION d'un système piloté figure en annexe 6, qui positionne les différents sous-mécanismes, EVENEMENTS, PROCESSUS et INVOCATIONS des principales METHODES qui seront développés dans la suite de cette section. L'annexe 7 présente dans le même esprit les mécanismes gérés par les EVENEMENTS dans l'AGENDA du MANAGER.

Le gestionnaire d'information

On précise ici les mécanismes d'échange d'information entre le SYSTEME et son ENVIRONNEMENT. Ils sont associés aux TABLES D'IMPORTATION et D'EXPORTATION du SYSTEME. Ces ENTITES déterminent en effet la nature des informations échangeables, et détiennent les METHODES qui réalisent ces échanges, à la demande du SYSTEME lui-même ou d'ENTITES dans son ENVIRONNEMENT.

TABLE D'IMPORTATION de S	
	indicateur 1 de S
	....
	indicateur n de S
	importation(INDICATEUR)

fig. 5.2a

TABLE D'EXPORTATION de S'	
	aspect-visible 1 de S'
	....
	aspect-visible n de S'
	exportation(ASPECT VISIBLE)

fig. 5.2c

INDICATEUR de S	
	S
	aspect-visible 1 de S'
	....
	aspect-visible n de S'
	valeur
	fonction-importation()
	get-valeur()

fig. 5.2b

ASPECT VISIBLE de S'	
	S'
systèmes autorisés	{S1, ..., Sn}
	condition-exportation()
	est-demandable-par(ENTITE)
	valeur
	fonction-état()
	get-valeur()

fig. 5.2d

Un SYSTEME S ne peut s'informer sur son ENVIRONNEMENT que *via* sa propre TABLE D'IMPORTATION. Celle-ci comporte une liste d'INDICATEURS (fig. 5.2a), qui définit l'ensemble des informations que le SYSTEME peut obtenir de son ENVIRONNEMENT. Un INDICATEUR de S repose sur des ASPECTS VISIBLES d'autres SYSTEMES que S (fig. 5.2b), tous collectivement appelés ici S'. Ces ASPECTS VISIBLES sont les ELEMENTS de la TABLE D'EXPORTATION de S' (fig. 5.2c). Un ASPECT VISIBLE de S' est défini par une FONCTION *fonction-état* (fig. 5.2d) qui renvoie, par exemple à S, une perception de S', si la *condition-exportation* est satisfaite. S ne peut recevoir cette information que s'il figure dans la liste des SYSTEMES autorisés (*est-demandable-par*(S) renvoie alors *true*).

[40] Lire une spécialisation au domaine de la production de tomates sous serre dans le document complémentaire cité en section d'introduction.

La valeur d'un INDICATEUR est obtenue en invoquant la METHODE *get-valeur* de l'INDICATEUR correspondant. C'est le mécanisme d'IMPORTATION, qui consiste seulement à invoquer la METHODE *importation* de la TABLE D'IMPORTATION de S, avec l'INDICATEUR (ici pour exemple à valeur entière) souhaité en argument.

```
int INDICATEUR::get-valeur() {
    SYSTEME S = le système dont this[41] est un INDICATEUR ;
    IMPORTATION de l' INDICATEUR this du système S ::
        S . get-table-importation() . importation(this) ;
}
```

La METHODE *importation* de la TABLE D'IMPORTATION de S, appliquée à un INDICATEUR `indic`, accède d'abord aux ASPECTS VISIBLES (d'autres SYSTEMES S') qui définissent `indic`. Puis, la METHODE *fonction-importation* de `indic` affecte une valeur à l'INDICATEUR (en fait à la PROPRIETE *valeur* de `indic`), à partir des *valeurs* des ASPECTS VISIBLES maintenant connues (et accessibles par leur METHODE *get-valeur*).

```
void TABLE D'IMPORTATION::importation(INDICATEUR indic) {
    SYSTEME S = le système dont indic est un INDICATEUR
    POUR TOUT ASPECT VISIBLE av DANS indic FAIRE
        EXPORTATION de av à la demande de S
    indic . valeur = indic . fonction-importation() ;
}
```

Pour un SYSTEME S, accéder à un ASPECT VISIBLE de S', c'est invoquer le mécanisme d'EXPORTATION qui consiste seulement à invoquer la METHODE *exportation* de la TABLE D'EXPORTATION de S', avec l'ASPECT VISIBLE souhaité en argument.

```
EXPORTATION d'un ASPECT VISIBLE av de S' à la demande de S ::
    S' . get-table-exportation() . exportation(av, S)
```

La METHODE *exportation* de la TABLE D'EXPORTATION de S', appliquée à un ASPECT VISIBLE `av`, affecte une valeur à `av` (précisément à la PROPRIETE *valeur* de `av`). La FONCTION *fonction-état* réalise cette évaluation à partir de l'ETAT de S', plus précisément à partir d'un ensemble de PROPRIETES et selon un mode d'évaluation qui donne son sens à l'ASPECT VISIBLE.

```
void TABLE D'EXPORTATION::exportation(ASPECT VISIBLE av, SYSTEME s) {
    SI av . est-demandable-par(s) ALORS
        SI av . condition-exportation () = true ALORS
            av . valeur = av . fonction-état() ;
}
```

5.3 Le mécanisme de révision de la stratégie

La révision de la stratégie est provoquée par l'OCCURRENCE d'un EVENEMENT standard : la *particularisation* CHECK REACTIVE TRAJECTORY EVENT, caractérisée par le couple {CHECK REACTIVE TRAJECTORY PROCESS, *procédure-exécution*}. CHECK REACTIVE TRAJECTORY PROCESS est une *particularisation* standard de PROCESSUS PONCTUEL portant sur un MANAGER. Le programme de sa METHODE *procédure-exécution* consiste simplement à invoquer la METHODE *check-reactive-trajectory* de ce MANAGER. La PROPRIETE *rythme-de-révision* de la STRATEGY du MANAGER peut être utilisée pour la PROGRAMMATION des CHECK REACTIVE TRAJECTORY EVENTS.

Le programme de *check-reactive-trajectory* est exactement celui du mécanisme CHECKING REACTIVE TRAJECTORY. Il considère chaque CONDITIONAL ADJUSTMENT dans l'ordre de leur insertion dans la REACTIVE TRAJECTORY, et réalise sur

[41] De façon générale, this désigne l'INSTANCE dont on est en train de décrire une METHODE ou une PROPRIETE, commune à toutes les INSTANCES de sa CLASSE.

lui l'INVOCATION des PREDICATS *triggering-landmark* et *state-predicate*, et, si les deux ont renvoyé *true*, l'INVOCATION de la PROCEDURE *strategy-adaptation*.

```
CHECK REACTIVE TRAJECTORY EVENT
→ CHECK REACTIVE TRAJECTORY PROCESS. procédure-exécution {
    MANAGER. check-reactive-trajectory() {
        pour tout ajustement dans MANAGER.strategy.reactive-trajectory :
            si   ajustement. triggering-landmark et
                 ajustement. state-predicate
            alors   ajustement. strategy-adaptation } }
```

Il faut noter que l'ordre d'insertion des CONDITIONAL ADJUSTMENTS dans la REACTIVE TRAJECTORY n'est pas neutre : chaque CONDITIONAL ADJUSTMENT opère sur une STRATEGY qui peut avoir été modifiée par le traitement des CONDITIONAL ADJUSTMENTS insérés avant lui dans la REACTIVE TRAJECTORY.

5.4 Les mécanismes d'ouverture et de fermeture des activités

L'ouverture et la fermeture des activités sont opérées par deux ensembles de PROCEDURES. Un des ensembles est organisé en un mécanisme (UPDATING SITUATION) dont l'INVOCATION INTENTIONNELLE est programmée dans la STRATEGY du MANAGER (à cet effet on peut notamment utiliser la PROPRIETE *rythme-de-lecture*). Les PROCEDURES de l'autre ensemble sont l'objet d'une INVOCATION REFLEXE, soit dans le mécanisme UPDATING SITUATION (il s'agit alors du mécanisme de PROPAGATION), soit en réaction à la fin d'exécution d'une OPERATION.

5.4.1 Ouverture et fermeture intentionnelle

L'examen intentionnel des conditions d'ouverture et de fermeture des activités est provoqué par l'OCCURRENCE d'un EVENEMENT standard : la *particularisation* UPDATE SITUATION EVENT, caractérisée par le couple {UPDATE SITUATION PROCESS, *procédure-exécution*}. UPDATE SITUATION PROCESS est une *particularisation* standard de PROCESSUS PONCTUEL. Le programme de sa METHODE *procédure-exécution* consiste simplement à invoquer la METHODE *update-situation* du MANAGER.

Le programme de *update-situation* du MANAGER réalise successivement deux tâches :

- exécuter le mécanisme UPDATING SITUATION lequel invoque la METHODE *update-status* sur le NOMINAL PLAN (en fait une INSTANCE de ACTIVITY SPECIFICATION) de chaque ACTIVITIES-RESOURCES BLOCK de la STRATEGY.
- la PROGRAMMATION en l'INSTANT courant d'un EVENEMENT invoquant le mécanisme MAKING INSTRUCTION LIST.

```
UPDATE SITUATION EVENT
→ UPDATE SITUATION PROCESS. procédure-exécution {
    MANAGER. update-situation() {
        pour tout block dans MANAGER.strategy :
            UPDATING SITUATION (block.nominal-plan) {
                nominal-plan.update-status() ;
                PROGRAMMATION MAKING INSTRUCTION LIST EVENT (block) ; }
        }
    }
```

La METHODE *update-status* d'ACTIVITY SPECIFICATION examine les conditions d'ouverture et de fermeture des activités. Elle est d'abord invoquée sur l'activité racine du NOMINAL PLAN. Si l'invocation de *update-status* sur une activité n'a pas abouti à l'annulation de cette activité (i.e. la *situation* diffère de `cancelled`) la METHODE est alors invoquée sur ses activités composantes, et ainsi de suite jusqu'aux PRIMITIVE ACTIVITY SPECIFICATIONS.
Pour une activité, la condition d'ouverture (ou de fermeture) est définie par deux sortes d'éléments :

- l'existence et, le cas échéant, le contenu du PREDICAT *condition-ouverture* (resp. *condition-fermeture*),
- la VALEUR des PROPRIETES *début-au-plus-tôt* et *début-au-plus-tard* (resp. *fin-au-plus-tôt* et *fin-au-plus-tard*).

En fait *update-status* n'agit pas directement sur la *situation* des activités qui ne sont ni PRIMITIVE ACTIVITY SPECIFICATION ni ITERATION ACTIVITY SPECIFICATION. Les activités ne peuvent changer de *situation* que par PROPAGATION à partir de changements de *situation* des PRIMITIVE ACTIVITY SPECIFICATIONS et des ITERATION ACTIVITY SPECIFICATIONS qui en sont composantes à une profondeur quelconque.

5.4.1.1 Cycle de vie d'une activité

5.4.1.1.1 Conditions de terminaison

Si l'activité sur laquelle porte *update-status* est `sleeping`, `closed` ou `cancelled`, *update-status* ne fait rien sur cette activité.
Si l'activité est `waiting`, la condition d'ouverture est testée (voir le paragraphe suivant). Si elle n'est plus jamais satisfiable, au sens où elle ne pourra plus se voir satisfaite, la mise en œuvre de l'activité n'est plus possible, l'exécution du NOMINAL PLAN est arrêtée, sauf s'il s'agit d'une activité optionnelle (composante unique d'une OPTIONAL ACTIVITY SPECIFICATION), et sauf s'il s'agit d'une activité optionnelle contrainte (composante unique d'une CONSTRAINED OPTIONAL ACTIVITY SPECIFICATION) dans laquelle *nombre-de-réalisations* est inférieur à n_1.
Si l'activité est `open` et si le *début-au-plus-tard* est atteint alors qu'une des OPERATIONS en jeu n'a pas démarré alors qu'elle aurait dû, la mise en œuvre de l'activité n'est plus possible, l'exécution du NOMINAL PLAN est arrêtée.
La contrainte de démarrage d'une OPERATION est spécifique à chaque *particularisation* d'activité. Pour les PRIMITIVE ACTIVITY SPECIFICATIONS, c'est l'OPERATION en jeu qui doit avoir démarré. Pour les *particularisations* pour lesquelles on peut identifier l'unique composante qui doit être ouverte avant les autres, la contrainte est satisfaite si elle l'est sur cette composante. Pour les *particularisations* telles que toutes les composantes doivent être ouvertes en même temps, la contrainte est vérifiée si elle l'est pour toutes les composantes. Pour les autres les *particularisations*, la contrainte est vérifiée si elle l'est pour au moins une composante.
Si l'activité, quel que soit son type, est `open`, la condition de fermeture est testée. Si elle n'est plus jamais satisfiable, au sens où elle ne pourra plus se voir satisfaite, la mise en œuvre complète de l'activité n'est plus possible, l'exécution du NOMINAL PLAN est arrêtée.

5.4.1.1.2 Corps du cycle de vie

Dans le cas général, c'est-à-dire si l'activité n'est ni une PRIMITIVE ACTIVITY SPECIFICATION ni une ITERATION ACTIVITY SPECIFICATION, *update-status* est immédiatement invoquée sur les composantes. Le changement de *situation* sera alors éventuellement réalisé par propagation du changement sur une composante.
Si l'activité est une PRIMITIVE ACTIVITY SPECIFICATION, et si la condition d'ouverture est satisfaite, la METHODE *update-status* invoque la METHODE *validate-open* (voir plus loin) pour un éventuel passage de la *situation* à `open`. Si l'activité est effectivement `open`, soit elle est candidate pour l'exécution de son OPERATION, soit cette OPERATION est déjà en cours d'exécution. Dans les deux cas, c'est la terminaison de cette OPERATION qui provoquera la fermeture de l'activité primitive, moment auquel la condition de fermeture devra être satisfaite.
Si l'activité est une ITERATION ACTIVITY SPECIFICATION, et si la condition d'ouverture est satisfaite, *update-status* invoque *validate-open* pour un éventuel passage de la *situation* à `open`. Si l'activité est effectivement `open`, *update-status* examine son unique composante. Si la *situation* de cette composante est `sleeping`, la *situation* de l'activité itérative passe à `closed` sans autre condition. Si cette *situation* est au contraire `waiting`, l'activité itérative passe à `closed` seulement si *validate-closed* (voir plus loin) renvoie *vrai* (sinon une nouvelle itération pourra avoir lieu).

5.4.1.2 Conditions locales d'ouverture et de fermeture

Le test des conditions d'ouverture et de fermeture d'une activité est réalisé de la façon suivante. On rappelle d'abord que toute activité peut être dotée d'un PREDICAT *condition-ouverture* et qu'elle est dotée de VALEURS pour *début-au-plus-tôt* et *début-au-plus-tard*, par un des moyens suivants :

- les valeurs sont attribuées à cette activité même lors de la spécification du NOMINAL PLAN,
- les valeurs sont calculées à partir de valeurs affectées à des activités liées et en fonction du sens véhiculé par les opérateurs (voir annexe 8),
- les valeurs sont affectées par PROPAGATION par *turn-to-open* ou *turn-to-closed* (*cf.* § 5.4.2.1),
- les valeurs sont attribuées par défaut n'ont pas encore été surchargées par un des moyens ci-dessus.

▪ Pour le test de la condition d'ouverture, on est à tout INSTANT dans un des trois cas suivants :
  - l'INSTANT courant est antérieur à *début-au-plus-tôt* : la condition d'ouverture est dite non encore satisfaite ; l'éventuel PREDICAT *condition-ouverture* n'est pas testé.
  - l'INSTANT courant est situé entre *début-au-plus-tôt* et *début-au-plus-tard* : si une *condition-ouverture* est spécifiée, la condition d'ouverture est satisfaite si le PREDICAT renvoie *true*, non satisfaite s'il renvoie *false*. Si aucune *condition-ouverture* n'est spécifiée, la condition d'ouverture est satisfaite.
  - l'INSTANT courant est postérieur à *début-au-plus-tard* : la condition d'ouverture est dite plus jamais satisfiable (voir conséquence au § 5.4.1.1.1) ; l'éventuel PREDICAT *condition-ouverture* n'est pas testé.

▪ Pour le test des conditions de fermeture, on est à tout INSTANT dans un des trois cas suivants :

- l'INSTANT courant est antérieur à *fin-au-plus-tôt* : même comportement que pour l'ouverture.
- l'INSTANT courant est situé entre *fin-au-plus-tôt* et *fin-au-plus-tard* : même comportement que pour l'ouverture.
- l'INSTANT courant est postérieur à *fin-au-plus-tard* : dans le cas général, elle est dite plus jamais satisfiable (voir conséquence au § 5.4.1.1). L'éventuel PREDICAT *condition-fermeture* n'est pas testé. Dans le cas particulier où toutes les OPERATIONS en cours d'exécution au titre de cette activité (ou de ses composantes) sont permanentes (voir § 4.7.2), la condition de fermeture est dite satisfaite.

Conjointement aux contraintes ci-dessus, une PRIMITIVE ACTIVITY SPECIFICATION ne peut être fermée que quand le *degré-de-progression* de l'OPERATION en jeu, quand elle n'est pas permanente, est devenu supérieur au *seuil-fermeture*.

5.4.1.3 Validité d'une ouverture et d'une fermeture vis-à-vis des activités liées

Les METHODES *validate-waiting*, *validate-open* et *validate-closed* examinent si le passage d'une activité aux *situation* `waiting`, `open` et `closed`, respectivement, respecte un certain nombre de contraintes. De façon générale, ces contraintes portent sur :

- la *situation* des composantes de l'activité, examinées par les METHODES *check-sons-if-waiting*, *check-sons-if-open* ou *check-sons-if-closed* ;
- la *situation* des activités dont elle est composante, examinées par les METHODES *check-if-son-waiting*, *check-if-son-open* ou *check-if-son-closed*.

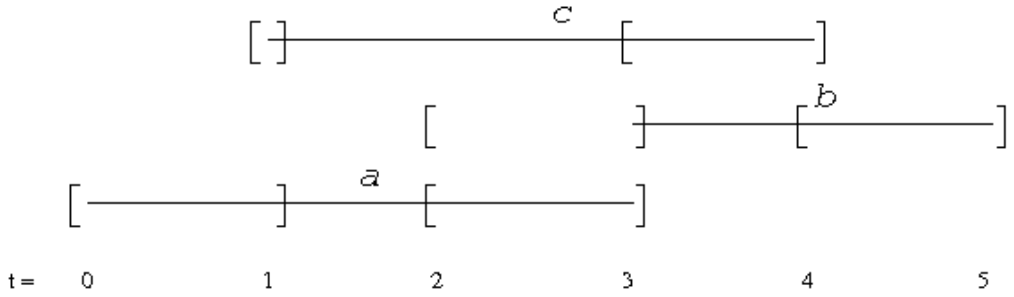
Les METHODES *validate-open* et *validate-closed* sont invoquées directement dans *update-status* pour les PRIMITIVE ACTIVITY SPECIFICATIONS et les ITERATION ACTIVITY SPECIFICATIONS (voir ci-dessus). Elles sont aussi invoquées (ainsi que *validate-waiting* dans ce cas) pour toutes les *particularisations* d'ACTIVITY SPECIFICATION à l'intérieur des METHODES *check-sons-if-?* et *check-if-son-?*. Ainsi, la validation d'un passage pour une activité *A* entraîne de proche en proche la validation de passages pour d'autres activités liées à *A*.

Les METHODES *validate-open* et *validate-closed* possèdent un argument qui exprime si la demande de validation d'un changement de *situation* d'une activité *A* provient d'un changement de *situation* d'une activité composante a_i (l'argument est alors *from_down*), ou bien d'une activité *B* dont elle est composante (l'argument est alors *from_up*). Un deuxième argument est justement l'activité a_i ou l'activité *B*, respectivement. Il est utilisé pour empêcher une nouvelle invocation sur a_i ou *B*, et ainsi une boucle de validations infinie. Cet argument est passé aux METHODES *turn-to-waiting*, *turn-to-open* et *turn-to-closed*, celles qui réalisent le changement de *situation* en cas de validation, et qui l'utilisent pour limiter la PROPAGATION aux seules activités qui le nécessitent.

Le comportement des METHODES *check-sons-if-waiting*, *check-sons-if-open* et *check-sons-if-closed* est donné en annexe 1, et celui des METHODES *check-if-son-waiting*, *check-if-son-open* et *check-if-son-closed* en annexe 2. Il est spécifique à chaque *particularisation* d'activité (voir aussi la section 4.5 –préconditions-).

5.4.1.4 Exemple développé

A titre d'exemple de mise en œuvre de l'ouverture/fermeture intentionnelle, et de clé de lecture des tableaux en annexes, soit le NOMINAL PLAN suivant : `P : and(overlap(`*c*, *b*), `meeting(`*a*, *b*)), figuré ci-dessous.



1. Avant le début de la simulation, l'activité racine `and` passe à `waiting` si `and` . *validate-waiting*(*from_up*, *null*) renvoie *vrai*, c'est-à-dire si (1.1) `overlap` . *validate-waiting*(*from_up*, `and`) renvoie *vrai*, ou si (1.2) `meeting` . *validate-waiting*(*from_up*, `and`) renvoie *vrai*.

`1.1 overlap` . *validate-waiting*(*from_up*, `and`) renvoie *vrai* si (1.1.1) *c* . *validate-waiting*(*from_up*, `overlap`) renvoie *vrai*.

`1.1.1` *c* . *validate-waiting*(*from_up*, `overlap`) renvoie *vrai* si *c* est `sleeping`, ce qui est vrai par construction, et si :

c . *check-sons-if-waiting()* renvoie *vrai*, ce qui est le cas puisque *c* n'a pas de composantes, et si

m_k . *check-if-son-waiting(c)* renvoie *vrai* pour toutes les activités m_k dont *c* est composante, excepté `overlap` puisque *from_up*, ce qui est le cas puisque *c* n'est la composante que de `overlap`.
Ainsi, le passage de `overlap` à `waiting` est validé.
La condition (1.2) est devenue neutre.
Ainsi, le passage de `and` à `waiting` est validé.

2. En t=0, et puisque `and` est `waiting`, l'INVOCATION intentionnelle de *update-status* cherche à valider le passage de `and` à `open` . `and` . *validate-open*(*from_up*,*null*) renvoie *vrai* si :
`(2.1)` `and` . *check-sons-if-open*() renvoie *vrai*, et si
m_k . *check-if-son-open*(`and`) renvoie *vrai* pour toutes les activités m_k dont `and` est composante, ce qui est le cas puisque `and` n'est la composante d'aucune activité.
2.1 `and` . *check-sons-if-open*() (annexe 1) renvoie *vrai* si (2.1.1) la *condition-ouverture* de `overlap` est satisfaite et si `overlap` . *validate-open*(*from_up*, `and`) renvoie *vrai*, ou si (2.1.2) la *condition-ouverture* de `meeting` est satisfaite et si `meeting` . *validate-open*(*from_up*, `and`) renvoie *vrai*.

2.1.1 `overlap` . *validate-open*(*from_up*, `and`) renvoie *vrai* si (2.1.1.1) la *condition-ouverture* de *c* est satisfaite et si *c* . *validate-open*(*from_up*, `overlap`) renvoie *vrai*.

2.1.1.1 La *condition-ouverture* de *c* n'est pas satisfaite, parce que l'INSTANT courant est antérieur à *début-au-plus-tôt*. Ainsi, l'ouverture de `overlap` n'est pas validée.

2.1.2 `meeting` . *validate-open*(*from_up*, `and`) renvoie *vrai* si (2.1.2.1) la *condition-ouverture* de *a* est satisfaite et si *a* . *validate-open*(*from_up*, `overlap`) renvoie *vrai*.

2.1.2.1 On suppose satisfaite la *condition-ouverture* de *a*.
a . *validate-open*(*from_up*, `meeting`) renvoie *vrai* si :
a . *check-sons-if-open*() renvoie *vrai*, ce qui est le cas puisque *a* n'a pas de composantes, et si
m_k . *check-if-son-open*(*a*) renvoie *vrai* pour toutes les activités m_k dont *a* est composante, excepté `meeting` puisque *from_up*, ce qui est le cas puisque *a* n'est la composante que de `meeting`.
Ainsi, le passage de *a* à `open` est validé.
Ainsi encore, le passage de `meeting` à `open` est validé.
Ainsi enfin, le passage de `and` à `open` est validé.

3. En t=1, la prochaine INVOCATION intentionnelle de *update-status* va chercher à valider le passage de `overlap` à `open` : il faut pour cela que la *condition-ouverture* de `overlap` soit satisfaite et que (3.1) `overlap` . *validate-open*(*from_up*, `and`) renvoie *vrai*.

3.1 De même qu'en t=0, `overlap` . *validate-open*(*from_up*, `and`) renvoie *vrai* si la *condition-ouverture* de *c* est satisfaite (ce qu'on va supposer) et si (3.1.1) *c* . *validate-open*(*from_up*, `overlap`) renvoie *vrai*.

3.1.1 *c* . *validate-open*(*from_up*, `overlap`) renvoie *vrai* si :
c . *check-sons-if-open*() renvoie *vrai*, ce qui est le cas puisque *c* n'a pas de composantes, et si
m_k . *check-if-son-open*(*c*) renvoie *vrai* pour toutes les activités m_k dont *c* est composante, excepté `overlap` puisque *from_up*, ce qui est le cas puisque *c* n'est la composante que de `overlap`.
Ainsi, le passage de *c* à `open` est validé.
Ainsi, le passage de `overlap` à `open` est validé.

4. En t=2, l'ouverture de *b* ne peut pas être examinée (et *a fortiori* validée) parce que b est encore `sleeping` (*update-status* ne teste intentionnellement l'ouverture que si l'activité est `waiting`).
c et *a* restent `open` (ainsi bien entendu que `overlap`, `meeting` et `and`).

5. En t=3, l'OPERATION en jeu dans *a* est `terminated`, et il y a donc lieu de valider le passage de *a* à `closed`. Comme cela est provoquée par l'OPERATION, on passe l'argument *from_down*. La validation nécessite que *a* . *check-sons-if-closed*(*null*) renvoie *vrai*, ce qui est le cas puisque *a* n'a pas de composantes, et que (5.1) m_k . *check-if-son-closed*(*a*) renvoie *vrai* pour toutes les activités m_k dont *a* est composante (y compris `meeting` puisque *from_down*, et en fait seulement elle).

5.1 `meeting` . *check-if-son-closed*(*a*) renvoie *vrai* si la *condition-ouverture* de *b* est satisfaite, ce que nous allons supposer et si (5.1.1) *b* . *validate-open*(*from_up*, `meeting`) renvoie *vrai*.

5.1.1 *b* . *validate-open*(*from_up*, `meeting`) renvoie *vrai* si :
(5.1.1.1) *b* `waiting` (ce n'est pas le cas) ou si *b* . *validate-waiting*(*from_up*, `meeting`), et si
b . *check-sons-if-open*() renvoie *vrai*, ce qui est le cas puisque *b* n'a pas de composantes, et si
(5.1.1.2) m_k . *check-if-son-open*(*b*) renvoie *vrai* pour toutes les activités m_k dont *b* est composante, excepté `meeting` puisque *from_up*, c'est-à-dire seulement pour `overlap`.

5.1.1.1 *b* . *validate-waiting*(*from_up*, `meeting`) renvoie *vrai* si *b* est sleeping (c'est le cas) et si :
b . *check-sons-if-waiting*() renvoie *vrai*, ce qui est le cas puisque *b* n'a pas de composantes, et si
(5.1.1.1.1) m_k . *check-if-son-waiting*(*b*) renvoie *vrai* pour toutes les activités m_k dont *b* est composante, excepté `meeting` puisque *from_up*, c'est-à-dire seulement pour `overlap`.

5.1.1.1.1 `overlap` . *check-if-son-waiting*(*b*) renvoie *vrai* si *c* est `open` (c'est le cas) ou, pour mémoire, s'il aurait pu être ouvert immédiatement si ça n'avait pas été le cas.
Le passage de *b* à `waiting` est validé.

`overlap` . *check-if-son-open*(*b*) renvoie *vrai*, car *b* n'est pas la première composante de `overlap`.
Le passage de *b* à `openg` est validé.
Le passage de *a* à `closed` est validé et réalisé par *turn-to-closed*.
La PROPAGATION (voir plus loin) entraîne alors le passage de *b* à `open` (noter qu'il a été validé).
c et *b* sont donc `open` (ainsi bien entendu que `overlap`, `meeting` et `and`).

6. En t=4, l'OPERATION en jeu dans *c* est `terminated`, et il y a donc lieu de valider le passage de *c* à `closed`. On passe l'argument *from_down*. La validation nécessite que *c* . *check-sons-if-closed*(*null*) renvoie *vrai*, ce qui est le cas puisque *c* n'a pas de composantes, et que (6.1) m_k . *check-if-son-closed*(*c*) renvoie *vrai* pour toutes les activités m_k dont *c* est composante (y compris `overlap` puisque *from_down*, et en fait seulement elle).

6.1 `overlap` . *check-if-son-closed*(*c*) renvoie *vrai* si *b* est `open`, ce qui est le cas.
Le passage de *c* à `closed` est validé.

7. En t=5, l'OPERATION en jeu dans *b* est `terminated`, et il y a donc lieu de valider le passage de *b* à `closed`. On passe l'argument *from_down*. La validation nécessite que *b* . *check-sons-if-closed*(*null*) renvoie *vrai*, ce qui est le cas puisque *b* n'a pas de composantes, et que m_k . *check-if-son-closed*(*b*) renvoie *vrai* pour toutes les activités m_k dont *b* est composante, c'est-à-dire (7.1) `overlap` et (7.2) `meeting`, puisque *from_down*.

7.1 `overlap` . *check-if-son-closed*(*b*) renvoie *vrai* si `overlap` . *validate-closed*(*from_down*, *b*) puisque *b* est la dernière composante de `overlap`. Cela est vérifié si `overlap est open (ce qui est vrai) et si :`
(7.1.1) `overlap` . *check-sons-if-closed*(*b*) renvoie *vrai*, et si
(7.1.2) m_k . *check-if-son-closed*(`overlap`) renvoie *vrai* pour toutes les activités m_k dont `overlap` est composante, y compris `and` puisque *from_down*, et en fait seulement elle.

7.1.1 `overlap` . *check-sons-if-closed*(*b*) renvoie *vrai*, parce qu'elle ne vérifie que des conditions sur la dernière composante de `overlap`, sauf si elle est passée en argument, ce qui est justement le cas.

7.1.2 `and` . *check-if-son-closed*(`overlap`) renvoie *vrai*, parce qu'elle ne vérifie aucune condition.
Ainsi, le passage de `overlap` à `closed` est validé.

7.2 `meeting` . *check-if-son-closed*(*b*) renvoie *vrai* si `meeting` . *validate-closed*(*from_down*, *b*) puisque *b* est la dernière composante de `meeting`. Cela est vérifié si `meeting est open (ce qui est vrai) et si :`
(7.2.1) `meeting` . *check-sons-if-closed*(*b*) renvoie *vrai*, et si
(7.2.2) m_k . *check-if-son-closed*(`meeting`) renvoie *vrai* pour toutes les activités m_k dont `meeting` est composante, y compris `and` puisque *from_down*, et en fait seulement elle.

7.2.1 `meeting` . *check-sons-if-closed*(*b*) renvoie *vrai* pour la même raison qu'en (7.1.1).

7.2.2 `and` . *check-if-son-closed*(`meeting`) renvoie *vrai* pour la même raison qu'en (7.1.2).
Ainsi, le passage de `meeting` à `closed` est validé.
La PROPAGATION (voir plus loin) entraîne alors le passage de `and` à `closed` (parce que toutes ses composantes sont `closed`).

5.4.2 Les procédures à invocation réflexe

5.4.2.1 Le mécanisme de propagation de situations

Le mécanisme de PROPAGATION est automatiquement invoqué au moment où la *situation* d'une activité passe à `waiting`, `open` ou `closed`, dans les METHODES *turn-to-waitin*g, *turn-to-open* et *turn-to-closed*.
La METHODE *turn-to-open* est invoquée sur une activité *A* avec deux arguments : la valeur *from_up* ou *from_down*, et l'activité à partir de laquelle on a opéré la validation du changement de *situation*. Après avoir changé la *situation*, elle propage les conséquences de ce changement vers les activités composantes ai et vers les activités *Bi* dont *A* est composante.
Avec les arguments *from_up* et *B** on propage la VALEUR `open` à toutes les composantes, par une invocation de *propagate-open-to-sons*(*null*) , puis à toutes les *Bi* sauf *B**, par une invocation sur *Bi* de *update-if-son-open*(*A*) .
Avec les arguments *from_down* et *a** on propage la VALEUR `open` à toutes les composantes sauf *a**, par une invocation de *propagate-open-to-sons*(*a**), puis à toutes les *Bi* , par une invocation de *update-if-son-ope*n(*A*).
Les METHODES *turn-to-waiting* et *turn-to-closed* ont le même comportement.
La METHODE *turn-to-open* modifie les spécifications de fenêtres d'ouverture et de fermeture d'activités, dans le cas où d'autres activités liées ont des VALEURS significatives pour les PROPRIETES *délais-o-o* et *délais-o-f*. La METHODE *turn-to-closed* fait de même dans le cas de VALEURS pour les PROPRIETES *délais-f-o* et *délais-f-f*.
Ce que font précisément les METHODES *propagate-?-to-sons* et *update-if-son-?* est spécifique de chaque *particularisation* de ACTIVITY SPECIFICATION (voir la section 4.5 –postconditions- et les annexes 3 et 4).
Le passage à `waiting` intervient dans deux cas :

- en début du mécanisme de SIMULATION, l'activité racine de chaque NOMINAL PLAN est rendue `waiting` pour permettre l'examen des conditions du passage à `open` ;
- au cours de la PROPAGATION elle-même, et en fonction du type des activités en jeu, la *situation* passe de `sleeping` à `waiting` : c'est par exemple le cas lorsqu'une composante non terminale d'une activité `before` devient `closed`. Noter aussi le passage exceptionnel de `closed` à `waiting`, pour l'unique composante d'une ITERATION ACTIVITY SPECIFICATION, si les conditions d'un nouveau passage à `open` peuvent être vérifiées.

Le passage à `open` intervient dans deux cas :

- à l'intérieur de la METHODE *update-status* si les conditions d'ouverture sont vérifiées ;
- au cours de la PROPAGATION elle-même, et en fonction du type des activités en jeu, la *situation* passe de `waiting` à `open` : c'est par exemple le cas lorsqu'une composante non terminale d'une activité `meeting` devient `closed`. Si les conditions d'ouverture de la composante suivante ne sont pas satisfaites à ce moment, c'est que la spécification du NOMINAL PLAN est incohérente et son exécution est arrêtée.

Le passage à `closed` intervient dans trois cas :

- à l'intérieur de la METHODE *update-status* si les conditions de fermeture sont vérifiées ;
- lorsqu'il s'agit d'une PRIMITIVE ACTIVITY SPECIFICATION et que l'OPERATION en jeu devient `terminated` ;
- au cours de la PROPAGATION elle-même, et en fonction du type des activités en jeu, la *situation* passe de `open` à `closed` : c'est par exemple le cas lorsqu'une composante d'une activité `coending` devient `closed`. Si les conditions de fermeture ne sont pas satisfaites à ce moment, c'est que la spécification du NOMINAL PLAN est incohérente et son exécution est arrêtée. Noter aussi le passage exceptionnel de `waiting` à `closed`, pour l'unique composante d'une ITERATION ACTIVITY SPECIFICATION, lorsque celle-ci devient `closed`.
  Noter encore qu'une activité primitive peut devenir `closed` sans que l'OPERATION en jeu soit devenue `terminated`.

Le passage à `cancelled` intervient dans deux cas :

- lorsqu'il s'agit d'une activité composante d'une DISJONCTION ACTIVITY SPECIFICATION et qu'une OPERATION incluse dans une autre composante devient `executing` ;
- lorsqu'il s'agit d'une OPTIONAL ACTIVITY SPECIFICATION ou d'une CONSTRAINED OPTIONAL ACTIVITY SPECIFICATION dont l'unique composante n'a pu être ouverte avant sa date de début au plus tard.

Noter que le passage à la VALEUR `cancelled` ne provoquent pas de PROPAGATION.

5.4.2.2 Le réflexe en fin d'exécution d'une opération

Lorsque le *degré-de-progression* d'une OPERATION devient *1*, l'INSTRUCTION qu'elle participe à spécifier devient `closed`, si *validate-closed* renvoie *vrai*. Ce passage provoque le mécanisme de PROPAGATION (voir ci-dessus). Si *validate-closed* renvoie *faux*, l'interprétation du NOMINAL PLAN est en échec.
Noter que le *degré-de-progression* ne peut atteindre la valeur *1* que pour les OPERATIONS non permanentes (voir § 4.7.2). Pour celles-ci, l'arrêt de réalisation de l'effet ne peut être provoqué que par le dépassement de la fenêtre de

fermeture (voir § 5.4.1.2) ou par propagation de l'ouverture (notamment dans le cas décrit dans le paragraphe suivant) ou de la fermeture d'une autre activité.

5.4.2.3 Le réflexe en début d'exécution d'une opération permanente

Lorsque la *situation* d'une OPERATION permanente (voir § 4.7.2) devient `executing`, et que l'INSTRUCTION qu'elle participe à spécifier est composante, à une profondeur quelconque, d'une activité `before` ou `meeting`, alors l'activité composante suivante dans la séquence[42] devient `waiting` si *validate-waiting* renvoie *vrai*. Ainsi, la terminaison de l'OPERATION permanente, ainsi que la fermeture de l'instruction interviendront dès que les conditions d'ouverture de l'activité suivante seront satisfaites. Noter que dans ce cas et seulement dans ce cas, la fermeture d'une composante d'une succession (`before` ou `meeting`) intervient au même instant que l'ouverture de la composante suivante (alors que dans le cas général, si la fermeture d'une composante intervient en *d*, l'ouverture de la composante suivante intervient au plus tôt en *d+1* –voir le § 4.5.1).

5.5 Le mécanisme d'établissement des instructions à exécuter

L'établissement des INSTRUCTIONS à exécuter est provoqué par l'OCCURRENCE d'un EVENEMENT standard : la *particularisation* MAKING INSTRUCTION LIST EVENT, caractérisée par le couple { MAKING INSTRUCTION LIST PROCESS, *procédure-exécution*}. MAKING INSTRUCTION LIST PROCESS est une *particularisation* standard de PROCESSUS PONCTUEL. Le programme de sa METHODE *procédure-exécution* consiste simplement à invoquer la METHODE *make-instruction-list* de l'OPERATING SYSTEM.

Le programme de *make-instruction-list*, appliqué à un ACTIVITIES RESOURCES BLOCK, réalise successivement les deux tâches suivantes :

- la mise en œuvre du mécanisme MAKING INSTRUCTION LIST pour cet ACTIVITIES RESOURCES BLOCK ;
- la PROGRAMMATION d'un EVEVEMENT invoquant le mécanisme ACTING INSTRUCTION LIST pour ce bloc. Noter que cet EVENEMENT, s'il se trouve être programmé au même INSTANT que des EVENEMENTS de la *particularisation* PROCEED OPERATION EVENT (cf. § 5.6.1) déjà présents dans l'agenda, est moins prioritaire que ces derniers.

```
MAKING INSTRUCTION LIST EVENT
→ MAKING INSTRUCTION LIST PROCESS. procédure-exécution {
    OPERATING SYSTEM. make-instruction-list(block) {
        OPERATING SYSTEM.instructions-to-act += MAKING INSTRUCTION LIST(block) ;
        PROGRAMMATION ACTING INSTRUCTION LIST EVENT(block) ; }
    }
```

Le mécanisme MAKING INSTRUCTION LIST génère un PACK OF EXECUTABLE INSTRUCTIONS à partir d'un ACTIVITIES-RESOURCES BLOCK. Plus précisément, il n'en considère que les ACTIVITY SPECIFICATIONS dont la *situation* est `open`. Dans le mécanisme UPDATING SITUATION, lorsqu'on considère tout à tour les différents ACTIVITIES-RESOURCES BLOCKS de la STRATEGY, on génère autant d'invocations du mécanisme ACTING INSTRUCTION LIST. Lorsque le mécanisme MAKING INSTRUCTION LIST est invoqué lorsqu'une allocation de RESOURCES est à réexaminer, notamment lorqu'une OPERATION est terminée, on ne génère un événement ACTING INSTRUCTION LIST EVENT que pour l'ACTIVITIES-RESOURCES BLOCK qui gère les RESOURCES en question.

Le mécanisme comporte trois étapes dans l'ordre suivant :

M1) La METHODE *expand-to-disjunctio*n opère sur le NOMINAL PLAN de l'ACTIVITIES-RESOURCES BLOCK et génère un ensemble de jeux alternatifs d'activités ouvertes. Certains de ces jeux peuvent être inconsistants, au sens où ils représentent une situation impossible ou inacceptable quant à l'engagement simultané de plusieurs RESOURCES, ou bien la mise en œuvre simultanée de plusieurs OPERATIONS.

M2) La FONCTION *set-all-feasible-alternatives* opère sur cet ensemble de jeux et génère un autre ensemble de jeux (PACKS OF EXECUTABLE INSTRUCTIONS), cette fois-ci tous consistants : l'établissement de la consistance d'un jeu passe éventuellement par la suppression d'activités pour lever un conflit d'usage de RESOURCES.

M3) La FONCTION *select-best-alternative* choisit le jeu consistant sur lequel opèrera le mécanisme ACTING INSTRUCTION LIST au titre de l'ACTIVITIES-RESOURCES BLOCK.

---

[42] L' « activité composante suivante dans la séquence » s'entend ici comme la prochaine activité (unique, par construction) devant (éventuellement) faire l'objet d'une ouverture.

Ainsi, dans l'activité `before(meeting(a, overlapping(b, c)), d)`, l'activité suivant `c` est `d` (bien que `c` ne soit suivie d'aucune composante dans le `overlapping`, et que le `overlapping` ne soit suivi d'aucune composante dans le `meeting`).

5.5.1 La création de l'ensemble de jeux possiblement inconsistants (M1)

A un NOMINAL PLAN, il correspond à un INSTANT un ensemble de jeux, possiblement inconsistants, de PRIMITIVE ACTIVITY SPECIFICATIONS. Par exemple, au NOMINAL PLAN : `equality(or(`a_1, a_2`), b)`, peut correspondre l'ensemble {(a_1, *b*), (a_2, *b*)}. On construit cet ensemble par une INVOCATION de la METHODE *expand-to-disjunction* du NOMINAL PLAN. Cette METHODE agit récursivement : l'INVOCATION de *expand-to-disjunction* sur une activité exploite les résultats des INVOCATIONS de *expand-to-disjunction* sur les composantes de l'activité, et ainsi de suite jusqu'aux activités primitives, où s'arrête la récursion puisqu'elles n'ont pas de composantes.

L'ensemble de jeux possiblement inconsistant est désigné à l'aide des notations suivantes :
- `s` : *un-set*(`a`$_1$, ..., `a`$_n$) désigne un jeu de PRIMITIVE ACTIVITY SPECIFICATIONS `a`$_1$, ..., `a`$_n$ indissociable, au sens où la STRATEGY exige de les mettre en œuvre toutes ensemble en l'INSTANT courant[43]. A défaut, aucune ne doit être mise en œuvre. Les RESOURCES que requiert chaque activité doivent être disponibles en cet INSTANT (*un-set* signifie « unsplittable set »).
- `s` : *set*(`a`$_1$, ..., `a`$_n$) désigne un jeu de PRIMITIVE ACTIVITY SPECIFICATIONS `a`$_1$, ..., `a`$_n$ tel que la STRATEGY autorise de n'en mettre en œuvre en l'INSTANT courant qu'un sous-ensemble si les RESOURCES requises ne sont pas disponibles pour toutes.
- *alt*(`s`$_1$, ..., `s`$_n$) désigne un ensemble de jeux alternatifs de PRIMITIVE ACTIVITY SPECIFICATIONS `s`$_1$, ..., `s`$_n$, au sens où la mise en œuvre d'un jeu exclut la mise en œuvre de tous les autres (*alt* signifie « alternative set »).
- *req*(`a`) qualifie une PRIMITIVE ACTIVITY SPECIFICATION `a` telle que la STRATEGY exige sa mise en œuvre, en l'INSTANT courant (*req* signifie « required »). C'est le cas des composantes d'une activité de type `strong_meeting` (*cf.* § 4.5), ou de celles dont le *degré-de-progression* de l'OPERATION en jeu était, au moment d'une suspension, supérieur au *seuil-suspension* (*cf.* § 4.7)[44].

Noter quelques règles de récriture (visant une réduction) sur ces notations :

set(*set*(`a`$_1$, ..., `a`$_n$))	→ *set*(`a`$_1$, ..., `a`$_n$)	*un-set*(`a`)	→ *set*(`a`)
set(*set*(`a`, `b`), *set*(`c`, `d`))	→ *set*(`a`, `b`, `c`, `d`)	*un-set*(*set*(`a`, `b`), *set*(`c`, `d`))	→ *un-set*(`a`, `b`, `c`, `d`)
set(`a`, `a`)	→ *set*(`a`)	*un-set*(*un-set*(`a`, `b`), *un-set*(`c`, `d`))	→ *un-set*(`a`, `b`, `c`, `d`)
alt(`s`, `s`)	→ *alt*(`s`)		
set(`a`, *req*(`a`))	→ *set*(*req*(`a`))		

5.5.1.1 L'expansion d'un plan en une disjonction de jeux (METHODE expand-to-disjunction)

Dans ce qui suit, on décrit les différents comportements de la METHODE *expand-to-disjunction* selon la *particularisation* de ACTIVITY SPECIFICATION à laquelle on l'applique.

▪ PRIMITIVE ACTIVITY SPECIFICATION :

Si `a` est une INSTANCE `open` de `cette` CLASSE, `a`.*expand-to-disjunction* renvoie :
alt(*set*(`a`)).
Si et seulement si le *degré-de-progression* de l'OPERATION en jeu est supérieur à son *seuil-suspension*, `a`.*expand-to-disjunction* renvoie :
alt(*set*(*req*(`a`))).

▪ BEFORE ACTIVITY SPECIFICATION, MEETING ACTIVITY SPECIFICATION, ITERATION ACTIVITY SPECIFICATION, OPTIONAL ACTIVITY SPECIFICATION, CONSTRAINED OPTIONAL ACTIVITY SPECIFICATION :

Soit `A` une INSTANCE de cette CLASSE et `a`$_i$ sa composante `open`, s'il y en a une.
`A`.*expand-to-disjunction* renvoie `a`$_i$.*expand-to-disjunction*.

▪ STRONG MEETING ACTIVITY SPECIFICATION :

Soit `A` une INSTANCE de cette CLASSE et `ai` sa composante `open`, s'il y en a une.

---

[43] Parce qu'elles relèvent d'une STRONG COSTARTING - - ou d'une STRONG EQUALITY ACTIVITY SPECIFICATION.

[44] Note importante : dans le codage de l'ontologie, le test du dépassement du *seuil-suspension* est réalisé par une INVOCATION REFLEXE (moniteur) attachée à la PROPRIETE degré-de-progression de la CLASSE de base OPERATION. Si le modélisateur spécialise ce moniteur au niveau de sous-classes d'OPERATION, cette spécialisation doit invoquer, en son début, le moniteur de la classe de base, pour que la comparaison *seuil-suspension* vs. d*egré-de-progression* soit faite.

Comme a_i est une PRIMITIVE ACTIVITY SPECIFICATION `par convention (cf. section 4.5)`, A.*expand-to-disjunction* renvoie :
alt(*set*(*req*(a_i))) si a_i n'est pas la première composante et si le *degré-de-progression* de l'OPERATION en jeu est *0*, et sinon
alt(*set*(a_i)) , ou
alt(*set*(r*eq*(a_i))), selon la position du *degré-de-progression* de l'OPERATION par rapport au *seuil-suspension*.

▪ STRONG COSTARTING ACTIVITY SPECIFICATION :

Soit A une INSTANCE de cette CLASSE et a_1, ..., a_n ses composantes open, s'il y en a.
Comme les a_i sont toutes des PRIMITIVE ACTIVITY SPECIFICATIONS, par convention, A.*expand-to-disjunct*ion renvoie :
alt(*un-set*(a_1, ..., a_n)) si le *degré-de-progression* est *0* pour toutes les OPERATIONS en jeu, et
alt(*set*(a_1, ..., a_n)) sinon.
Noter ici aussi que tous les a_i sont ensemble remplacés par *req*(a_i) si le *degré-de-progression* des OPERATIONS en jeu est supérieur à la VALEUR, qui doit être commune, de *seuil-suspension*.

▪ STRONG EQUALITY ACTIVITY SPECIFICATION :

Soit A une INSTANCE de cette CLASSE et a_1, ..., a_n ses composantes open, s'il y en a.
Comme les a_i sont toutes des PRIMITIVE ACTIVITY SPECIFICATIONS `par convention (cf. section 4.5)`, A.*expand-to-disjunction* renvoie :
alt(*un-set*(a_1, ..., a_n)).
Noter que *un-set* s'applique (et non *set*) quel que soit le *degré-de-progression* des OPERATIONS en jeu, pour imposer une progression conjointe des OPERATIONS.
Noter encore que tous les a_i sont ensemble remplacés par *req*(a_i) si le *degré-de-progression* des OPERATIONS en jeu est supérieur à la VALEUR, qui doit être commune, de *seuil-suspension*.

▪ OVERLAPPING ACTIVITY SPECIFICATION, INCLUSION ACTIVITY SPECIFICATION, COSTARTING ACTIVITY SPECIFICATION, COENDING ACTIVITY SPECIFICATION, EQUALITY ACTIVITY SPECIFICATION, CONJUNCTION ACTIVITY SPECIFICATION :

Soit A une INSTANCE de cette CLASSE et a_1, ..., a_n ses composantes open, s'il y en a.
Soit E_{ai} l'ensemble des composantes (*set* ou *un-set*) du résultat de a_i.*expand-to-disjunction*.
Alors A.*expand-to-disjunction* est un ensemble d'alternatives (*alt*) possédant comme composantes C_k (*set* ou *un-set*) les éléments du produit cartésien $E_{a1} \bullet \ldots \bullet E_{an}$[45,46].

▪ IMPLICIT CONJUNCTION ACTIVITY SPECIFICATION :

Soit A une INSTANCE de cette CLASSE et a la CONJUNCTION ACTIVITY SPECIFICATION résultant de l'EXPLICITATION de A (par *expand-to-conjunction*). A.*expand-to-disjunction* renvoie a.*expand-to-disjunction*.

▪ DISJUNCTION ACTIVITY SPECIFICATION, UNORDERED-BEFORE ACTIVITY SPECIFICATION :

Soit A une INSTANCE de cette CLASSE et a_1, ..., a_n ses composantes open, s'il y en a.
Soit E_{ai} l'ensemble des composantes (*set* ou *un-set*) du résultat de a_i.*expand-to-disjunction*.
Alors A.*expand-to-disjunction* renvoie un ensemble d'alternatives (*alt*) possédant comme composantes C_k (*set* ou *un-set*) les éléments E_k de l'union $E_{a1} \bullet \ldots \bullet E_{an}$[47].

---

[45] Par exemple, soit A : overlapping(or (a, b), or (c, d)).
E_{a1} : {*set*(a), *set* (b)}. E_{a2} : {*set* (c), *set* (d)}.
$E_{a1} \bullet E_{a2}$: { (*set* (a), *set* (c)), (*set* (a), *set* (d)), (*set* (b), *set* (c)), (*set* (b), *set* (d)) }.
C_1 : *set* (a, c). C_2 : *set* (a, d). C_3 : *set* (b, c). C_4 : *set* (b, d).
Le résultat de A.*expand-to-disjunction* est *alt*(*set* (a, c), *set* (a, d), *set* (b, c), *set* (b, d)).

[46] Par exemple, soit A : overlapping(a, strong_equality (b, c)).
E_{a1} : {*set*(a)}. E_{a2} : {*un-set* (b, c)}.
$E_{a1} \bullet E_{a2}$: { (*set* (a), *un-set* (b, c)) }.
C_1 : *set* (*set* (a), *un-set* (b, c)).
Le résultat de A.*expand-to-disjunction* est *alt*(*set* (*set* (a), *un-set* (b, c))), réduit à *alt*(*set*(a, *un-set* (b, c))).

[47] Par exemple, soit A : or(a, strong_equality (b, c)).
E_{a1} : {*set*(a)}. E_{a2} : {*un-set* (b, c)}.
$E_{a1} \bullet E_{a2}$: {*set* (a), *un-set* (b, c)}.
C_1 : *set* (a) ; C_2 : *un-set* (b, c).

Au total, compte tenu des différentes spécifications possibles d'un NOMINAL PLAN, l'ensemble de jeux possiblement inconsistants est une structure S, avec la forme générale suivante :

S : *alt*(s_1, ..., s_{ns}, us_1, ..., us_{nus})

où

s_i : *set*([$a_{i,1}$, ..., $a_{i,ji}$] [, *req*($b_{i,1}$), ..., *req*($b_{i,ki}$)] [, $us_{i,1}$,..., $us_{i,mi}$])

us_i : *unset*([$a_{i,1}$, ..., $a_{i,ji}$] [, *req*($b_{i,1}$), ..., *req*($b_{i,ki}$)])

$a_{i,j}$, $b_{i,j}$: PRIMITIVE ACTIVITY SPECIFICATION

La même activité primitive peut figurer dans plusieurs s_i et plusieurs us_i, en position de « feuille » (i.e. en position la plus profonde dans une branche).

5.5.1.2 L'élagage de l'ensemble de jeux possiblement inconsistants

Quelques règles permettent de transformer un ensemble de jeux en un ensemble plus petit, c'est-à-dire possédant moins de feuilles, correct, c'est-à-dire qui respecte la signification des notations *set*, *un-set* et *req*, et à partir duquel la suite de la procédure sélectionne le même jeu d'INSTRUCTIONS à mettre en œuvre.
Bien noter qu'on désigne par le mot « jeu » les composantes directes du *alt* (i.e. au premier niveau).

Règle 1 :
« S'il y a, dans un ensemble de jeux, un jeu dont une feuille est un *req*, alors supprimer tous les jeux dont aucune feuille n'est un *req*. »
En effet, un jeu contenant un *req* sera préféré à un jeu qui n'en contient pas, quel que soit le contenu des PREFERENCE RULES.

alt(*set*(a, b), *set*(*req*(c))) → *alt*(*set*(*req*(c)))

Noter que si le jeu finalement choisi par la fonction *select-best-altarnative* ne contient pas toutes les activités requises présentes dans l'un ou l'autre des jeux candidats, l'exécution du plan ne peut être poursuivie (puisqu'au moins une activité requise ne pourra voir son OPERATION exécutée).

Règle 2 :
« Au cours de l'allocation des activités d'un jeu candidat, et lorsque l'allocation d'une activité échoue, le jeu candidat est remplacé par un ensemble de jeux réduits des activités du jeu. Alors, supprimer les jeux réduits qui commencent par la combinaison inconsistante. »

alt(*set*(a, b,c,d), a *alloué, allocation de* b *en échec* → *alt*(*set*(a,c,d), *set*(b,c,d))

Règle 3 :
« Supprimer, dans tous les jeux qui la contiennent, une PRIMITIVE ACTIVITY SPECIFICATION telle que la *condition-de-faisabilité* de l'OPERATION en jeu est nécessairement non satisfaite[48]. »

alt(*set*(a, b), *set*(a, d)) et a . *operation* . *condition-de-faisabilité false* → *alt*(*set*(b), *set*(d))

Règle 4 :
« Supprimer les *unset* dans lesquels une feuille a été supprimée (par la règle 2). »

alt(*un-set*(a, b)) et a supprimée → *alt*()

5.5.1.3 Ordre d'examen des activités dans un jeu possiblement inconsistant

L'établissement de l'ensemble de jeux possiblement inconsistants se termine par un éventuel changement d'ordre des activités dans chacun des jeux. Le changement est opéré par l'application de règles, dont certaines ont une justification générale, et d'autres une justification liée au domaine de l'application. De plus, ces règles peuvent être combinées ou non, et la manière de les combiner peut être adaptée au contexte de l'étude. Il est donc donné au modélisateur la possibilité de spécifier lui-même l'ordre d'examen des activités dans un jeu d'activités possiblement inconsistant, en faisant agir des règles prédéfinies et/ou des règles particulières au domaine.
L'ordre d'examen des activités dans la procédure d'allocation des ressources n'est pas neutre sur son résultat[49]. La spécification de l'ordre est incluse dans la stratégie du pilote du système, lequel peut opter pour une spécification uniforme ou non sur l'ensemble des ACTIVITIES-RESOURCES BLOCKS de la STRATEGY. Pour une spécification

---

Le résultat de A.*expand-to-disjunction* est *alt*(*set*(a), *un-set*(b, c)).

[48] La version courante du codage de l'ontologie supprime une activité primitive telle que la condition de faisabilité de l'opération est (simplement) non satisfaite.

[49] Soit le jeu candidat *set*(a,b), avec a requérant une entité-ressource dans R1 et b requérant une entité-ressource dans R2, avec R2 une sous-classe de R1. Soit {r11} les instances de R1 et {r21} les instances de R2. L'allocation prioritaire de a peut lui allouer r21, rendant impossible l'allocation de b. L'allocation prioritaire de b lui alloue nécessairement r21, rendant possible l'allocation de a avec r11.

uniforme, le modélisateur surchargera la METHODE *procédure-ordre-allocation* du MANAGER. Pour des spécifications propres aux différents ACTIVITIES-RESOURCES BLOCKS, c'est la même METHODE, mais appartenant à cette dernière CLASSE, qui sera surchargée.
Les règles d'ordonnancement à vocation générale prédéfinies dans le codage de la présente ontologie sont les suivantes :

Règle 1 :
« Placer en tête du jeu les activités telles que la PROPRIETE *continuation* de l'OPERATED OBJECT SPECIFICATION ou d'une OPERATION RESOURCE SPECIFICATION ou de la PERFORMER SPECIFICATION est différente de new. »
En effet, la continuation d'une activité (*cf.* 5.5.2.4.4) impose une contrainte sur l'allocation de ressources à cette activité, contrainte qu'il est vraisemblablement préférable de satisfaire avant d'allouer les ressources aux autres activités du jeu.
Cette règle améliore l'efficacité du mécanisme MAKING INSTRUCTION LIST. On note toutefois que si elle n'était pas appliquée, le mécanisme devrait générer le même jeu d'INSTRUCTIONS à exécuter.

Règle 2 :
« Placer en queue du jeu les activités telles que la FONCTION de sélection de l'OPERATED OBJECT SPECIFICATION ou d'une OPERATION RESOURCE SPECIFICATION ou de la PERFORMER SPECIFICATION est `max`. »
En effet, réaliser d'abord la satisfaction maximale d'une telle activité (en terme d'allocation de ressource) risque de rendre impossible l'allocation de ressources aux activités suivantes. On préfère tendre à exécuter le plus possible d'activités, quitte à ce que la FONCTION de sélection *max* ne s'applique qu'aux ressources laissées disponibles par les activités déjà allouées.
De plus, la non application de cette règle rendrait non neutre l'ordre dans lequel les activités dotées d'une FONCTION de sélection *max* sont placées dans la liste. Ainsi par exemple, les interprétations des NOMINAL PLANS `and`(*a*, *b*) et `and`(*b*, *a*), où *a* seulement est dotée d'une FONCTION de sélection *max*, ne génèreraient-elles pas les deux mêmes ensembles d'INSTRUCTIONS allouées des mêmes ressources.

Règle 3 :
« Ordonner les activités du jeu selon les valeurs décroissantes de la PROPRIETE *priorité-d-allocation*. »
Cette règle permet au modélisateur d'intégrer, dans les valeurs qu'il donne à cette PROPRIETE des PRIMITIVE ACTIVITY SPECIFICATIONS, et de manière experte, différents facteurs de priorité qu'il a identifiés dans son domaine d'application.

5.5.2 La création de l'ensemble de jeux consistants (M2)

Cette deuxième phase du mécanisme MAKING INSTRUCTION LIST opère à partir d'une structure `S` (décrite en fin de paragraphe 5.5.1.1) qui contient les jeux possiblement inconsistants relatif à un ACTIVITIES RESOURCES BLOCK (ensemble élagué par l'application des trois règles du paragraphe 5.5.1.2).

```
PROCEDURE InstructionPackList set-all-feasible-alternatives(S jeux-possiblement-inconsistants) {
InstructionPackList instructionPacksToCast = ∅;  // l'ensemble soumis aux PREFERENCE RULES
POURTOUT jeu s DANS jeux-possiblement-inconsistants /*S*/ FAIRE {
    établir la consistance/inconsistance de s
    SI   s est consistant
    ALORS  ajouter s à instructionPacksToCast
    SINON  substitute ← ensemble des plus grands sous-ensembles consistants de s
           ajouter les éléments de substitute à instructionPacksToCast
}
RETURN instructionPacksToCast;
}
```

La METHODE *set-all-feasible-alternatives* de l'OPERATING SYSTEM, qui réalise cette phase, traite successivement et dans un ordre quelconque chaque jeu dans la structure `S`. Un jeu `s` est soumis aux PREFERENCE RULES dans la phase M3 s'il est consistant (on rappelle qu'un jeu est inconsistant s'il représente une situation impossible ou inacceptable quant à l'engagement simultané de plusieurs RESOURCES, ou bien la mise en œuvre simultanée de plusieurs OPERATIONS). Dans le cas contraire, on cherche à le remplacer par des sous-ensembles s_1', ..., s_n' de composantes de `s`, qui forment des jeux consistants. On élimine parmi eux les jeux qui sont un sous-ensemble d'un autre. Les jeux restant remplacent `s` dans `S`. La PROCEDURE retourne un ensemble de jeux d'INSTRUCTIONS, (structure InstructionPackList) parce qu'en cas de consistance d'un jeu, les PRIMITIVE ACTIVITY SPECIFICATIONS qui en sont les ELEMENTS se voient allouer les RESOURCES qu'elles requièrent et sont ainsi complètement déterminées.

Le reste de cette section se concentre sur le traitement d'un seul jeu d'activités. On présente d'abord les structures de données sur lesquelles s'appuient les procédures qui établissent sa consistance/inconsistance et qui le remplacent s'il est inconsistant par un ou plusieurs jeux consistants, puis les procédures elles-mêmes.

5.5.2.1 Structures de données pour l'établissement des jeux consistants

5.5.2.1.1 Notion de dégradation minimale d'un jeu d'activités

Afin de l'appliquer à un jeu d'activités s trouvé inconsistant, on présente ici la procédure de construction les plus grands jeux réduits qu'on peut construire à partir de s. Chaque nouveau jeu est le jeu s auquel on enlève une et une seule composante (différente à chaque fois), pour dégrader le moins possible la spécification initiale du NOMINAL PLAN. On applique pour cela la FONCTION *split-activity-set*, sur s. Le contenu du jeu renvoyé dépend de la forme de s.

- pour s : *set*(x_1, ..., x_n) , où x_i : a_i ou *unset*(a_1, ..., a_n), s.*split-activity-set* renvoie :
  { s_1 : *set*(x_2, ..., x_n), ..., s_k : *set*(x_1, ..., x_{k-1}, x_{k+1}, x_n), ..., s_n : *set*(x_1, ..., x_{n-1}) }.

C'est le cas où aucune PRIMITIVE ACTIVITY SPECIFICATION n'est requise (par la notation *req*).
Comme cas particulier, pour s : *set*(x_1), s.*split-activity-set* renvoie { s_1 : *set*() }.

- pour s : *un-set*(x_1, ..., x_n), où x_i : a_i ou *req*(a_i), s.*split-activity-set* renvoie :
  { s_1 : ∅ }.

Par définition, les composantes d'un *un-set* sont indissociables.

- pour s : *set*(a_1, ..., a_n, *req*(b)), s.*split-activity-set* renvoie :
  { s_1 : *set*(a_2, ..., a_n, *req*(b)), ..., s_k : *set*(a_1, ..., a_{k-1}, a_{k+1}, a_n, *req*(b)), ..., s_n : *set*(a_1, ..., a_{n-1}, *req*(b)) }.

Toute composante *req* se retrouve dans tous les jeux.
Comme cas particulier, pour s : *set*(*req*(b)), s.*split-activity-set* renvoie { s_1 : s }.

- pour s : *set*(a_1, ..., a_n, *un-set*(c, req(d))), s.*split-activity-set* renvoie :
  { s_1 : *set*(a_2, ..., a_n, *un-set*(c, req(d))), ..., s_k : *set*(a_1, ..., a_{k-1}, a_{k+1}, a_n, *un-set*(c, req(d))), ..., s_n : *set*(a_1, ..., a_{n-1}, *un-set*(c, req(d))) }.

Une composante *un-set* qui contient une composante *req* se retrouve dans tous les jeux.

Un jeu s tel que s.*split-activity-set* renvoie un ensemble non vide et différent de s est dit dégradable. Les éléments de la liste renvoyée par s.*split-activity-set* sont des jeux dits dégradés de s.

5.5.2.1.2 Le treillis des jeux dégradés

La FONCTION *build-lattice* construit, à partir du jeu de PRIMITIVE ACTIVITY SPECIFICATIONS initial s, un treillis $\mathcal{L}_s$ dont le nœud racine correspond à s[50]. On définit l'augmentation du treillis à partir d'un nœud x comme l'ajout des nœuds correspondant aux jeux dégradés de x et le marquage de la liaison entre chaque nœud ainsi ajouté et x. Pour construire le treillis, on réalise une augmentation d'abord à partir de s puis, récursivement, à partir de chaque nœud ainsi ajouté, si le jeu correspondant est dégradable. Le treillis $\mathcal{L}_s$ est représenté par une liste du même nom. C'est une liste de sous-listes L_k de jeux de PRIMITIVE ACTIVITY SPECIFICATIONS correspondant aux nœuds de niveau k. L'indice k correspond au degré de dégradation de s :

- la sous-liste L_0 ne contient que l'élément s.
- la sous-liste L_1 est la liste renvoyée par s. *split-activity-set*.
- la sous-liste L_k est la concaténation des listes renvoyées par l'application de *split-activity-set* aux éléments de L_{k-1}. On assure que chaque élément n'est présent qu'une fois dans la sous-liste.
- $k \leq kmax$, avec kmax tel que aucun élément de L_{kmax} ne soit dégradable.

Soit, par exemple, s : *set*(a1, a2, *req*(b), *un-set*(c, *req*(d))).
La liste $\mathcal{L}$s est : { L_0 : { *set*(a1, a2, *req*(b), *un-set*(c, *req*(d))) }
L_1 : { *set*(a2, *req*(b), *un-set*(c, *req*(d))), *set*(a1, *req*(b), *un-set*(c, *req*(d))) }
L_2 : { *set*(*req*(b), *un-set*(c, *req*(d))) }
}
Soit encore, s' : *unset*(a1, a2, *req*(b)).

[50] On dit « correspond à » parce qu'un nœud n'est pas seulement défini par le jeu d'activités (*cf.* § 5.5.2.1.4).

La liste $\mathcal{L}_s$' est : { L_0' : { *unset*(a1, a2, *req*(b)) }
}
On remarque que *split-activity-set* appliquée à e : *set*(*req*(b), *un-set*(c, *req*(d))) renvoie e parce que les deux composantes de e contiennent un *req*. Le jeu e n'est pas dégradable. On ne peut plus augmenter $\mathcal{L}_S$.

Pour mémoriser les liaisons entre les nœuds, on associe à chaque jeu d'un L_k, pour k=1, ..., kmax :
- la liste, éventuellement vide, des jeux dégradés à partir de lui,
- la liste des jeux, éventuellement vide, dont il est un jeu dégradé.

Une fois établie la structure du treillis, en tenant compte notamment des opérateurs *req* et *unset*, on supprime toutes leurs interventions. Le treillis $\mathcal{L}_S$ et $\mathcal{L}_{S'}$ deviennent :

{L_0 : { *set*(a_1, a_2, b, c, d) } {$L_{0'}$: { *unset*(a_1, a_2, b) }
L_1 : { *set*(a_2, b, c, d), *set*(a_1, b, c, d) } }
L_2 : { *set*(b, c, d) }
}

En effet, dans l'exemple $\mathcal{L}_S$, c'est bien un des quatre ensemble d'activités (initial ou dégradé) qu'il conviendra de mettre en œuvre et aucun autre. Dans la suite de la procédure, les opérateurs *set* et *unset*) sont traités de la même façon. On désigne par ActivitySpecificationList la structure de donnée correspondant à l'opérateur *set* (ou *unset*) et ses arguments.

On note $\mathcal{L}$ l'ensemble des treillis $\mathcal{L}_S$ construits à partir de tous les jeux s de S.

5.5.2.1.3 Structure de données pour les ressources allouées à une activité

On associe à chaque activité dans un nœud donné une structure V qui représente les RESOURCES allouées à sa mise en œuvre, compte tenu des allocations déjà faites aux autres activités dans le même noeud. Lors de la création du nœud, V est vide pour toutes les activités. V est ensuite augmentée dans le mécanisme PRE-ALLOCATING RESOURCES. La structure de V est la suivante, définissant la CLASSE INSTRUCTION ALLOCATION :

V : { oo-used-resources, r-used-resources, p-used-resources }
oo-used-resources : liste d'INSTANCES de *particularisations* de OPERATED OBJECT
r-used-resources : liste d'INSTANCES de *particularisations* de RESOURCES propres d'opérations
p-used-resources : liste d'INSTANCES de *particularisations* de PERFORMER

On appelle CandidateInstruction une INSTANCE de la CLASSE SINGLE INSTRUCTION SET doté du couple de PROPRIETES {a, V_a}, où a est une activité et V_a la structure V correspondante pour un nœud.

Bien noter qu'une même activité a peut être associée à des structures V différentes dans deux nœuds différents, si les activités précédant a dans chacun des nœuds contraignent différemment l'allocation de RESOURCES à a.

Une fois qu'un jeu d'activités aura été choisi par la FONCTION *select-best-alternative* (phase M3, *cf.* section 5.5.3), chaque structure V associée à une activité dans une CandidateInstruction du jeu (nœud) deviendra la VALEUR d'une PROPRIETE de l'activité[51] (devenue alors INSTRUCTION). Son premier composant (oo-used-resources) devient la VALEUR de la *liste-opérée* de l'OPERATED OBJECT SPECIFICATION de l'activité.

5.5.2.1.4 Structure de données pour les ressources allouées à un jeu d'activités

On associe à chaque nœud une structure U qui représente les RESOURCES allouées à la mise en œuvre de toutes les activités du nœud. Lors de la création de L_S, U est vide. U est ensuite augmentée dans le mécanisme PRE-ALLOCATING RESOURCES. La structure d'un nœud est donc finalement la suivante :

Node : { U, liste de CandidateInstructions }

La structure de U est la suivante, définissant la CLASSE INSTRUCTION SET ALLOCATION :

U : { liste de ResourceUsageItems }

[51] La version courante du codage de l'ontologie ne réalise pas cette affectation..

`ResourceUsageItem` : { `r`, liste de `CoEngagementItems` }
`r` : INSTANCE (directe ou indirecte) de RESOURCE ou d'OPERATION
`CoEngagementItem` : { `symb, liste-instances` }
`symb` : identificateur de CLASSE d'une *particularisation* de RESOURCE ou d'OPERATION
`liste-instances` : liste d'INSTANCES de la CLASSE d'identificateur `symb`

Il y a dans `U` un `ResourceUsageItem` par RESOURCE engagée à la mise en œuvre d'au moins une activité dans le nœud. Un `ResourceUsageItem` portant sur la RESOURCE `r` indique quelles RESOURCES sont co-engagées avec `r` (voir la définition dans l'article RESOURCE SHARING VIOLATION CONDITION, § 4.2). Les co-engagements sont la `liste de CoEngagementItems`. Un `CoEngagementItem` porte sur une seule *particularisation* de RESOURCES co-engagées avec `r`. Il indique quelles INSTANCES de cette *particularisation* sont co-engagées avec `r`.

5.5.2.2 Etablissement des jeux consistants pour le plan complet (METHODE set-all-feasible-alternatives)

C'est la PROCEDURE déjà présentée *set-all-feasible-alternatives* qui réalise cette tâche. Déjà présentée de façon sommaire en introduction du paragraphe 5.5.2, elle peut être maintenant être récrite en tenant compte des structures ci-dessus. On utilise les notations suivantes : Lattice pour la CLASSE des treillis, LatticeList pour la CLASSE des ensembles de treillis, InstructionPackList pour la CLASSE des ensembles de PACK OF EXECUTABLE INSTRUCTIONS

La FONCTION *build-lattice* est appliquée à chaque ensemble d'activités possiblement inconsistant. Elle établit le treillis des nœuds (*cf.* § 5.5.2.1.2), chacun correspondant soit à l'ensemble initial de PRIMITIVE ACTIVITY SPECIFICATIONS, soit à un ensemble dégradé.
La METHODE *pre-allocate* opère sur un treillis (lattice). Pour chaque nœud, elle établit deux caractères :
- la consistance ou l'inconsistance du jeu d'activités correspondant : un jeu est consistant si on a pu allouer à toutes les activités les RESOURCES qu'elles requièrent sans violer de contraintes d'allocation ;
- l'optimalité ou la sous-optimalité du jeu : un jeu est optimal s'il est consistant et s'il n'existe pas dans le treillis un autre jeu consistant qui en soit un sur-ensemble.

. La METHODE *get-consistant-upper-subsets* dresse ensuite la liste des jeux optimaux dans le treillis. La liste comprend un unique jeu correspondant au nœud lattice lui-même s'il est consistant, ou bien, s'il est inconsistant, l'union des listes renvoyées par l'application de *get-consistant-upper-subsets* sur les jeux dégradés à partir de lui.

```
PROCEDURE InstructionPackList set-all-feasible-alternatives(S jeux-possiblement-inconsistants) {
InstructionPackList instructionPacksToCast = ∅;  // l'ensemble soumis aux PREFERENCE RULES
LatticeList L  =  ∅ ;
POURTOUT jeu s DANS jeux-possiblement-inconsistants FAIRE {
    Ls = build-lattice(s) ;
    L  .push(Ls) ; }
TANTQUE L ≠ ∅  FAIRE {
    Lattice lattice = L . pop() ;
    lattice.pre-allocate(L);        // avec augmentation éventuelle de L
    InstructionPackList substitute = lattice.get-consistant-upper-subsets() ;
    POURTOUT PACK OF EXECUTABLE INSTRUCTIONS pack DANS substitute FAIRE
        instructionPacksToCast.add-if-new(pack) ;
     }
RETURN instructionPacksToCast;
  }
```

Une PRIMITIVE ACTIVITY SPECIFICATION peut figurer dans deux jeux différents `s1` et `s2` soumis à la PROCEDURE *set-all-feasible-alternatives*. Dans chacun des deux jeux, elle peut se voir allouer, en fonction des réquisitions des autres activités dans le jeu, soit des RESOURCES identiques, soit des RESOURCES différentes. Dans le second cas, les INSTRUCTIONS résultant de ces allocations seront différentes. Si les deux jeux `s1` et `s2` étaient deux ensembles des mêmes PRIMITIVE ACTIVITY SPECIFICATIONS (et dans le même ordre), chaque activité se verrait allouer les mêmes RESOURCES dans les deux jeux. Les deux PACK OF EXECUTABLE INSTRUCTIONS résultants ne pourraient pas être départagés par les PREFERENCE RULES, quelles qu'elles soient. On veille donc à ne pas soumettre aux règles de préférence deux PACK OF EXECUTABLE INSTRUCTIONS identiques (c'est ce qu'assure la METHODE *add-if-new*). Noter que si un jeu d'activités figure dans deux treillis différents, cela peut être dans chacun d'eux soit dans le noeud racine soit au titre de jeu dégradé à partir de la racine.

Noter enfin qu'une PRIMITIVE ACTIVITY SPECIFICATION e, considérée à l'intérieur d'un treillis L_s, peut supporter plusieurs allocations différentes de RESOURCES : c'est le cas si elle utilise les FONCTIONS de sélection *disj*() ou *sets*(n) (voir dans la section 4.4 la description des OPERATED OBJECT SPECIFICATIONS et PERFORMER SPECIFICATIONS, et dans la section 4.2 la description des OPERATION RESOURCE SPECIFICATIONS). Dans cette situation, le treillis L_s est remplacé dans L par autant de treillis qu'il y a d'allocations différentes pour e. Ces treillis ne diffèrent que sur l'allocation de RESOURCES à e.

5.5.2.3 Etablissement des jeux consistants pour un seul treillis issu du plan (METHODE pre-allocate)

On rappelle que le jeu s correspondant à la racine du treillis est soumis aux PREFERENCE RULES s'il est consistant, et que dans le cas contraire on cherche à le remplacer par des sous-ensembles s_1', ..., s_n' de composantes de s, qui forment des jeux consistants. On présente ici comment la METHODE *pre-allocate* exploite la structure en treillis pour réaliser cette tâche.

```
void Lattice::pre-allocate(LatticeSet L ) {
//L  est l'ensemble de treillis dont this vient d'être retiré comme premier élément
//A la construction de this, is-inconsistant(), is-suboptimal() et is-allocated() sont false
//     pour tous les jeux
POURTOUT niveau k de this FAIRE {
    jeuxNiveauk = l'ensemble des jeux ni inconsistants ni sous-optimaux à ce niveau ;
    POURTOUT jeu DANS jeuxNiveauk FAIRE {
        jeu-épuré = l'ensemble des activités de jeu non déjà pourvues en ressources
        TANTQUE jeu-épuré non vide et on n'a pas rencontré d'inconsistance FAIRE {
            e = la première activité de jeu-épuré
            UList allocs = e . get-specimens() ;
                    //   l'ensemble des alternatives d'allocations de ressources pour l'élément e,
                    //   valides compte tenu des allocations aux activités précédant e dans jeu
            SI on a trouvé au moins une allocation valide des ressources pour e
            ALORS {
                . créer autant de treillis que d'allocations (U alloci) après la première
                . dans tous les treillis, dans le jeu (ou sa copie), Ve = alloci,
                . dans tous les treillis, ajouter l'allocation (alloci) pour e à l'allocation globale pour jeu
                . dans tous les treillis, propager les deux actions ci-dessus dans les jeux dégradés
                     qui contiennent e
                . dans L , remplacer this par ces treillis
            }
            SINON {
                déclarer jeu inconsistant
                déclarer inconsistants les jeux dégradés qui commencent par les mêmes activités
            }
        }   // fin TANTQUE non vide
    }   // fin POURTOUT jeu
}   // fin POURTOUT niveau
}
```

La METHODE *pre-allocate* se positionne sur le nœud racine du treillis et tente successivement d'allouer à chaque PRIMITIVE ACTIVITY SPECIFICATION du jeu correspondant les RESOURCES qu'elle requiert. Le jeu est consistant si toutes ses PRIMITIVE ACTIVITY SPECIFICATIONS ont été examinées sans échec. Si une PRIMITIVE ACTIVITY SPECIFICATION ne peut pas être pourvue en RESOURCES, compte tenu des allocations déjà faites pour les activités la précédant dans le jeu et des contraintes d'allocation diverses, le jeu est déclaré inconsistant. Dans ce cas, on déclare aussi inconsistants les jeux dégradés qui requièrent le même ensemble de RESOURCES qu'il n'est pas possible d'allouer conjointement. Pratiquement, il s'agit des jeux dégradés dont les premières activités sont justement celles examinés dans le jeu racine jusqu'à la rencontre de l'inconsistance. Par exemple, si le nœud racine contient le jeu [abcde], et qu'une inconsistance est trouvée lors de l'examen de c, alors les nœuds dégradés [abcd], [abce] et [abc] sont aussi inconsistants. Par contre, et par exemple, le jeu [abde] ne peut pas encore être déclaré inconsistant. On examine ensuite tour à tour les jeux dégradés du nœud racine, à condition qu'ils n'aient pas été déclarés inconsistants. Lorsque tous les jeux dans un niveau du treillis ont été examinés, on examine ceux du niveau inférieur. On continue ainsi tant qu'on n'a pas atteint un niveau où tous les jeux ont été déclarés inconsistants. Dans ce cas, la procédure peut être arrêtée puisque tous les niveaux encore inférieurs ne contiennent que des jeux inconsistants. Noter encore que lorsque toutes les activités d'un jeu ont été allouées avec succès (pas de rencontre

d'inconsistance), les nœuds dégradés à partir de lui sont déclarés sous-optimaux : en effet l'application des REGLES DE PREFERENCE éliminerait de toutes façons plus tard le jeu le moins complet. De même que les jeux inconsistants, les jeux sous-optimaux ne font pas l'objet d'une tentative d'allocation.

Une remarque particulière porte sur la variable jeu-épuré. Une activité du jeu n'apparaît pas dans jeu-épuré si et seulement si elle est déjà associée à une structure V non vide (alors nécessairement validée). Ce cas est rencontré uniquement lorsque la structure V a été valuée par propagation à partir d'un jeu à un niveau supérieur dans le treillis. *A contrario*, ce n'est notablement pas le cas :

- pour toutes les activités dans le nœud racine (parce qu'il n'est pas modifié par propagation) ;
- lorsqu'on a spécifié dans au moins une spécification de réquisition de RESOURCE que, dans le cas d'interruption/reprise de l'activité, ce sont les RESOURCES allouées au moment de l'interruption (sauf cas particuliers) qui doivent être à nouveau allouées. La recherche d'INSTANCES de RESOURCES est alors simplifiée, mais son résultat doit cependant être validé (du point de vue des contraintes d'allocation) compte tenu des allocations aux activités précédentes dans le jeu. L'activité apparaît dans jeu-épuré parce que, au moment de la construction de jeu-épuré, la structure V associée est vide.

Une fois le treillis complètement exploité, chaque nœud (plus précisément le jeu correspondant) est dans un des états suivants :

- toutes les PRIMITIVE ACTIVITY SPECIFICATIONS sont pourvues en RESOURCES ; dans ce cas, le jeu a été ou non déclaré sous-optimal ;
- au moins une PRIMITIVE ACTIVITY SPECIFICATION n'est pas pourvue, parce que le jeu a été déclaré soit inconsistant (lors de son propre examen ou bien par propagation de l'inconsistance d'un jeu à un niveau supérieur) soit sous-optimal.

A ce stade, la METHODE *get-consistant-upper-subsets* collecte dans le treillis les jeux complètement alloués et non sous-optimaux, selon le schéma suivant :

```
InstructionPackList Lattice::get-consistant-upper-subsets() {
alternatives = ∅ ;
POURTOUT niveau k FAIRE {
    jeuxNiveauk = l'ensemble des jeux à ce niveau ;
    POURTOUT jeu DANS jeuxNiveauk FAIRE {
        SI jeu est complètement alloué et n'est pas sous-optimal
        ALORS
            ajouter jeu à alternatives
    }
renvoyer alternatives
}
```

5.5.2.4 Allocation de ressources à une activité dans un jeu (METHODE get-specimens)

C'est la METHODE *get-specimens* de SINGLE INSTRUCTION SET (voir § 5.5.2.1.3) qui réalise cette tâche. Elle correspond au mécanisme de DETERMINATION de l'activité. L'allocation de RESOURCES à une activité dans un jeu conjugue étroitement deux sous-tâches :

- la détermination des RESOURCES requises par l'activité ;
- l'analyse de l'acceptabilité de ces réquisitions de RESOURCES, compte tenu des réquisitions des autres activités dans le jeu.

Ces deux tâches sont d'abord décrites séparément, avant d'aborder leur imbrication. On traitera enfin la question de l'allocation de RESOURCES à une activité suspendue en cours d'exécution et candidate à la continuation.

5.5.2.4.1 La détermination des ressources requises par une activité

Cette tâche, décrite plus formellement en annexe 5, consiste à interpréter l'OPERATED OBJECT SPECIFICATION, l'OPERATION SPECIFICATION et la PERFORMER SPECIFICATION en termes de RESOURCES requises, en opérant leurs DEVELOPPEMENTS. Comme ces DEVELOPPEMENTS produisent chacun une liste d'ensembles alternatifs de RESOURCES[52], la DETERMINATION d'une PRIMITIVE ACTIVITY SPECIFICATION produit une liste d'alternatives de réquisition. Chaque alternative de réquisition est un ensemble d'INSTANCES de RESOURCES satisfaisant la spécification et permettant la

[52] On rappelle qu'une spécification sous-tend une liste non dégénérée d'alternatives de RESOURCES lorsqu'on utilise les FONCTIONS de sélection *disj* ou *sets(n)*. Les FONCTIONS de sélection *any(n)*, *all* et *max* renvoient bien une liste, mais qui ne contient qu'une seule alternative.

mise en œuvre de l'activité si les RESOURCES sont `disponibles`. On rappelle qu'une RESOURCE est `disponible` si et seulement si toutes les AVAILABILITY CONSTRAINTS portant sur elle sont satisfaites.

Soit, pour illustration, une activité primitive dont les trois COMPOSANTS sont :

`oo-spec` : { < `O`, *true*, *all*, *return_instances* >, *disj*() }, avec `O` une CLASSE d'ENTITES comportant 2 INSTANCES `o1` et `o2`.

`ot-spec` : { < `OT1`, *true*, *1*, *instantiate* > ▯ { `ORS1`, `ORS2` } }, avec `ORS1` et `ORS2`, deux OPERATION RESOURCE SPECIFICATIONS :

`ORS1` : { < `R1`, *true*, *all*, *return_instances* >, *sets*(*1*), `proportional` }, avec `R1` une classe de RESOURCES comportant 2 et seulement 2 *particularisations* `R1a` (INSTANCES `s1`, `s3`) et `R1b` (INSTANCES `s2`, `s4`)

`ORS2` : { < `R2`, *true*, *all*, *return_instances* >, *any*(*1*) , `proportional` }, avec `R2` une classe de RESOURCES avec les INSTANCES `t1`, `t2`

`p-spec` : { < `P`, *true*, *all*, *return_instances* >, *sets*(*1*) }, avec `P` une CLASSE de PERFORMERS comportant 2 et seulement 2 *particularisations* `P1` (INSTANCES `p1`, `p3`, de *puissance 1*) et `P2` (INSTANCES `p2`, `p4`, de *puissance 2*)

Le DEVELOPPEMENT de `oo-spec` renvoie la liste { (OPERATED OBJECT{`o1`}) (OPERATED OBJECT{`o2`}) }[53].

Le DEVELOPPEMENT de `p-spec` renvoie la liste { (`p1`) (`p2`) }.

Pour l'alternative (`p1`), le DEVELOPPEMENT des deux RESOURCES SET SPECIFICATIONS de `ot-spec` renvoie :

pour `ORS1` : { (`s1`) (`s2`) }

pour `ORS2` : { (`t1`) },

ce qui entraîne d'associer à l'alternative (`p1`) l'ensemble d'alternatives de réquisition de RESOURCES { (`s1`, `t1`), (`s2`, `t1`) }. L'ensemble d'alternatives de réquisition correspondant à l'ensemble (`p1`) est donc

{ (`p1`, `s1`, `t1`), (`p1`, `s2`, `t1`) }

Pour l'alternative (`p2`), le DEVELOPPEMENT des deux RESOURCES SET SPECIFICATIONS de `ot-spec` renvoie :

pour `ORS1` : { (`s1`, `s3`) (`s2`, `s4`) (`s1`, `s2`)}

pour `ORS2` : { (`t1`, `t2`) },

ce qui entraîne d'associer à l'alternative (`p2`) l'ensemble d'alternatives de réquisition de RESOURCES { (`s1`, `s3`, `t1`, `t2`), (`s2`, `s4`, `t1`, `t2`) , (`s1`, `s2`, `t1`, `t2`) }. L'ensemble d'alternatives de réquisition correspondant à l'ensemble (`p2`) est donc

{ (`p2`, `s1`, `s3`, `t1`, `t2`), (`p2`, `s2`, `s4`, `t1`, `t2`) , (`p2`, `s1`, `s2`, `t1`, `t2`) }

Le résultat des DEVELOPPEMENTS de `ot-spec` et de `p-spec` est :

{ (`p1`, `s1`, `t1`), (`p1`, `s2`, `t1`), (`p2`, `s1`, `s3`, `t1`, `t2`), (`p2`, `s2`, `s4`, `t1`, `t2`), (`p2`, `s1`, `s2`, `t1`, `t2`) }

Rapporté au DEVELOPPEMENT de `oo-spec`, on obtient finalement la liste suivante d'alternatives de réquisition :

{ (OPERATED OBJECT{`o1`}, `p1`, `s1`, `t1`), (OPERATED OBJECT{`o1`}, `p1`, `s2`, `t1`), (OPERATED OBJECT{`o1`}, `p2`, `s1`, `s3`, `t1`, `t2`), (OPERATED OBJECT{`o1`}, `p2`, `s2`, `s4`, `t1`, `t2`), (OPERATED OBJECT{`o1`}, `p2`, `s1`,`s2`, `t1`, `t2`),
(OPERATED OBJECT{`o2`}, `p1`, `s1`, `t1`), (OPERATED OBJECT{`o2`}, `p1`, `s2`, `t1`), (OPERATED OBJECT{`o2`}, `p2`, `s1`, `s3`, `t1`, `t2`), (OPERATED OBJECT{`o2`}, `p2`, `s2`, `s4`, `t1`, `t2`), (OPERATED OBJECT{`o2`}, `p2`, `s1`,`s2`, `t1`, `t2`) }

5.5.2.4.2 Acceptabilité d'une réquisition de ressources par une activité

5.5.2.4.2.1 Principe général

On rappelle qu'on s'intéresse à une activité dans un jeu possiblement inconsistant d'activités. On suppose que la DETERMINATION a fourni une ou plusieurs alternatives de réquisition de RESOURCES par cette activité. Il s'agit maintenant d'examiner si ces réquisitions sont satisfiables. On utilisera pour exemple une des alternatives du paragraphe précédant : {(OPERATED OBJECT{`o1`},`p2`,`s1`,`s2`,`t1`,`t2`)}, qu'on désigne maintenant plus brièvement par {`o1`,`p2`,`s1`,`s2`,`t1`,`t2`)}. Il faut ajouter ici que cette alternative doit être augmentée avec l'INSTANCE d'OPERATION. On va en effet tester des contraintes de consistance de l'activité et du jeu dans lequel elle est incluse. Ces contraintes sont des ACTIVITY INCONSISTENCY CONDITIONS, des RESOURCE SHARING VIOLATION CONDITIONS et des ACTIVITIES RESOURCES INCONSISTANT COMMITMENTS. On peut vérifier dans la section 4 que ces contraintes limitent les co-engagements non seulement entre RESOURCES, mais aussi entre RESOURCES et OPERATIONS. L'alternative de réquisition prise en exemple est donc en fait : (`q`,`o1`,`p2`,`s1`,`s2`,`t1`,`t2`), où `q` est l'INSTANCE de la *particularisation* Q d'OPERATION résultant du DEVELOPPPEMENT de l'OPERATION SPECIFICATION de l'activité.

Supposant donc réalisée la DETERMINATION des RESOURCES requises, la METHODE *get-specimens* appliquée à l'activité peut se résumer de la manière suivante :

[53] Les INSTANCES d'OPERATED OBJECT sont soit déjà créées, lorsque l'objet de l'activité est explicitement désigné lors de la spécification du NOMINAL PLAN, ou dans le cas de continuation d'activité, soit créées au moment du DEVELOPPEMENT de l'OPERATED OBJECT SPECIFICATION.

```
UList SingleInstructionSet::get-specimens (U Unode, EntityListList resReqs ) {
// this est doté du couple de PROPRIETES {a, Va}, où a est une activité primitive et Va sa structure V
// Unode : la structure U avec les RESOURCES déjà allouées aux activités précédant a dans le nœud
// en cours d'examen dans pre-allocate
// resReqs est l'ensemble des alternatives de réquisition pour a

    UList result = ∅ ;
    POUR TOUTE EntityList resReq DANS resReqs FAIRE {
        // construction d'une structure U pour l'activité a avec l'ensemble d'entités resReq
        U Uthis = new U(resReq) ;
        //   check-aic : test des ACTIVITY INCONSISTENCY CONDITIONS,
        //   y compris celles incluses dans une ACTIVITIES RESOURCES INCONSISTANT COMMITMENT
        SI Uthis . check-aic(a)  ALORS {
                // augmentation du U associé au nœud avec le U associé à l'activité a
                U Utmp = Unode . return-extension-with-alloc(Uthis) ;
                // check-rsvc : test des RESOURCE SHARING VIOLATION CONDITIONS
                // et des ACTIVITIES RESOURCES INCONSISTANT COMMITMENTS
                SI Utmp . check-rsvc(a) ALORS
                    // on renvoie la structure U associée à l'activité a
                    result . push(Uthis) ;
                }
        }
    }
return result ;
}
```

Ainsi, pour une activité, on retourne une liste de structures U_i. Chaque structure U_i correspond à une alternative de réquisition qui satisfait toutes les contraintes ; elle est construite avec les RESOURCES requises par l'activité, c'est-à-dire à l'exclusion des RESOURCES qui ne sont allouées qu'aux autres activités dans le jeu en examen. On rappelle que la PROCEDURE *pre-allocate* (qui a invoqué *get-specimens*) réalise alors trois sous-tâches :

- génération d'autant de copies du treillis en cours d'examen qu'il y a de structures U_i retournées ;
- dans chaque treillis, et dans le nœud (jeu) en examen ou sa copie, exploitation de la structure U_i d'abord pour valuer la structure V_i de la `CandidateInstruction` correspondant à l'activité, puis pour augmenter la structure U_N du jeu ;
- dans chaque treillis, propagation de la structure U_i vers les nœuds dégradés dont les premières activités sont justement celles qui ont déjà fait l'objet d'un *get-specimens* dans le jeu en examen.

La structure U correspondant à l'alternative de réquisition (q, o1,p2,s1,s2,t1,t2) est la suivante :

```
U1 : {    (  q  {          (OPERATED_OBJECT o1)  (P2 p2)  (R1a s1) (R1b s2) (R2 t1  t2) }   )
          (  o1{  (Q q)                          (P2 p2)  (R1a s1) (R1b s2) (R2 t1  t2) }   )
          (  p2{  (Q q)    (OPERATED_OBJECT o1)           (R1a s1) (R1b s2) (R2 t1  t2) }   )
          (  s1{  (Q q)    (OPERATED_OBJECT o1)  (P2 p2)          (R1b s2) (R2 t1  t2) }   )
          (  s2{  (Q q)    (OPERATED_OBJECT o1)  (P2 p2)  (R1a s1)         (R2 t1  t2) }   )
          (  t1{  (Q q)    (OPERATED_OBJECT o1)  (P2 p2)  (R1a s1) (R1b s2) (R2 t2) } )
          (  t2{  (Q q)    (OPERATED_OBJECT o1)  (P2 p2)  (R1a s1) (R1b s2) (R2 t1) } )
      }
```

La structure V_1 construite à partir de U_1 sera : `V1 : { (o1) (p2) (q) (s1,s2,t1,t2) }`

On examine dans les trois paragraphes suivants :

- la procédure d'augmentation d'une structure U (typiquement celle d'un nœud, U_N) par une autre (typiquement celle d'une activité, U_i) et par un simple ensemble de RESOURCES (typiquement celui issu d'une DETERMINATION) ;
- le comportement des METHODES *check-aic* et *check-rsvc* d'une structure U.

5.5.2.4.2.2 Augmentation d'une structure U par une autre

Ce sont les METHODES *add-allocation* (invoquée dans *add-and-propagate-allocation*, voir annexe 5) et *return-extension-with-alloc* (invoquée dans *get-specimens*, *cf.* § 5.5.2.4.2.1) qui réalisent cette tâche[54].
On les décrit ici en prenant pour exemple U_N (identique à U_1 si on suppose que U_1 correspond à la première activité du jeu) et la structure U_2 ci-dessous :

```
U2 : {    ( q'{       (OPERATED_OBJECT o1)  (P1 p1)    (P2 p2)    (R1a s1)) } )
          ( o1{  (Q q')                     (P1 p1)    (P2 p2)    (R1a s1) } )
          ( p1{  (Q q') (OPERATED_OBJECT o1)           (P2 p2)    (R1a s1) } )
          ( p2{  (Q q') (OPERATED_OBJECT o1)  (P1 p1)              (R1a s1) } )
          ( s1{  (Q q')(OPERATED_OBJECT o1)   (P1 p1)    (P2 p2)            } )
      }
```

La structure U_N est modifiée en U_N' :

```
UN':{ ( q  {        (OPERATED_OBJECT o1)          (P2 p2) (R1a s1) (R1b s2) (R2 t1  t2) } )
      ( o1{  (Q q q')                     (P1 p1) (P2 p2) (R1a s1) (R1b s2) (R2 t1  t2) } )
      ( p2{  (Q q q') (OPERATED_OBJECT o1)  (P1 p1) (R1a s1) (R1b s2)         (R2 t1  t2) } )
      ( s1{  (Q q q') (OPERATED_OBJECT o1)  (P1 p1) (P2 p2) (R1b s2)         (R2 t1  t2) } )
      ( s2{  (Q q)    (OPERATED_OBJECT o1)          (P2 p2) (R1a s1)         (R2 t1  t2) } )
      ( t1{  (Q q)    (OPERATED_OBJECT o1)          (P2 p2) (R1a s1) (R1b s2) (R2 t2) } )
      ( t2{  (Q q)    (OPERATED_OBJECT o1)          (P2 p2) (R1a s1) (R1b s2) (R2 t1) } )
      ( q'{           (OPERATED_OBJECT o1)  (P1 p1) (P2 p2) (R1a s1) } )
      ( p1{  (Q q')   (OPERATED_OBJECT o1)          (P2 p2) (R1a s1) } )
  }
```

Les trois règles suivantes sont en effet appliquées :
- il y a dans la structure U_N modifiée un et un seul ResourceUsageItem par instance de RESOURCE faisant l'objet d'un ResourceUsageItem dans U_1 ou dans U_2 ;
- il y a dans la structure U_N modifiée un et un seul OperationUsageItem par instance de OPERATION faisant l'objet d'un OperationUsageItem dans U_1 ou dans U_2 ;
- dans un ResourceUsageItem, portant sur la RESOURCE r, il y a un et un seul CoEngagementItem par CLASSE de RESOURCE ou d'OPERATION dont une INSTANCE est co-engagée avec r soit dans U_1 soit dans U_2 ;
- un OperationUsageItem, portant sur l'OPERATION q, il n'y a que les CoEngagementItem qu'il contient dans U_1 ou dans U_2 , puisque les deux instances d'OPERATION sont par construction différentes dans U_1 et U_2 ;
- dans un CoEngagementItem de r, portant sur la CLASSE C, il y autant d'INSTANCES de C qu'il y a de RESOURCES de cette CLASSE co-engagées avec r dans U_1 ou dans U_2.

5.5.2.4.2.3 Augmentation d'une structure U par un ensemble de ressources

La METHODE *return-extension-with-comb* (invoquée dans *get-specimens*, *cf.* § 5.5.2.4.3.1) augmente la structure U_i d'une activité avec un ensemble de RESOURCES E, en appliquant les trois règles suivantes :
- il y a dans la structure U_i modifiée un et un seul ResourceUsageItem par instance de RESOURCE faisant l'objet d'un ResourceUsageItem dans U_i avant l'augmentation, ou qui est un élément de E ;
- dans un ResourceUsageItem, portant sur la RESOURCE r, il y a un et un seul CoEngagementItem par CLASSE de RESOURCE ou d'OPERATION dont une INSTANCE est soit co-engagée avec r dans U_i soit un élément de E ;
- dans un CoEngagementItem de r, portant sur la CLASSE C, il y autant d'INSTANCES de C qu'il y a d'INSTANCES de C co-engagées avec r dans U_i ou d'INSTANCES de C dans E.

Ainsi, par exemple, avec E : {d}, où d est de CLASSE P1, et

```
U2 : {    ( q'{       (OPERATED_OBJECT o1)  (P1 p1)  (P2 p2)    (R1a s1)) } )
          ( o1{  (Q q')                     (P1 p1)  (P2 p2)    (R1a s1) } )
          ( p1{  (Q q') (OPERATED_OBJECT o1)         (P2 p2)    (R1a s1) } )
```

[54] Les deux METHODES diffèrent par le fait que la première modifie la structure U sur laquelle elle porte et ne retourne rien, alors que la seconde ne modifie pas la structure U sur laquelle elle porte et retourne une structure U nouvelle.

```
        (  p2{  (Q q') (OPERATED_OBJECT o1)  (P1 p1)            (R1a s1) }  )
        (  s1{  (Q q')(OPERATED_OBJECT o1)  (P1 p1)  (P2 p2)             }  )
    }
```

.. la structure U_2 devient :

```
U2 : {  (  q'{       (OPERATED_OBJECT o1)  (P1 p1  d)  (P2 p2)    (R1a s1)) }  )
        (  o1{  (Q q')                     (P1 p1  d) (P2 p2)    (R1a s1) }  )
        (  p1{  (Q q') (OPERATED_OBJECT o1)  (P1 d)   (P2 p2)    (R1a s1) }  )
        (  p2{  (Q q') (OPERATED_OBJECT o1)  (P1 p1  d)            (R1a s1) }  )
        (  s1{  (Q q')(OPERATED_OBJECT o1)  (P1 p1  d)  (P2 p2)             }  )
        (  d  {  (Q q') (OPERATED_OBJECT o1)  (P1 p1)  (P2 p2)    (R1a s1) }  )
    }
```

5.5.2.4.2.4 Cas des agrégats de ressources

Les règles énoncées dans les deux paragraphes ci-dessus sont valides dans le cas où les RESOURCES en jeu sont toutes des SINGLE RESOURCES. Elles doivent être étendues de la façon suivante dans le cas où des RESOURCES sont des AGGREGATED RESOURCES (agrégats de 'ressources éléments'). On les exprime pour le problème de l'augmentation d'une structure `U par un liste de` RESOURCES `E. La même extension est nécessaire, et immédiate, pour celui de l'`augmentation d'une structure `U par une autre.`

Règles :

- il y a dans la structure U_N modifiée un `ResourceUsageItem` par INSTANCE de RESOURCE faisant l'objet d'un `ResourceUsageItem` dans U_i avant l'augmentation, ou qui est un élément de E ;
- il y a en outre dans la structure U_N modifiée un `ResourceUsageItem` par ressource élément, à quelque profondeur que ce soit, dans un agrégat (AGGREGATED RESOURCE) faisant l'objet d'un `ResourceUsageItem` dans U_i avant l'augmentation, ou qui est un élément de E ;
- dans un `ResourceUsageItem`, portant sur la RESOURCE `r`, il y a un `CoEngagementItem` par CLASSE de RESOURCE ou d'OPERATION dont une INSTANCE est soit co-engagée avec r dans U_i soit un élément de E ;
- dans un `ResourceUsageItem`, portant sur la RESOURCE `r`, il y a en outre un `CoEngagementItem` par CLASSE de ressource élément, à quelque profondeur que ce soit, dans un agrégat faisant l'objet d'un `CoEngagementItem` dans le `ResourceUsageItem` ;
- dans un `CoEngagementItem` de `r`, portant sur la CLASSE `C`, il y autant d'INSTANCES de `C` qu'il y a d'INSTANCES de `C` co-engagées avec `r` dans U_i ou d'INSTANCES de `C` dans E, que ces INSTANCES de `C` soient des ressources élémentaires ou agrégées.

Exemple :
Soit E : {a}, où a est une HOMOGENEOUS AGGREGATED RESOURCE de CLASSE A caractérisée par sa CLASSE d'ELEMENTS R et un *effectif* de VALEUR *2*. Soit R1 et R2 deux *particularisations* terminales de R, dont les INSTANCES sont, respectivement {r1, r3} et {r2, r4}. Soit enfin {r1, r2} les ELEMENTS de a après son INSTANCIATION lors du DEVELOPPEMENT de l'OPERATION RESOURCE SPECIFICATION correspondante.
Soit d'autre part la structure U suivante, avant son augmentation par E.

`U : {  (  p  {  }  )  }`, où p est une SINGLE RESOURCE de CLASSE P.

La structure U augmentée est :

```
U : {  (  p  {  (A a)  (R1 r1)  (R2 r2)  }  )  // les CoEngagementItems ne portent pas sur R
       (  a  {  (P p)                    }  )  // a n'est pas dite co-engagée avec r1 ni r2
       (  r1{  (P p)                     }  )  // r1 n'est pas dite co-engagée avec a
       (  r2{  (P p)                     }  )
    }
```

5.5.2.4.2.5 Test de satisfaction des différents types de contraintes

La METHODE *check-aic* appliquée à une structure U pour une activité (voir les descriptions de la METHODE *get-specimens*) teste les ACTIVITY INCONSISTENCY CONDITIONS portant sur une CLASSE dont cette activité est une INSTANCE. Si l'ACTIVITY INCONSISTENCY CONDITION n'est pas incluse dans un ACTIVITIES RESOURCES INCONSISTANT COMMITMENT, *check-aic*

renvoie *faux* si elle seule est violée. Dans le cas contraire, elle renvoie *faux* si elle est violée et si toutes les autres contraintes incluses dans l'ACTIVITIES RESOURCES INCONSISTANT COMMITMENT sont violées.
La METHODE *check-rsvc* appliquée à une structure U pour un jeu d'activités fait de même pour les RESOURCE SHARING VIOLATION CONDITIONS.
On donne un exemple de chacun des trois types de contraintes, et on examine si elle est satisfaite par la tentative d'allocation représentée par les structures U_1, U_2 (supposées correspondre aux deux seules activités d'un jeu) et U_N' ci-dessus.

1) Les ACTIVITY INCONSISTENCY CONDITIONS :

Soit, par exemple, la contrainte :
{PRIMITIVE ACTIVITY SPECIFICATION ; ((P2 > *0*) (R2 > *1*)) ; allways_true },
exprimant qu'un SINGLE WORKER de CLASSE P2 ne peut jamais utiliser (quelle que soit l'activité et sans condition) plus d'une OPERATION RESOURCE de CLASSE R2.
Il faut vérifier le respect de cette contrainte successivement pour chaque activité du jeu en examen, c'est-à-dire ici sur les structures U_1 puis U_2 du § 5.5.2.4.2.1.
Pour que la contrainte soit violée par une structure U_i, trois conditions doivent être satisfaites :
- l'activité primitive correspondante est une INSTANCE directe ou indirecte de la CLASSE d'activité qui caractérise la contrainte, ce qui est vrai ici puisque PRIMITIVE ACTIVITY SPECIFICATION est la racine de la hiérarchie de toutes les activités primitives ;
- sa *holding-condition* est satisfaite, ce qui est évident ici (allways_true) ;
- tous les OCCURRENCE DOMAINS de l'OCCURRENCE CROSS DOMAIN sont observés (*cf.* § 4.2).

Un OCCURRENCE DOMAIN est observé si on peut construire à partir de la liste des ENTITES RESOURCES et OPERATION en tête des ResourceUsageItems, une liste d'INSTANCES de la CLASSE qui le caractérise ou d'une CLASSE descendante, liste dont la taille respecte la relation binaire (argumentée par la valeur entière) de l'OCCURRENCE DOMAIN. Pour une ENTITE en tête des ResourceUsageItems qui est un OPERATED OBJECT, on le retient si son *entité-ressource* est de la CLASSE qui caractérise l'OCCURRENCE DOMAIN ou d'une CLASSE descendante. Dans notre exemple, les OCCURRENCE DOMAINS (P2 > *0*) et (R2 > *1*) sont observés dans U_1 parce qu'on peut construire les listes (p2) et (t1 t2) respectivement[55].
Noter que puisque l'ACTIVITY INCONSISTENCY CONDITION est rencontrée dans l'activité correspondant à U_1, il n'est pas nécessaire de la tester dans l'activité correspondant à U_2 : tout jeu comprenant l'activité correspondant à U_1 est inconsistant.
Noter encore que la METHODE *holding-condition* peut être exploitée pour vérifier la condition de faisabilité de l'OPERATION en jeu, lorsque cette condition dépend des ressources allouées. On rappelle que la METHODE *condition-de-faisabilité* de l'OPERATION est testée lors de l'élagage des jeux possiblement inconsistants (voir § 5.5.1.2.1), c'est-à-dire avant l'allocation des ressources. Le test d'une ACTIVITY INCONSISTENCY CONDITION sera équivalent au test de la condition de faisabilité si la *holding-condition* (ne) repose (que) sur la condition de faisabilité et si tous les OCCURRENCE DOMAINS de l'OCCURRENCE CROSS DOMAIN sont nécessairement observés. A cette fin, on pourra déclarer, par exemple : ((OPERATION > *0*)), puisqu'une OPERATION est toujours instanciée dans la procédure d'allocation.

2) Les RESOURCE SHARING VIOLATION CONDITIONS :

Soit, par exemple, la contrainte :
{ P2 ; ((R > *2*)) ; allways_true },
exprimant qu'un PERFORMER de CLASSE P2 ne peut utiliser en même temps plus de 2 RESOURCES de CLASSE R (outils, par exemple). On donne aussi que les deux CLASSES R1 et R2 de l'exemple suivi dans cette section sont deux particularisations de R.
Une telle contrainte peut se trouver violée par une seule activité, ce qui justifierait de la tester sur les U_i correspondant aux activités. Cependant, une RESOURCE SHARING VIOLATION CONDITION sera normalement utilisée pour limiter la partageabilité d'une RESOURCE requise par plusieurs activités. En conséquence, on vérifie le respect de ce type de contraintes sur la structures U_N associée au nœud en examen.
Pour que la contrainte soit violée par la structure U_N, trois conditions doivent être satisfaites :
- un ResourceUsageItem doit porter sur une INSTANCE directe ou indirecte de P2, ce qui est vrai ici puisque p2 est une instance directe de P2 ;

[55] On peut dire de manière équivalente qu'un OCCURRENCE DOMAIN est observé si on peut trouver, dans un ResourceUsageItem quelconque, des CoEngagementItems (i) avec un identificateur (*cf.* § 5.5.2.1.4) d'une CLASSE identique à, ou descendant de, celle caractérisant l'OCCURRENCE DOMAIN et (ii) tel que la taille de l'union de leurs listes d'INSTANCES (*cf.* aussi § 5.5.2.1.4) respecte la relation binaire (argumentée par la valeur entière) de l'OCCURRENCE DOMAIN. Dans notre exemple, les OCCURRENCE DOMAINS (P2 > *0*) et (R2 > *1*) sont observés dans U_1 parce qu'on trouve les CoEngagementItems (P2 p2) et (R2 t1 t2) respectivement.

- sa *holding-condition* est vérifiée, ce qui est évident ici (`allways_true`) ;
- tous les OCCURRENCE DOMAINS de l'OCCURRENCE CROSS DOMAIN sont `observés` (*cf.* § 4.2).

Le seul OCCURRENCE DOMAIN (`R` > *2*) est `observé` si on peut trouver, dans le `ResourceUsageItem` portant ici sur `p2`, des `CoEngagementItems` (i) avec l'identificateur (*cf.* § 5.5.2.1.4) de la CLASSE `R` ou d'une classe descendant de `R` et (ii) tel que la taille de l'union de leurs listes d'INSTANCES (*cf.* aussi § 5.5.2.1.4) respecte la relation binaire (argumentée par la valeur entière) de l'OCCURRENCE DOMAIN. Dans notre exemple, les `CoEngagementItems` (`R1a s1`), (`R1b s2`) et (`R2 t1 t2`) permettent de construire la liste (`s1`, `s2`, `t1`, `t2`) de taille z=4, ce qui respecte la relation binaire « z > *2* ». La contrainte est donc violée.

3) Les ACTIVITIES RESOURCES INCONSISTANT COMMITMENTS

Soit, par exemple, la contrainte :

{ (RSVC, AIC), `allways_true` },

où RSVC est une RESOURCE SHARING VIOLATION CONDITION ainsi spécifiée :

{ `R1`; ((OPERATION > 1)) ; `allways_true` }

et AIC est une ACTIVITY INCONSISTENCY CONDITION ainsi spécifiée :

{ PRIMITIVE ACTIVITY SPECIFICATION ; ((`P1` > *0*)) ; `allways_true` }

Cette contrainte empêche de mettre une RESOURCE de CLASSE `R1` au service de plusieurs OPERATIONS quelconques, en même temps qu'un PERFORMER de CLASSE `P1` est alloué à une activité quelconque.

Cette contrainte est violée si et seulement sa *holding-condition* propre est vérifiée (ce qui est évident ici) et si RSVC et AIC sont violées en même temps.

AIC est testée successivement pour chaque activité du jeu en examen, c'est-à-dire ici sur les structures U_1 puis U_2, de la manière exposée plus haut. Elle se trouve violée non par la tentative d'allocation de la première activité du jeu, correspondant à U_1, mais plus tard par la tentative d'allocation de l'activité correspondant à U_2. L'OCCURRENCE DOMAIN (`P1` > *0*) est alors `observé` parce qu'on peut construire la liste (`p1`) à partir de la liste des ENTITES RESOURCES et OPERATION en tête des `ResourceUsageItems de` U_2, et que cette liste de taille z=1 respecte la relation binaire « z > *0* ».

RSVC est testée sur la structure `U associée au noeud`, à chacune de ses augmentations, d'abord par U_1 (ce qui résulte en U_N) puis par U_2 (ce qui résulte en U_N'). U_N ne montre pas de violation de RSVC : en effet, dans chacun des deux `ResourceUsageItems` portant sur `s1` et `s2` (INSTANCES de `R1`), on ne relève que le `CoEngagementItem` (`Q q`), c'est-à-dire qu'on ne peut y trouver des `CoEngagementItems` avec l'identificateur de la CLASSE OPERATION ou d'une sous-classe de OPERATION et tel que la taille z de l'union de leurs listes d'INSTANCES respecte la relation binaire (« z > *1* ») de l'OCCURRENCE DOMAIN. Au contraire, U_N' viole RSVC : en effet, dans le `ResourceUsageItems` portant sur `s1`, on relève le `CoEngagementItem` (`Q q q'`), ce qui permet de construire la liste (`q q'`) de taille z=2 qui respecte cette même relation binaire.

5.5.2.4.3 La conjonction de la détermination des réquisitions et du test des contraintes

5.5.2.4.3.1 Principe et procédure générale

On sait que, pour le jeu d'activités en cours d'examen dans un treillis, on cherche une ou plusieurs allocations valides pour chaque activité tour à tour (relire la PROCEDURE *pre-allocate*). Le contenu des deux paragraphes précédents supposait que la recherche d'allocations valides s'opérait en deux phases successives : d'abord l'établissement d'une ou plusieurs listes alternatives de RESOURCES requises, complètes au sens où elles comportent les RESOURCES issues des trois spécifications (OPERATED OBJECT SPECIFICATION, OPERATION SPECIFICATION et PERFORMER SPECIFICATION), puis le test de la satisfaction des contraintes d'allocation par chacune de ces listes complètes.

La METHODE *get-specimen*s est en réalité fondée sur une option différente, de nature heuristique : on tente de détecter les violations de contrainte le plus tôt possible. Plus précisément, on établit des listes alternatives partielles de RESOURCES requises à partir d'une des trois spécifications, et on teste les contraintes portant sur ces RESOURCES. Puis, s'il n'y a pas violation, on introduit dans toutes les alternatives les RESOURCES requises au titre d'une seconde spécification (et on teste à nouveau les contraintes), puis au titre de la troisième spécification.

On fait ainsi l'hypothèse qu'en moyenne sur l'examen d'un grand nombre d'activités, la somme des coûts des DETERMINATIONS partielles, successives et incrémentales et des tests sur les listes partielles qui en résultent sera inférieure au coût de l'unique DETERMINATION complète et des tests sur la liste complète qui en résultent. Ainsi, la METHODE *get-specimen*s est-elle celle de la figure ci-après.

La METHODE *get-trp-specimen*s procède de la même façon, mais en invoquant d'abord la METHODE *get-rp-specimen*s (qui retourne les alternatives de réquisitions valides à partir des seules OPERATION RESOURCE SPECIFICATIONS et de la PERFORMER SPECIFICATION), puis en augmentant chaque alternative retournée avec l'INSTANCE d'OPERATION qui résulte

de le DEVELOPPEMENT de l'OPERATION SPECIFICATION. Pour chaque alternative augmentée, on pratique les tests seulement sur les contraintes portant sur les RESOURCES concernées à ce stade.

La METHODE *get-rp-specimen*s procède de la même façon, mais en invoquant d'abord la METHODE *get-p-specimen*s (qui retourne les alternatives de réquisitions valides à partir de la seule PERFORMER SPECIFICATION), puis en augmentant chaque alternative retournée avec les alternatives de réquisitions valides retournées par la METHODE *get-r-specimens* à partir des seules OPERATION RESOURCE SPECIFICATIONS. Pour chaque alternative augmentée, on pratique les tests seulement sur les contraintes portant sur les RESOURCES concernées à ce stade.

On commence donc par le DEVELOPPEMENT de la PERFORMER SPECIFICATION et des OPERATION RESOURCE SPECIFICATIONS. On crée alors deux structures `U` qui ne contiennent que des `ResourceUsageItems` portant sur des PERFORMERS et, respectivement, des RESOURCES propres d'OPERATION. On augmente la première avec la seconde avant l'INSTANCIATION de l'OPERATION dans l'OPERATION SPECIFICATION. A ce moment, la structure `U` pour l'activité ne contient que des `ResourceUsageItems` portant sur des RESOURCES propres d'OPERATION et des PERFORMERS. Enfin, juste avant le DEVELOPPEMENT de l'OPERATED OBJECT SPECIFICATION, la structure `U` pour l'activité (trp-alloc dans l'algorithme) contient des `ResourceUsageItems` portant sur des RESOURCES propres d'OPERATION, sur des PERFORMERS et sur l'OPERATION.

```
UList SingleInstructionSet::get-specimens (U Unode) {
// this est doté du couple de PROPRIETES {a, V_a}, où a est une activité primitive et V_a sa structure V
// Unode : la structure U avec les RESOURCES déjà allouées aux activités précédant a dans le nœud
// en cours d'examen dans pre-allocate
    UList result = ∅ ;
    // trp-allocs = l'ensemble des structures U valides, partielles parce qu'en ne considérant que
    // l'OPERATION SPECIFICATION, les OPERATION RESOURCE SPECIFICATIONS et la PERFORMER SPECIFICATION
    UList trp-allocs = this . get-trp-specimens(Unode) ;
    OperatedObjectSpecification ooSpec = a . get-oo-spec() ;
    SI not(ooSpec)  ALORS return trp-allocs ;      // seulement admis avec une TIME GRANULATED OPERATION
    SINON {
        EntityList ooExpandedList = f(ooSpec)     // résultat de l'EXPANSION de la spécification
        // on ne cherche des allocations pour les OPERATED OBJECTS
        // que si on a trouvé des allocations conjointes valides des RESOURCES propres et des PERFORMERS
        SI trp-allocs non vide ALORS {
            SI ooSpec.requiresNonVoidAllocation() et ooExpandedList = ∅
            ALORS return result ; // ← ∅
            SINON  {    SI not(ooSpec.requiresNonVoidAllocation()) et ooExpandedList = ∅
                        ALORS return trp-allocs ; }
            // pour chaque structure U valide partielle, l'augmenter avec les réquisitions de
            // l'OPERATED OBJECT SPECIFICATION     et tester les contraintes
            POUR TOUT U trp-alloc DANS trp-allocs FAIRE {
                // ooReqList = la liste des alternatives de réquisition par l'OPERATED OBJECT SPECIFICATION
                // compte tenu de la fonction de sélection et de son éventuel argument
                EntityListList ooReqList = g(ooExpandedList) ;  // g dépend de la fonction de sélection
                                // et de la disponibilité propre des objets opérés vus comme des ressources
                boolean ok = false ;
                TANT QUE ok = false ET ooReqList non vide FAIRE {
                    EntityList ooReq = ooReqList . pop() ;
                    // augmentation de la structure U pour l'activité avec l'ensemble d'entités ooReq
                    U Uthis = trp-alloc . return-extension-with-comb(ooReq) ;
                    //   check-aic : test des ACTIVITY INCONSISTENCY CONDITIONS
                    //   y compris celles incluses dans une ACTIVITIES RESOURCES INCONSISTANT COMMITMENT
                    SI Uthis . check-aic(a)  ALORS {
                        // augmentation d'une copie du U associé au nœud avec le U pour l'activité  a
                        U Utmp = Unode . return-extension-with-alloc(Uthis) ;
                        // check-rsvc : test des RESOURCE SHARING VIOLATION CONDITIONS
                        // et des ACTIVITIES RESOURCES INCONSISTANT COMMITMENTS
                        SI Utmp . check-rsvc(a) ALORS {
                            // on retient Uthis (qui sera effectivement associée à l'activité dans pre-allocate)
                            result . push(Uthis) ;
                            // on arrête (ok = true) si la fonction de sélection le justifie (cf. § 5.5.2.4.3.2)
                        }
                        SINON {//   violation par check-rsvc
                            // on arrête la recherche (ok = true) si la fonction de sélection le justifie
                        }
                    }
                    SINON {//   violation par check-aic
                        // on arrête la recherche (ok = true) si la fonction de sélection le justifie }
                }   // fin examen une alternative de ooReq
            }   // fin examen une allocation valide partielle de OPERATION et PERFORMER
        }   // fin SI trp-allocs non vide
    } ;  // fin 'ooSpec non null'
return result ; }
```

5.5.2.4.3.2 Retour sur la détermination : le rôle des fonctions de sélection

On décrit ici la prise en compte de la spécificité de chaque FONCTION de sélection -*any*(n), *all*, *max*, *disj* et *sets*(n)- lors de l'établissement de la liste des alternatives de réquisition partielles pour chaque spécification : OPERATED OBJECT SPECIFICATION, OPERATION SPECIFICATION et PERFORMER SPECIFICATION. On spécialise donc ce qui est représenté de manière générale dans l'algorithme de *get-specimens* (et qui le serait de la même façon dans ceux des autres METHODES *get-?-specimens*) par la FONCTION *g* et le commentaire « on arrête la recherche si la fonction de sélection le justifie ». La description qui suit est valide quelle que soit la spécification à laquelle s'applique la FONCTION de sélection. On désigne par Ue la structure U pour l'activité telle qu'elle résulte des spécifications déjà examinées. On désigne par Unode la structure U courante pour le jeu d'activités.

1) *any*(n) : on peut allouer à l'activité indifféremment n'importe quel sous-ensemble de n éléments disponibles pris dans la liste d'INSTANCES résultant de l'EXPANSION de la spécification (ooExpandedList dans l'algorithme).
On examine tour à tour ces sous-ensembles d_i. On retient le premier tel que, ni l'augmentation de la structure Ue avec d_i, ni l'augmentation de Unode avec Ue, ne violent aucune contrainte.

2) *all* : il faut allouer à l'activité toutes les RESOURCES dont la liste est retournée par l'EXPANSION de la spécification, à condition pour chacune qu'elle soit disponible.
On examine cet unique ensemble d. On le retient si, ni l'augmentation de Ue avec d, ni l'augmentation de Unode avec Ue, ne violent aucune contrainte.

3) *max* : on souhaite allouer le plus possible d'éléments parmi l'ensemble D des RESOURCES disponibles, dont la liste est retournée par l'EXPANSION de la spécification. Soit N la taille de D.
On considère successivement les sous-ensembles de D de taille N, puis ceux de taille N-1, etc. et enfin ceux de taille 1. Si, lorsqu'on considère la taille k, on trouve un sous-ensemble d de D tel que ni l'augmentation de Ue avec d, ni l'augmentation de Unode avec Ue, ne violent aucune contrainte, alors on retient d et on n'examine ni les autres sous-ensembles de taille k ni les sous-ensembles de taille inférieure à k.

4) *disj* : chaque RESOURCE de la liste retournée par l'EXPANSION de la spécification constitue une alternative d'allocation. On veut soumettre ces alternatives d'allocation à la procédure de choix (section 5.5.3).
On examine tour à tour les sous-ensembles d_i qui ne contiennent qu'un seul élément. On retient tous les sous-ensembles tels que, ni l'augmentation de Ue avec d_i, ni l'augmentation de Unode avec Ue, ne violent aucune contrainte.

5) *sets*(n) : soit D l'ensemble des RESOURCES disponibles, dont la liste est retournée par l'EXPANSION de la spécification, et N la taille de D. On souhaite soumettre à la procédure de choix (section 5.5.3) autant d'alternatives d'allocation (sous la forme d'un sous-ensemble de D) qu'il y a de manière de tirer n éléments dans les k *particularisations* terminales de la CLASSE de RESOURCE en question.
On examine tour à tour les « profils » de tirage des n éléments. Par exemple, si n=2 et k=2, on peut tirer les 2 éléments dans une *particularisation* ou dans l'autre, ou bien tirer un élément dans chaque *particularisation*. Pour un profil déterminé, on construit la liste des tirages respectant le profil. Par exemple, si on en est à tirer les 2 éléments dans une seule *particularisation* qui possède trois INSTANCES i_1, i_2 et i_3, alors il y a trois tirages possibles : (i_1, i_2), (i_1, i_3) et (i_2, i_3). On examine ensuite chaque tirage d_i, et on retient le premier tel que, ni l'augmentation de Ue avec d_i, ni l'augmentation de Unode avec Ue, ne violent aucune contrainte. On procède ainsi pour chaque profil de tirage.

5.5.2.4.4 La continuation d'une activité suspendue en cours d'exécution

Toute spécification de RESOURCES peut requérir qu'en cas de reprise d'une activité suspendue (c'est-à-dire dont la *situation* de l'OPERATION en jeu est suspended), ce sont les ou des RESOURCES qui étaient allouées antérieurement qui doivent être à nouveau allouées, en continuation. Les commentaires sur la PROPRIETE *continuation*, en section 4.4, mentionnent des règles de continuation d'allocation pour chaque spécification de RESOURCES. Noter cependant que l'OPERATION suspended reste obligatoirement en vigueur à la reprise.
Les METHODES *get-p-specimens*, *get-r-specimen*s et *get-specimens* codent ces règles respectivement pour une PERFORMER SPECIFICATION, une OPERATION RESOURCE SPECIFICATION et une OPERATED OBJECT SPECIFICATION. En règle générale, lorsque la VALEUR de *continuation* est new, l'allocation est raisonnée comme si l'activité venait d'être ouverte. Lorsque la VALEUR est différente de new, la METHODE n'opère pas un nouveau DEVELOPPEMENT de la spécification : elle construit une liste de RESOURCES à partir de celles qui étaient allouées à l'INSTANT même de l'interruption (dans la structure V de l'activité, c'est le COMPOSANT correspondant à la spécification), ou bien à partir de la liste résultant de la première EXPANSION de la SPECIFICATION D'ENSEMBLE D'ENTITES caractérisant la spécification de RESOURCES. Cette liste vient alors augmenter la structure Ue pour l'activité, telle qu'elle résulte des

spécifications déjà examinées. On teste alors le respect des contraintes d'allocation par Ue et par la structure Unode associée au jeu et augmentée de Ue.

5.5.3 Le choix d'un jeu consistant (M3*)*

L'ensemble de jeux consistants établi par la METHODE *set-all-feasible-alternatives* (phase M2 du mécanisme MAKING INSTRUCTION LIST) est une structure InstructionPackList (*cf.* § 5.5.2), c'est-à-dire une liste de PAKS OF EXECUTABLE INSTRUCTIONS.
Il s'agit de sélectionner un seul de ces jeux sur la base des PREFERENCE RULES (règles de préférence).
(*à développer*)

5.6 Le mécanisme de mise en œuvre des opérations

5.6.1 Evénements, processus et procédures

La mise en œuvre des OPERATIONS est provoquée par la PROGRAMMATION, en fin du mécanisme MAKING INSTRUCTION LIST et en l'INSTANT courant, d'un EVENEMENT standard : la *particularisation* ACTING INSTRUCTION LIST EVENT, caractérisée par le couple { ACTING INSTRUCTION LIST PROCESS, *procédure-exécution*}. ACTING INSTRUCTION LIST PROCESS est une *particularisation* standard de PROCESSUS PONCTUEL. Le programme de sa METHODE *procédure-exécution* consiste simplement à invoquer la METHODE *act-instruction-list* de l'OPERATING SYSTEM.

Le programme de *act-instruction-list* correspond entièrement au mécanisme ACTING INSTRUCTION LIST. Il opère sur un PACK OF EXECUTABLE INSTRUCTIONS. Pour chaque INSTRUCTION du PACK, un EVENEMENT standard (PROCEED OPERATION EVENT) est programmé à l'INSTANT courant, qui porte sur une *particularisation* standard de PROCESSUS CONTINU, dédiée à la réalisation d'une OPERATION.

```
ACTING INSTRUCTION LIST EVENT
→ ACTING INSTRUCTION LIST PROCESS. procédure-exécution {
        InstructionPack pack =  OPERATING SYSTEM . instruction-to-act ;
        ACTING INSTRUCTION LIST(pack) :
        OPERATING SYSTEM . act-instruction-list(pack) {
            inCourseOtList  les opérations en cours ;
            pour toute instruction dans pack :
                si l' OPERATION ot de l'instruction n'est pas dans inCourseOtList {
                    INSTANCIATION de PROCEED OPERATION cp ;
                    PROGRAMMATION PROCEED OPERATION EVENT(cp)}
                sinon {
                    ot . situation = suspended ;
                    cp . procédure-arrêt () ; } // cp : le processus qui gère ot
            }
        }
```

Le PROCESSUS CONTINU créé est une INSTANCE de la *particularisation* PROCEED-OPERATION :

- lorsque le *degré-de-progression* de l'OPERATION est *0*, l'EVENEMENT PROCEED OPERATION EVENT lui soumet des directives correspondant aux METHODES *procédure-initialisation* puis *procédure-poursuite* ;
- lorsque le *degré-de-progression* de l'OPERATION est supérieur à *0*, l'EVENEMENT lui soumet une unique directive correspondant à la METHODE *procédure-poursuite*.

```
OCCURRENCE PROCEED OPERATION EVENT (cp) :
    → OPERATION ot = cp . processed-object() ;
      si ot . degré-de-progression = 0
      alors cp . procédure-initialisation() :
          initialize-proceed-operation() {
              ot . situation = executing ; }
    → à chaque pas de cp
          faire cp . procédure-poursuite()
```

Le PROCESSUS PROCEED-OPERATION porte sur l'OPERATION, renvoyée par la METHODE *processed-object*. C'est dire que son effet porte directement sur les PROPRIETES de l'OPERATION, essentiellement *situation*, *degré-de-progression*, *date-début* et *date-fin*.

La progressivité d'une OPERATION, matérialisée par l'évolution du *degré-de-progression*, est définie comme l'étalement dans le temps de son effet sur la ou les ENTITES qui en sont l'objet. On souhaite représenter deux manières d'étaler cet effet :

- l'effet est d'abord réalisé complètement sur une partie de l'objet, puis complètement sur une autre, et ainsi de suite jusqu'à ce que toutes les parties aient subi l'effet complet.
  Cette manière présente deux variantes :
  - les parties de l'objet sont identifiées comme des ensembles d'ENTITES, elles-mêmes ELEMENTS à une quelconque profondeur de l'objet de l'OPERATION ; on instancie alors une UNIT-GRANULATED OPERATION ;
  - les parties de l'objet représentent une portion de la VALEUR d'une PROPRIETE de l'objet de l'OPERATION ; on instancie alors une QUANTITY-GRANULATED OPERATION ;
- l'effet est partiellement réalisé sur tout l'objet, puis plus complètement réalisé sur tout l'objet, et ainsi de suite jusqu'à ce que l'objet ait subi le plein effet de l'OPERATION ; on instancie alors une TIME-GRANULATED OPERATION.

Lors de l'INSTANCIATION du PROCESSUS, on donne à son ATTRIBUT *pas* la VALEUR de la PROPRIETE *pas* de l'OPERATION. De plus, on assigne à sa METHODE *procédure-initialisation* la PROCEDURE standard *initialize-proceed-operation*, qui donne notamment la VALEUR `executing` à la *situation* de l'OPERATION et de l'INSTRUCTION en jeu.

A sa METHODE *procédure-poursuite* on assigne une PROCEDURE standard, différente selon le *mode-de-progression* de l'OPERATION. On rappelle que cette PROCEDURE, quelle qu'elle soit, est invoquée à chaque *pas* de PROCEED-OPERATION.

- Si l'OPERATION est une UNIT-GRANULATED OPERATION, cette PROCEDURE, dénommée *full-effect-on-units*, enchaîne un certain nombre d'INVOCATIONS de la METHODE *procédure-transition-état* de l'OPERATION (*cf.* section 4.7). Ce nombre est la *vitesse* de l'OPERATION.

```
→ {   à chaque pas de PROCEED-OPERATION cp
      faire cp . procédure-poursuite() :
          // si l'OPERATION en jeu est une UNIT-GRANULATED OPERATION :
          full-effect-on-units(cp) {
              OPERATION ot = cp . processed-object() ;
              total_n : nombre total d'unités à opérer ;
              n = ot . vitesse ;
              pour les n premières 'unité' non encore opérées :
                  ot . procédure-transition-état(unité) ;
              augmenter ot . degré-de-progression de n / total_n;
              si ot . degré-de-progression = 1 alors cp . procédure-arrêt ; }
  }
```

Chaque INVOCATION de la *procédure-transition-état* porte sur une INSTANCE, dite 'unité', de la *classe-entité* de l'OPERATION. Cette 'unité' est tirée d'une liste d'INSTANCES, initialisée avec toutes les 'unités' qui sont les OPERATED OBJECTS de l'INSTRUCTION s'ils sont de la *classe-entité*, ou sinon qui sont ELEMENTS à une profondeur quelconque des *entités-ressources* de tous les OPERATED OBJECTS de l'INSTRUCTION[56]. A chaque INVOCATION de *full-*

[56] On verra (*cf.* § 5.6.2) que cette liste est l'union des *liste-entités-non-opérées* des OPERATED OBJECTS de l'INSTRUCTION.

effect-on-units, on lui retire les 'unités' sur lesquelles a porté la *procédure-transition-état* et le *degré-de-progression* de l'OPERATION est augmenté.
Si l'opération est un transfert, on rappelle (voir § 4.7.3) que les 'unités' sur lesquelles porte la *procédure-transition-état* sont par spécification les ELEMENTS immédiats des *entités-ressources* de tous les OPERATED OBJECTS de la source de l'INSTRUCTION. La *procédure-transition-état* prédéfinie retire les 'unités' de la liste des ELEMENTS de la source et les ajoute à la liste des ELEMENTS de la destination de l'instruction. Le nombre de ces 'unités' est spécifié dans la TRANSFER INFORMATION qui est attachée à l'INSTRUCTION. La présence d'un nombre suffisant d'ELEMENTS dans la source est vérifiée dans la procédure générale d'établissement des jeux consistants (§ 5.5.2). Lorsque les ENTITES en jeu sont des RESSOURCES à capacité, les capacités en sortie (pour la source) et en entrée (pour la destination) sont aussi vérifiées dans ladite procédure. La capacité en sortie est égale, par défaut, au nombre d'ELEMENTS des objets opérés de la source.
Si c'est une QUANTITY-GRANULATED OPERATION, la PROCEDURE, dénommée *full-effect-on-quantity*, augmente à chacune de ses INVOCATIONS le *degré-de-progression* de l'OPERATION, en fonction de la *vitesse*. La PROCEDURE *full-effect-on-quantity* gère aussi une liste d'INSTANCES et initialisée avec les *entités-ressources* de tous les OPERATED OBJECTS de l'INSTRUCTION (lesquelles sont par construction des INSTANCES de la *classe-entité* de l'OPERATION). Dans le cas des opérations en général, à chaque INVOCATION de *full-effect-on-quantity*, si l'ETAT du premier élément de la liste (l'élément courant) n'a pas encore été changé par la *procédure-transition-état*, celle-ci est invoquée sur l'élément. Si l'ETAT a été changé mais sans que toute la VALEUR de la PROPRIETE de l'élément soit censée avoir fait l'objet de l'OPERATION, la *procédure-transition-état* n'est pas invoquée et seul le *degré-de-progression* est augmenté. L'élément sera retiré de la liste lorsque suffisamment d'invocations de full-effect-on-quantity auront été faites pour que toute la valeur de la propriété de l'élément soit censée avoir fait l'objet de l'operation.

```
→ {   à chaque pas de PROCEED-OPERATION cp
      faire cp . procédure-poursuite() :
          // si l'OPERATION est une QUANTITY-GRANULATED OPERATION :
          full-effect-on-quantity(cp) {
              OPERATION ot = cp . processed-object() ;
              total_quantity : quantité totale à opérer ;
              entity : la première entité-ressource non totalement opérée ;
              si entity n'est pas du tout opérée ou si ot est une OPERATION permanente alors
                  ot . procédure-transition-état(entity) ;
              augmenter ot . degré-de-progression de ot . vitesse / total_quantity ;
              si ot . degré-de-progression = 1 alors cp . procédure-arrêt ; }
}
```

Dans le cas des opérations de tansfert, une fraction du transfert est opérée lors de chaque *pas*, en touchant les objets tour à tour. Toucher les objets signifie ici modifier la VALEUR de la propriété qui est l'*identificateur* de la TRANSFER INFORMATION de l'INSTRUCTION. On diminue la VALEUR dans l'objet source et on augmente la VALEUR dans l'objet destination. La différence de VALEUR est égale à la VALEUR de la PROPRIETE *quantité* de la TRANSFER INFORMATION. Dans un *pas* donné, il est possible que la *quantité* soit prise partiellement dans un objet source et dans un ou plusieurs autres objets source pour le complément. De même la quentité transférée peut être dirigée vers un objet destination si sa capacité le permet, ou vers plusieurs si nécessaire.

- Si c'est une TIME-GRANULATED OPERATION, la PROCEDURE, dénommée *step-effect-on-object*, opère comme *full-effect-on-quantity*, au calcul de l'augmentation du *degré-de-progression* près (*cf.* section 4.7).

On voit que ce n'est qu'indirectement que l'INSTANCE de PROCEED-OPERATION a un effet sur le CONTROLLED SYSTEM : le mécanisme de SIMULATION invoque d'abord sa METHODE *procédure-poursuit*e, laquelle invoque la METHODE *procédure-transition-état* de l'OPERATION sur laquelle il porte.

```
→ {   à chaque pas de PROCEED-OPERATION cp
      faire cp . procédure-poursuite() :
          // si l'OPERATION est une TIME-GRANULATED OPERATION :
          step-effect-on-object(cp) {
              OPERATION ot = cp . processed-object() ;
              entity : la première entité-ressource non totalement opérée ;
              si entity n'est pas du tout opérée ou si ot est une OPERATION permanente alors
                  ot . procédure-transition-état(entity) ;
              augmenter ot . degré-de-progression de ot . vitesse ;
              si ot . degré-de-progression = 1 alors cp . procédure-arrêt ; }
}
```

La METHODE *procédure-arrêt* a toujours comme VALEUR une référence à la PROCEDURE standard *stop-proceed-operation*. Elle est invoquée dans la METHODE *procédure-poursuite*, dès que le *degré-de-progression* atteint *1* ou bien dès que la date de fin de l'activité primitive a été dépassée (ceci est notamment une origine normale de la terminaison d'une OPERATION permanente). Elle donne la VALEUR `terminated` à la *situation*. Puis *stop-proceed-operation* déclenche le mécanisme MAKING INSTRUCTION LIST, parce que la fin de l'exécution de l'OPERATION libère les RESOURCES qui lui étaient allouées. Les OPERATIONS en jeu dans les autres INSTRUCTIONS générées au titre du même ACTIVITIES-RESOURCES BLOCK seront suspendues par le mécanisme ACTING INSTRUCTION LIST (et les PROCESSUS correspondants stoppés) si elles ne sont pas à nouveau proposées par le MAKING INSTRUCTION LIST. En attendant cet examen, les effets de ces opérations sont poursuivis.

Les différentes INSTANCES de PROCEED-OPERATION créées à partir d'un PACK OF EXECUTABLE INSTRUCTIONS représentent des PROCESSUS CONTINUS qui se déroulent en même temps. Leur synchronisation est assurée par le mécanisme de SIMULATION (*cf.* section 5.1).

5.6.2 Structures de données pour gérer la progressivité des opérations

La gestion de la progressivité d'une OPERATION sur son objet consiste à identifier à tout INSTANT quelle partie de l'objet a changé d'ETAT, pour les *mode-de-progression* `unit` et `quantity`, ou bien à quel degré l'effet de l'OPERATION a été réalisé, pour le *mode-de-progression* `time`. Cette gestion autorise deux fonctionnalités :

- la synchronisation des effets de plusieurs OPERATIONS sur le même objet : chaque OPERATION réalise son effet en considérant des ETATS avant effet plus réalistes que si les effets des autres OPERATIONS étaient seulement soit nullement soit totalement réalisés ;
- la scission de l'objet d'une activité, lorsque deux parties de l'objet présentent des ETATS suffisamment différents pour qu'on puisse ou doive différencier leurs évolutions futures ou pour qu'il soit justifié de leur appliquer des conduites différentes. Une telle scission est par exemple justifiée lorsqu'une OPERATION sur un objet P : P' ∪ P'', a été suspendue suffisamment longtemps une fois P' opéré pour que les ETATS de P' et P'' soient différents une fois P'' opéré ou si l'OPERATION n'est pas reprise sur P''. On remarque que cette fonctionnalité n'est accessible que pour les *mode-de-progression* `unit` et `quantity`, qui permettent d'identifier une partie opérée et une partie non opérée.

A ces fins, on complète la CLASSE OPERATED OBJECT, déjà présentée sommairement en section 4.2.5.1, et la structure de donnée associée. On rappelle qu'un OPERATED OBJECT est élément de la liste renvoyée par le DEVELOPPEMENT de l'OPERATED OBJECT SPECIFICATION d'une PRIMITIVE ACTIVITY SPECIFICATION en examen (*cf.* § 5.5.2.4.1), dénommée alors une INSTRUCTION, et qu'il possède une PROPRIETE *entité-ressource*, dont la VALEUR est la référence à une ENTITE sur ou dans laquelle doit se réaliser l'effet de l'OPERATION.
On lui ajoute d'abord les PROPRIETES suivantes, initialisées lors de l'INVOCATION de la METHODE *initialize-proceed-operation* :

- l'*entité-ressource-mémoire* mémorise l'ETAT de l'*entité-ressource* juste avant le début de réalisation de l'effet de l'OPERATION. C'est une référence à une ENTITE, instanciée lors de l'INVOCATION de la METHODE *initialize-proceed-operation* par copie de l'*entité-ressource*. Cette PROPRIETE n'est exploitée que lors de la réalisation d'une QUANTITY-GRANULATED OPERATION ou d'une TIME-GRANULATED OPERATION.
- la liste des ELEMENTS, dénommée par la suite *liste-sub-operated-objects*, n'a une VALEUR que si l'*entité-ressource* est une POPULATION MULTIPLE ou une LOCATION. Elle contient les références à des OPERATED OBJECTS dont les *entités-ressources* sont les POPULATIONS (resp. LOCATIONS) qui sont les ELEMENTS de la POPULATION MULTIPLE (resp. LOCATION). Ces OPERATED OBJECTS-là sont instanciés lors de l'INSTANCIATION de celui-ci (this). La *liste-sub-operated-objects* est vide notamment si l'*entité-ressource* est une POPULATION SIMPLE (resp. PRIMITIVE LOCATION), ou si ce n'est ni une POPULATION ni une LOCATION.

- *opéré-totalement* possède la VALEUR booléenne *vrai* ou *faux*. Si la *liste-sub-operated-objects* est vide, la VALEUR est *vrai* si la *liste-entités-non-opérées* est vide, ou si la *quantité-non-opérée* est nulle, ou si la *proportion-non-opérée* est nulle, et *faux* sinon. Si la *liste-sub-operated-objects* n'est pas vide, la VALEUR est *vrai* si tous les OPERATED OBJECTS de cette liste ont une VALEUR *vrai* pour *opéré-totalement*, et *faux* sinon.

Les PROPRIETES suivantes n'ont de VALEUR que si la *liste-sub-operated-objects* est vide.

- la *liste-entités-non-opérées* comprend les ENTITES qui n'ont pas encore été l'argument d'une INVOCATION de la *procédure-transition-état*. Dans le cas d'une QUANTITY- ou TIME-GRANULATED OPERATION, la VALEUR initiale de cette PROPRIETE est une liste d'un seul élément : l'*entité-ressource* de l'OPERATED OBJECT (qui est une INSTANCE de la *classe-entité* de l'OPERATION en jeu dans l'INSTRUCTION). Dans le cas d'une UNIT-GRANULATED OPERATION, la VALEUR initiale est la liste renvoyée par l'INVOCATION, sur l'*entité-ressource*, de *get-involved-entities* argumentée par la *classe-entité* de l'OPERATION (*cf.* § 4.2.5.1).
- la *liste-entités-opérées* comprend les ENTITES qui ont figuré dans la liste ci-dessus et sur lesquelles a été réalisé ensuite l'effet de l'OPERATION.
- l'*effectif-non-opéré* n'est exploité que lors de la réalisation d'une UNIT-GRANULATED OPERATION et quand l'*entité-ressource* est une POPULATION SIMPLE. Il est initialisé avec l'*effectif* de la POPULATION SIMPLE.
- l'*effectif-opéré* a pour VALEUR, à tout INSTANT, le nombre d'exemplaires de l'*ensemble-représentatif* censés avoir été complètement opérés.
- le *nb-éléments-représentatifs-non-opérés* n'est exploité que lors de la réalisation d'une UNIT-GRANULATED OPERATION et quand l'*entité-ressource* est une POPULATION SIMPLE. Pour une POPULATION SIMPLE HOMOGENE, il est initialisé avec la taille de la *liste-entités-non-opérées*, c'est-à-dire *1*. Pour une POPULATION SIMPLE HETEROGENE, il est initialisé avec la somme des éléments de la *liste-de-représentativité*.
- le *nb-éléments-représentatifs-opérés* a pour VALEUR, à tout INSTANT, le nombre d'éléments censés être opérés dans l'exemplaire de l'*ensemble-représentatif* censé être en cours d'opération.
- la *quantité-opérée* n'est exploitée que lors de la réalisation d'une QUANTITY-GRANULATED OPERATION. Elle a pour VALEUR, à tout INSTANT, le produit du nombre d'INVOCATIONS de la *procédure-transition-état* par la *vitesse* de l'OPERATION. En d'autres termes, c'est la quantité de la PROPRIETE en jeu de l'objet qui est censée avoir changé d'ETAT. On rappelle que la *vitesse* est une quantité absolue opérée à chaque *pas* de l'OPERATION.
- la *quantité-non-opérée* a pour VALEUR, à tout INSTANT, la quantité de la PROPRIETE en jeu de l'objet qui est censée ne pas avoir changé d'état.
- la *proportion-opérée* n'est exploitée que lors de la réalisation d'une TIME-GRANULATED OPERATION. Elle a pour VALEUR, à tout INSTANT, le produit du nombre d'INVOCATIONS de la *procédure-transition-état* par la *vitesse* de l'OPERATION. En d'autres termes, c'est la proportion de l'effet censée être atteinte sur l'objet. On rappelle que la *vitesse* est une proportion de l'effet réalisé à chaque *pas* de l'OPERATION.
- la *proportion-non-opérée* a pour VALEUR, à tout INSTANT, la proportion de l'effet encore non réalisée sur l'objet.

On ajoute ensuite la METHODE *current-operated-object*. Appliquée a une INSTANCE d'OPERATED OBJECT `o`, elle renvoie `null` si *opéré-totalement* est *vrai*. Sinon, elle renvoie `o` si la *liste-sub-operated-objects* est vide. Si celle-ci n'est pas vide, la METHODE s'applique récursivement à chaque ELEMENT. Elle s'arrête au premier sur lequel elle renvoie une ENTITE qui n'est pas `null`. Alors `o` . *current-operated-object* renvoie cette ENTITE.

On peut maintenant préciser que la METHODE *current-operated-object* (*cf.* section 4.4) appliquée à une INSTANCE `s` d'OPERATED OBJECT SPECIFICATION, examine chaque OPERATED OBJECT de sa *liste-opérée* et s'arrête au premier tel que la METHODE *current-operated-object* appliquée à cet élément renvoie une ENTITE qui n'est pas `null`. Alors `s` . *current-operated-object* renvoie cette ENTITE.

5.6.2.1 Exploitation des structures pour la réalisation de l'effet

Cas des UNIT-GRANULATED OPERATIONS :

On rappelle que lors de chaque INVOCATION de *full-effect-on-units*, on applique la *procédure-transition-état* successivement à `n` éléments, où `n` est la *vitesse* de l'OPERATION. On précise ici comment sont déterminés ces `n` éléments, et comment la *procédure-transition-état* leur est appliquée, selon la CLASSE des *entités-ressources* des OPERATED OBJECTS de l'INSTRUCTION en cours de mise en œuvre.

Soit *rem_n* le nombre d'éléments sur lesquels la *procédure-transition-état* n'a pas encore été invoquée, dans une INVOCATION de *full-effect-on-units*. La valeur *rem_n* est initialisée à `n` et doit décroître vers *0*.

Tant que *rem_n* est positif, et que *current-operated-object* appliquée à l'OPERATED OBJECT SPECIFICATION renvoie un OPERATED OBJECT `current` (i.e. il reste un OPERATED OBJECT tel que *opéré-totalement* est *faux*), on fait :

- Si l'*entité-ressource* de `current` n'est pas une POPULATION (*i.e.* son *effectif-non-opéré* n'a pas de VALEUR significative) :

z ← la taille de la *liste-entités-non-opérées* de `current`,

(i) si *rem_n* est inférieur à *z* :
. la *procédure-transition-état* est appliquée aux *rem_n* premiers éléments de la *liste-entités-non-opérées*
. après chaque application, l'élément est transféré vers la *liste-entités-opérées* de `current`.
. puis *rem_n* devient nul.

(ii) si *rem_n* est supérieur ou égal à *z* :

. on invoque la *procédure-transition-état* sur les *z* éléments de la *liste-entités-non-opérées*,
. après chaque application, l'élément est transféré vers la *liste-entités-opérées*. Alors *opéré-totalement* devient *vrai* pour `current`,
. *rem_n* est diminué de *z*.

▪ Si l'*entité-ressource* de `current` est une POPULATION SIMPLE HOMOGENE, que celle-ci soit ou non élément d'une POPULATION MULTIPLE (voir ci-dessus la définition de la METHODE *current-operated-object*),

On considère l'unique élément de la *liste-entités-non-opérées* de `current`.
On invoque la *procédure-transition-état* sur cet élément si et seulement si l'*effectif-opéré* est nul. Puis,

(i) si *rem_n* est inférieur à l'*effectif-non-opéré* :
. l'*effectif-non-opéré* est diminué, et l'*effectif-opéré* augmenté, de *rem_n*,
. puis *rem_n* devient nul.

(ii) si *rem_n* est supérieur ou égal à l'*effectif-non-opéré* :
. *rem_n* est diminué de *effectif-non-opéré*,
. puis l'*effectif-non-opéré* devient nul et l'*effectif-opéré* devient l'*effectif* de la POPULATION. Alors *opéré-totalement* devient *vrai* pour l'OPERATED OBJECT en cours d'opération.

▪ Si l'*entité-ressource* de `current` est une POPULATION SIMPLE HETEROGENE, que celle-ci soit ou non élément d'une POPULATION MULTIPLE,

z ← le *nb-éléments-représentatifs-non-opérés* de `current`.

(i) si *rem_n* est inférieur à *z* (l'exemplaire, celui en cours d'opération, de l'*ensemble-représentatif* de la POPULATION sera censé ne pas être complètement opéré après cette INVOCATION de *full-effect-on-units*) :
. si et seulement si l'*effectif-opéré* est nul, on invoque la *procédure-transition-état* sur certains éléments de la *liste-entités-non-opérées* de cet OPERATED OBJECT : en les examinant à partir de la tête de la liste, on retient ceux tels que la valeur `r` correspondante de 'représentativité cumulée' (au sens du § 3.4) respecte :

`r` > *nb-éléments-représentatifs-opérés*,

seulement jusqu'au premier (inclus) tel que la valeur `r` respecte :

`r` ≥ *nb-éléments-représentatifs-opérés* + rem_n.

Le premier des éléments ainsi retenus est finalement retiré s'il a déjà été opéré en partie, c'est-à-dire :
(cas 1 : ce premier élément est aussi l'élément de tête de l'*ensemble-représentatif*) si le *nb-éléments-représentatifs-opérés* vérifie :

$0 < $ *nb-éléments-représentatifs-opérés* $< r_0$,

ou bien (cas 2 : ce n'est pas l'élément de tête) si la valeur r_{i-1} correspondant à l'élément précédent vérifie :

nb-éléments-représentatifs-opérés $> r_{i-1}$. Puis,

. le *nb-éléments-représentatifs-non-opérés* et le *nb-éléments-représentatifs-opérés* sont respectivement diminué et augmenté de *rem_n*,
. enfin *rem_n* devient nul.

(ii) si *rem_n* est supérieur ou égal à *z* (on termine l'exemplaire en cours) :
. si et seulement si l'*effectif-opéré* est nul, on invoque la *procédure-transition-état* sur les éléments de la *liste-entités-non-opérées*, tels que la valeur `r` correspondante de 'représentativité cumulée' respecte :

`r` > *nb-éléments-représentatifs-opérés*,

sauf le premier d'entre eux s'il a déjà été opéré en partie (voir caractérisation ci-dessus).
. l'*effectif-non-opéré* est diminué de *1* et l'*effectif-opéré* est augmenté de *1*. Si l'*effectif-non-opéré* devient nul, *opéré-totalement* devient *vrai* pour l'OPERATED OBJECT en cours d'opération.
. puis *rem_n* est diminué de *nb-éléments-représentatifs-non-opérés*.
. enfin le *nb-éléments-représentatifs-non-opérés* redevient la somme des éléments de la *liste-de-représentativité* et le *nb-éléments-représentatifs-opérés* redevient nul.

Cas des QUANTITY-GRANULATED OPERATIONS :

Lors de chaque INVOCATION de *full-effect-on-quantity*, on est censé opérer une quantité d'une PROPRIETE égale à la *vitesse* `s` de l'OPERATION. Soit *rem_s* la quantité sur laquelle la *procédure-transition-état* n'a pas encore été invoquée, dans une INVOCATION de *full-effect-on-quantity*. La valeur *rem_s* est initialisée à `s` et doit décroître vers *0*.

Tant que *rem_s* est positif, on considère l'OPERATED OBJECT `current` renvoyé par *current-operated-object* appliquée à l'OPERATED OBJECT SPECIFICATION. Si et seulement si sa *quantité-opérée* est nulle, on applique la *procédure-transition-état* à son *entité-ressource*.

z ← la *quantité-non-opérée* de `current`,

(i) si *rem_s* est inférieur à *z* :
. la *quantité-opérée* est augmentée (et la *quantité-non-opérée* diminuée) de *rem_s*,
. *rem_s* devient nul.

(ii) si *rem_s* est supérieur ou égal à *z* :
. *rem_s* est diminué de la *quantité-non-opérée*,
. la *quantité-opérée* est valuée avec la valeur de la PROPRIETE, la *quantité-non-opérée* devient nulle,
. *opéré-totalement* devient *vrai*,
. l'*entité-ressource* est transférée dans la *liste-entités-opérées*.

Cas des TIME-GRANULATED OPERATIONS :

Lors de chaque INVOCATION de *step-effect-on-object*, on est censé réaliser un pourcentage de l'effet égal à la *vitesse* `q` de l'OPERATION. Soit *rem_q* le pourcentage de l'effet non encore réalisé, dans une INVOCATION de *step-effect-on-object*. La valeur *rem_q* est initialisée à `q` et doit décroître vers *0*.

Tant que *rem_q* est positif, on considère l'OPERATED OBJECT `current` renvoyé par *current-operated-object* appliquée à l'OPERATED OBJECT SPECIFICATION. Si et seulement si sa *proportion-opérée* est nulle, on applique la *procédure-transition-état* à son *entité-ressource*.

z ← la *proportion-non-opérée* de `current`,

(iii) si *rem_q* est inférieur à *z* :
. la *proportion-opérée* est augmentée (et la *proportion-non-opérée* diminuée) de *rem_q*,
. *rem_q* devient nul.

(iv) si *rem_q* est supérieur ou égal à *z* :
. *rem_q* est diminué de la *proportion-non-opérée*,
. la *proportion-opérée* est valuée avec *1*, la *proportion-non-opérée* devient nulle,
. *opéré-totalement* devient *vrai*,
. l'*entité-ressource* est transférée dans la *liste-entités-opérées*.

5.6.2.2 Exploitation des structures pour la scission d'objets opérés

On envisage dans ce paragraphe deux mécanismes de SCISSION aux finalités et contenus différents :

- la scission d'une ENTITE `E` qui est COMPOSANT ou ELEMENT du CONTROLLED SYSTEM à une profondeur quelconque, en deux nouvelles ENTITES E_1 et E_2. On remplace l'ENTITE `E` dans le CONTROLLED SYSTEM, et dans la STRATEGY si elle y est référencée, par une nouvelle ENTITE `E'` dont E_1 et E_2 sont les ELEMENTS. E_1 et E_2 conservent la structure (en termes de COMPOSANTS ou d'ELEMENTS) des sous-parties de `E` qu'elles représentent respectivement. L'ENTITE `E` n'est plus référençable. Une telle scission permet, dans la suite de la SIMULATION, la restitution plus exacte de l'effet des PROCESSUS qui auraient porté sur `E`, en captant les différences d'effet dans les PROPRIETES d'ENTITES distinctes E_1 et E_2. Ces différences d'effet se produisent dans deux cas :
  - une différence d'ETAT a déjà été créée entre E_1 et E_2, parce qu'elles ont réagi différemment à l'incidence d'un facteur externe ou interne au CONTROLLED SYSTEM, ou parce que la progressivité de l'effet d'une OPERATION a identifié des sous-parties opérées (E_1) et non opérées (E_2) ;
  - on sait qu'un facteur externe ou interne au CONTROLLED SYSTEM ne va plus agir de façon uniforme et homogène sur `E`, mais différemment sur E_1 et E_2.
- la scission de l'ensemble des objets d'une activité, au motif évident qu'on veut scinder l'activité `A` en deux activités A_1 et A_2 gérant distinctement deux sous-ensembles. Plus précisément, on scinde la liste `L` des objets de `A` en deux listes L_1 et L_2 qui constitueront les objets de A_1 et A_2. Cette scission peut nécessiter la scission (au sens de l'alinéa précédent) de telle ou telle ENTITE `E` qui est un élément de `L`.
  La scission d'une activité `A` peut être justifiée de trois façons :
  - on souhaite adapter la conduite à une différence d'ETAT, constatée ou prévue, entre deux sous-parties L_1 et L_2, différence liée à la progressivité d'une OPERATION où à l'incidence d'un facteur externe ou interne au CONTROLLED SYSTEM,

- on souhaite créer une différence d'ETAT entre deux sous-parties L_1 et L_2 en leur faisant subir les effets d'OPERATIONS distinctes,
- il y a intérêt, d'un point de vue organisationnel et indépendamment de l'ETAT, par exemple pour des raisons liées à l'utilisation des RESOURCES, à distinguer les activités portant sur deux sous-parties L_1 et L_2.

Ces deux mécanismes sont déclenchés de la façon suivante :

- la SCISSION d'une activité (c'est-à-dire de son objet) ne peut être commandée que par le MANAGER. Celui-ci peut la concevoir comme une modification de sa STRATEGY conditionnée à l'ETAT courant, et représentée par un CONDITIONAL ADJUSTMENT. Dans ce cas, la SCISSION sera invoquée, si le PREDICAT *state-predicate* est vérifié, dans la METHODE *strategy-adaptation*. La SCISSION peut aussi être la conséquence d'un mécanisme d'ALARME installé sur le CONTROLLED SYSTEM et sollicitant le MANAGER.
  L'intervention du MANAGER est requise même si la SCISSION d'activité est justifiée par une différence d'ETAT déjà observable. Cette différence pourra n'être effectivement observée par le MANAGER que lors du prochain EVENEMENT CHECK REACTIVE TRAJECTORY EVENT.
- la SCISSION d'une ENTITE (c'est-à-dire son remplacement) peut être provoquée de deux manières :
  - en l'invoquant explicitement dans une PROCEDURE (*procédure-exécution*, *procédure-initialisation*, *procédure-poursuite*) qui code l'effet d'un PROCESSUS sur le CONTROLLED SYSTEM.
    En particulier, cette PROCEDURE peut être *procédure-poursuite* de PROCEED OPERATION pour capter, dès qu'elle devient notable, une différence d'ETAT sur les parties opérées et non opérées d'une *entité-ressource* objet d'une activité.
  - en la considérant comme une conséquence d'un EVENEMENT dans l'ENVIRONNEMENT du PRODUCTION SYSTEM : la SCISSION est alors invoquée dans la *procédure-exécution* du PROCESSUS PONCTUEL auquel L'EVENEMENT adresse une directive.
  - en la plaçant en conséquence d'une INVOCATION REFLEXE : la SCISSION est alors invoquée dans l'*action-reflexe* de la SPECIFICATION D'INVOCATION REFLEXE correspondante.

Dans ce qui suit, on présente ces deux mécanismes de façon générale, puis particularisés au cas où la SCISSION est basée sur la distinction entre parties opérées et non opérées de l'objet d'une activité.

5.6.2.2.1 Scission de la spécification de l'objet d'une activité primitive

Le mécanisme de SCISSION de l'OPERATED OBJECT SPECIFICATION d'une PRIMITIVE ACTIVITY SPECIFICATION est réalisée par la METHODE *split-operated-object-spec* du MANAGER, argumentée par l'INSTANCE d'INSTRUCTION. De façon générale, il vise à établir deux listes d'OPERATED OBJECTS dont les éléments sont issus de la *liste-opérée* de l'OPERATED OBJECT SPECIFICATION de la PRIMITIVE ACTIVITY SPECIFICATION. En particulier ce couple de listes peut devenir argument de la METHODE *split-primitive-activity-spec*, qui scinde une activité primitive en deux activités distinctes.
On remarque que, puisqu'on part d'une *liste-opérée*, on ne peut mettre en œuvre cette SCISSION que si les OPERATED OBJECTS sont identifiés, soit parce que le DEVELOPPEMENT de l'OPERATED OBJECT SPECIFICATION a été réalisée, soit parce qu'ils ont été déterminés et construits avec la PRIMITIVE ACTIVITY SPECIFICATION.
De même que pour la METHODE *pre-split* (*cf.* § 3.3 et 3.4), le contenu de la METHODE *split-operated-object-spec* ainsi que la manière d'exploiter la valeur qu'elle renvoie, sont adaptés au contexte et au motif de son INVOCATION. De façon générale, elle a le contenu suivant :

```
mécanisme de SCISSION de l'objet d'une INSTRUCTION i par le MANAGER m :
   m . split-operated-object-spec(i) {
      liste d'OPERATED OBJECTS l1 = () ;
      liste d'OPERATED OBJECTS l2 = () ;
      liste de listes d'OPERATED OBJECTS résultat = {l1, l2} ;
      pour tout OPERATED OBJECT o
                        de liste-opérée de l'OPERATED OBJECT SPECIFICATION de i faire {
        // selon les PROPRIETES de o, l'empiler dans l1 ou l2
      }
      retourner résultat ;
   }
```

Lorsque la SCISSION de l'objet d'une INSTRUCTION est justifiée par une différence créée entre les parties opérées et non opérées, elle a typiquement le contenu suivant :

```
mécanisme de SCISSION de l'objet d'une INSTRUCTION i par le MANAGER m :
// sur la base des parties opérées et non opérées par l'OPERATION en jeu dans i
   m . split-operated-object-spec(i) {
      liste d'OPERATED OBJECTS l1 = () ;  // avec entités-ressources opérées
      liste d'OPERATED OBJECTS l2 = () ; // avec entités-ressources non opérées
      liste de listes d'OPERATED OBJECTS résultat = {l1, l2} ;
      pour tout OPERATED OBJECT o dans i faire {
         si o . liste-entités-non-opérées est vide      alors ajouter o à l1 ;
         sinon  {
            si o . liste-entités-opérées est vide        alors ajouter o à l2 ;
            sinon  {                                  // o n'est que partiellement opéré
               ENTITE e = o . entité-ressource ;
               // cas où e est une LOCATION :
                  PARTITION S = split(e, PARTITION, i) ;              // SCISSION de e
   new_o ← l'OPERATED OBJECT de i tel que S est son entité-ressource ;
 ajouter le premier ELEMENT de new_o à l1 et le second à l2;
               // cas où e est une POPULATION :
                  POPULATION MULTIPLE S = split(e, i) ;               // SCISSION de e
                  ... // puis idem / LOCATION
            }
         }
      }
      retourner résultat ;
   }
```

5.6.2.2.2 Scission d'une entité du système piloté

Seules les LOCATIONS et les POPULATIONS peuvent être l'objet du mécanisme de SCISSION d'ENTITE.
On rappelle (*cf.* § 3.3 et 3.4) que la SCISSION d'une LOCATION (resp. POPULATION) consiste à invoquer sa METHODE *pre-split* puis à la remplacer, dans le CONTROLLED SYSTEM, par la LOCATION (resp. POPULATION MULTIPLE) renvoyée.
On peut désormais ajouter que partout dans tout NOMINAL PLAN où l'*entité-ressource* est l'objet d'une activité, on remplace l'OPERATED OBJECT correspondant par un autre possédant en ELEMENTS (*i.e.* dans la *liste-sub-operated-objects*) deux OPERATED OBJECTS correspondant aux ENTITES résultant de la SCISSION de l'*entité-ressource*.
C'est la FONCTION *split*[57], argumentée par l'ENTITE à scinder, qui réalise ces tâches.

```
mécanisme de SCISSION d'une LOCATION l :
   LOCATION    split(l,  symb) { // symb : symbole d'une particularisation de LOCATION
      L = l . pre-split(symb) ;   // L : INSTANCE de la CLASSE symb
      pour toute PRIMITIVE ACTIVITY SPECIFICATION dans laquelle l est entité-ressource faire {
         o ← l'OPERATED OBJECT tel que l est son entité-ressource ;
         créer un OPERATED OBJECT new_o  avec L comme entité-ressource ;
         remplacer o par new_o ;
      }
      dans le CONTROLLED SYSTEM, remplacer l par L ;
      renvoyer L ;
   }
```

[57] Cette FONCTION n'est pas une METHODE d'ENTITE parcequ'elle a pour effet de détruire l'ENTITE en jeu. Sans qu'il soit nécessaire de le préciser ici, *split* peut être une METHODE de la CLASSE SIMULATION.

```
mécanisme de SCISSION d'une POPULATION p :
   POPULATION MULTIPLE    split(p) {
      POPULATION MULTIPLE P = p . pre-split() ;
      pour toute PRIMITIVE ACTIVITY SPECIFICATION dans laquelle p est entité-ressource faire {
         o ← l'OPERATED OBJECT tel que p est son entité-ressource ;
         créer un OPERATED OBJECT new_o avec P comme entité-ressource ;
         remplacer o par new_o ;
      }
      dans le CONTROLLED SYSTEM, remplacer p par P ;
      renvoyer P ;
   }
```

Bien que ce mécanisme puisse être mis en œuvre dans d'autres cas, on s'intéresse ici à la situation où la SCISSION est justifiée par une différence d'ETATS notable entre la partie opérée et la partie non opérée de l'*entité-ressource*, ce qui entraîne que *pre-split* est argumentée avec la PRIMITIVE ACTIVITY SPECIFICATION en jeu (*cf.* § suivant).

```
mécanisme de SCISSION d'une entité-ressource LOCATION e dans une  INSTRUCTION i :
// sur la base des parties opérées et non opérées par l'OPERATION en jeu dans i
   PARTITION L = split(e, PARTITION, this) ;
   renvoyer L ;
```

```
mécanisme de SCISSION d'une entité-ressource POPULATION e dans une  INSTRUCTION i :
// sur la base des parties opérées et non opérées par l'OPERATION en jeu dans i
   POPULATION MULTIPLE P = split(e, this) ;
   renvoyer P ;
```

On précise dans les deux paragraphes suivants le contenu spécifique qu'on doit donner à *pre-split* selon que l'ENTITE-RESSOURCE est une LOCATION ou une POPULATION, avant de préciser comment une SCISSION peut être déclenchée en cours de réalisation de l'effet d'une OPERATION.

5.6.2.2.2.1 Scission d'un espace

Appliquée à une INSTANCE de LOCATION `l`, *pre-split* prend en argument le symbole d'une *particularisation* de LOCATION. Elle renvoie une INSTANCE de la *particularisation* en lui donnant deux ELEMENTS `l'` et `l''`, INSTANCES de la même CLASSE que `l` (*cf.* § 3.3). Si *pre-split* est munie d'un second argument qui est la référence à une INSTRUCTION `i`, `l'` et `l''` sont construits sur la base des parties opérées et non opérées par l'OPERATION en jeu dans l'INSTRUCTION.

Si l'OPERATION en jeu dans `i` est une QUANTITY-GRANULATED OPERATION dont la PROPRIETE en jeu est, typiquement, la *surface*, l'INVOCATION `l` . *pre-split*(`partition`, `i`) renvoie une PARTITION dont les ELEMENTS sont `l'` et `l''`, telle que :

- `l'` est dans l'ETAT de l ; sa *surface* est la *quantité-opérée* de l'OPERATED OBJECT de `i` dont l'*entité-ressource* est `l` ;
- `l''` est dans l'ETAT de l'*entité-ressource-mémoire* du même OPERATED OBJECT ; sa *surface* est la *quantité-non-opérée* du même OPERATED OBJECT.

Si l'OPERATION en jeu dans `i` est une UNIT-GRANULATED OPERATION, l'INVOCATION `l` . *pre-split*(`partition`, `i`) renvoie une PARTITION dont les ELEMENTS sont `l'` et `l''`, telle que :

- les ELEMENTS de `l'` sont ceux de la *liste-entités-opérées* de l'OPERATED OBJECT de `i` dont l'*entité-ressource* est `l` ;
- les ELEMENTS de `l''` sont ceux de la *liste-entités-non-opérées* du même OPERATED OBJECT.

Dans tous les cas, *pre-split* peut invoquer la METHODE *pre-dispatch-resident-entities* (*cf.* § 3.3) et exploiter son résultat pour répartir dans `l'` et `l''` les *entités-résidentes* de `l`.

5.6.2.2.2.2 Scission de populations

Appliquée à une INSTANCE de POPULATION `p`, *pre-split* renvoie une POPULATION MULTIPLE en lui donnant deux ELEMENTS `p'` et `p''` INSTANCES de la même CLASSE que `p` (*cf.* § 3.4). Si *pre-split* est munie d'un second argument qui est la

référence à une INSTRUCTION `i`, `p'` et `p''` sont construits sur la base des parties opérées et non opérées par l'OPERATION en jeu dans l'INSTRUCTION. Dans ce cas, la SCISSION de POPULATION, et par conséquent l'INVOCATION de `p` . *pre-split*(`i`), n'ont de sens que si l'OPERATION en jeu dans `i` est une UNIT-GRANULATED OPERATION.

Si `p` est une POPULATION SIMPLE HOMOGENE, `p` . *pre-split*(`i`) renvoie, placés en ELEMENTS d'une POPULATION MULTIPLE, deux POPULATIONS SIMPLES HOMOGENES `p'` et `p''` dont les *ensembles-représentatifs* sont celui de p et telles que :

- l'*effectif* de `p'` est l'*effectif-opéré* de l'OPERATED OBJECT de `i` dont l'*entité-ressource* est `p` ;
- l'*effectif* de `p''` est l'*effectif-non-opéré* du même OPERATED OBJECT.

Si `p` est une POPULATION SIMPLE HETEROGENE, `p` . *pre-split*(`i`) renvoie, placés en ELEMENTS d'une POPULATION MULTIPLE, deux POPULATIONS SIMPLES HETEROGENES `p'` et `p''` dont les *ensembles-représentatifs* sont celui de p et telles que :

- l'*effectif* de `p'` est l'*effectif-opéré* de l'OPERATED OBJECT de `i` dont l'*entité-ressource* est `p` ;
- l'*effectif* de `p''` est l'*effectif-non-opéré* du même OPERATED OBJECT[58].

Si `p` est une POPULATION MULTIPLE, `p` . *pre-split*(`i`) renvoie, placés en ELEMENTS d'une POPULATION MULTIPLE, deux POPULATIONS MULTIPLES `p'` et `p''` telles que :

- `p'` contient les POPULATIONS SIMPLES opérées totalement, en conservant les éventuelles structurations en POPULATIONS MULTIPLES qu'elles avaient dans `p` ;
- `p''` contient les POPULATIONS SIMPLES non opérées totalement, en conservant aussi les éventuelles structurations.

Pratiquement, on réalise deux copies de `p`, `p'` et `p''`. Leurs éléments terminaux (feuilles) sont les POPULATIONS SIMPLES. Puis on supprime dans `p'` les feuilles non opérées totalement et dans `p''` les feuilles opérées totalement. Enfin, toute POPULATION MULTIPLE, hormis `p'` et `p''`, qui n'a qu'un élément est remplacé par cet élément[59].

5.6.2.2.2.3 Déclenchement de la scission d'une entité en cours d'opération

On précise ici comment se déclenche la SCISSION d'une ENTITE, alors qu'une OPERATION est en cours de mise en œuvre sur cette ENTITE. On est dans le cas où la SCISSION est justifiée par une différence d'ETAT notable entre la partie opérée et la partie non opérée (*cf.* entête du § 5.6.2.2). On rappelle d'une part que, par spécification de la FONCTION *split*, on ne peut scinder qu'une *entité-ressource* (et non une ENTITE qui serait COMPOSANT ou ELEMENT à une profondeur quelconque d'une *entité-ressource*), et d'autre part que la SCISSION n'est possible que si le *mode-de-progression* de l'OPERATION est `unit` ou `quantity`.

On exploite une METHODE additionnelle des UNIT-GRANULATED OPERATIONS et des QUANTITY-GRANULATED OPERATIONS, *split-condition*. Sa VALEUR est la référence à un PREDICAT qui renvoie *vrai* s'il est justifié de scinder l'*entité-ressource* de l'OPERATED OBJECT courant de l'INSTRUCTION dans laquelle l'OPERATION est en jeu. L'OPERATED OBJECT courant est celui renvoyé par une INVOCATION de *current-operated-object* sur l'OPERATED OBJECT SPECIFICATION de l'INSTRUCTION. On le passe en argument de *split-condition*. Il suffit alors de tester la *split-condition* à chaque *pas* de PROCESSUS PROCEED OPERATION[60], et d'invoquer la FONCTION *split* sur l'*entité-ressource* si le PREDICAT renvoie *vrai*.

---

58 Cette procédure n'est correcte que s'il n'y a pas d'exemplaire de l'*ensemble-représentatif* qui ne soit que partiellement opéré au moment de la SCISSION. Dans le cas contraire, c'est le premier exemplaire de `p''` qui est partiellement opéré. Alors, le *nombre-éléments-représentatifs-opérés* de l'OPERATED OBJECT correspondant devrait être ajusté en conséquence dans la METHODE *split-primitive-activity-spec* du MANAGER, à condition que *split-operated-object* lui passe l'information, ce qui n'est pas le cas dans la rédaction en entête du § 5.6.2.2.

59 Soit par exemple : `p` = PM1 {PM2 { PM3 {ps1, ps2},
PM4 {ps3, ps4}},
PM5 {ps5, ps6}},
et soit ps1, ps2 et ps3 les seules POPULATIONS SIMPLES dont *opéré-totalement* est *vrai*. Alors :
`p'` = PM2' { PM3' {ps1, ps2},
ps3
et `p''` = PM1'' { ps4,
PM5'' {ps5, ps6}}

60 Cette procédure ne permet donc pas de scinder une *entité-ressource* pendant une période de suspension de l'activité, sur la justification que la différence d'ETAT au moment de la suspension durerait depuis suffisamment longtemps. Cette fonctionnalité peut être obtenue en programmant dans ACTING INSTRUCTION LIST, pour toute activité suspendue, un EVENEMENT PROCEED OPERATION EVENT particulier, tel que la *procédure-poursuite* du PROCESSUS PROCEED OPERATION ne ferait que tester la *split-condition* et réaliser la SCISSION le cas échéant, sans faire progresser l'effet de l'OPERATION.

```
→ {   à chaque pas de PROCEED-OPERATION cp
      faire cp . procédure-poursuite() :
          // pour exemple : si l'OPERATION est une QUANTITY-GRANULATED OPERATION :
          full-effect-on-quantity(cp) {
              OPERATION ot = cp . processed-object() ;
              ...       // réalisation de l'effet (cf. § 5.6.1)
              PRIMITIVE ACTIVITY SPECIFICATION i = ot . get-instruction() ;
              OPERATED OBJECT SPECIFICATION oo-spec = i . get-component(0) ;
              OPERATED OBJECT oo = oo-spec . current-operated-object() ;
              si ot . split-condition(oo) = vrai alors {
                  ENTITE e = oo . entité-ressource ;
                  // cas où e est une LOCATION
                  LOCATION L = split(e, partition) ;
                  // cas où e est une POPULATION
                  POPULATION MULTIPLE P = split(e) ;
              }
          }
}
```

5.7 Les mécanismes sur les ressources

5.7.1 Immobilisation / mobilisation

Ces deux mécanismes peuvent être décrits de manières identiques, en remplaçant la notion d'immobilisation par celle de mobilisation et *vice-versa*. On ne donne ici qu'une des deux expressions.

Le mécanisme d'IMMOBILISATION d'une RESOURCE est opéré dans deux situations :

- lorsque le modélisateur a explicitement programmé dans l'AGENDA du SYSTEME un EVENEMENT RESOURCE IMMOBILIZATION EVENT, en fixant lui-même sa *date* ou sa *fenêtre d'occurrence*, son *degré-de-priorité* et sa *probalilité*. En même temps, le modélisateur affecte une ENTITE objet (nécessairement une INSTANCE de RESOURCE) au PROCESSUS standard RESOURCE IMMOBILIZATION PROCESS (attaché automatiquement à l'EVENEMENT, et mls en œuvre par la METHODE *procédure-exécution*).
- dans le mécanisme de MOBILISATION, lorsque le modélisateur n'a pas inhibé l'AUTOGENERATION. En effet, celle-ci programme automatiquement un EVENEMENT RESOURCE IMMOBILIZATION EVENT, en fixant l'OCCURRENCE avec les informations attachées à cet effet à l'EVENEMENT d'origine. L'AUTOGENERATION peut être inhibée en assignant un pointeur nul à la *procédure-autogénération*, surchargeant ainsi la procédure assignée par défaut (*cf.* § 3.1).

Dans les deux cas, la *procédure-exécution* change la VALEUR de l'*état-de-disponibiité* de la RESOURCE objet, de `disponible` à `non_disponible`.

Le mécanisme d'IMMOBILISATION est donc doté, on vient de le voir, d'une AUTOGENERATION. Noter que la RESOURCE objet du PROCESSUS de l'EVENEMENT d'origine devient automatiquement la RESOURCE objet du PROCESSUS de l'EVENEMENT généré. On peut ainsi programmer une série d'immobilisation/mobilisation portant sur cette INSTANCE de RESOURCE.

Il est en outre doté d'une POSTCONSEQUENCE. Celle-ci est la PROGRAMMATION en l'INSTANT courant et avec une *probabilité* de *1*, d'un EVENEMENT MAKE INSTRUCTION LIST EVENT.

5.8 Ordonnancement des événements

On s'intéresse ici à deux ensembles d'événements :

- les EVENEMENTS prédéfinis dans l'ontologie des systèmes pilotés (qui gouvernent des PROCESSUS eux-mêmes prédéfinis) : UPDATE SITUATION EVENT, MAKING INSTRUCTION LIST EVENT, ACTING INSTRUCTION LIST EVENT, PROCEED OPERATION EVENT, CHECK REACTIVE TRAJECTORY EVENT, RESOURCE [IM]MOBILIZATION EVENT,
- les EVENEMENTS propres à une application particulière et par conséquent non définis dans la-dite ontologie.

On rappelle que la SIMULATION consiste à placer de tels EVENEMENTS dans l'AGENDA du SYSTEME, puis à invoquer la procédure générale *run-simulation* décrite dans la section 5.1. Les EVENEMENTS sont rangés dans l'AGENDA par ordre

d'antériorité des INSTANTS d'OCCURRENCE (le premier EVENEMENT est toujours celui antérieur à tous les autres). Lorsque deux EVENEMENTS sont programmés au même INSTANT, on sait qu'ils peuvent être rangés en utilisant le critère du *degré-de-priorité*. Le MECANISME de SIMULATION décrit dans la présente ontologie suppose l'utilisation d'une METHODE *before-in-agenda-events* (celle écrite par le modélisateur de l'application particulière, ou bien celle prédéfinie dans l'implémentation des concepts de base et utilisée par défaut de la précédente) qui utilise effectivement ce critère, après celui de l'antériorité, mais avant tout autre qui serait propre à une application particulière.

Dans ce cadre, les *degrés-de-priorité* suivants sont donnés automatiquement aux EVENEMENTS prédéfinis :

RESOURCE [IM]MOBILIZATION EVENT :	35
UPDATE SITUATION EVENT :	40
MAKING INSTRUCTION LIST EVENT :	45
ACTING INSTRUCTION LIST EVENT :	50
CHECK REACTIVE TRAJECTORY EVENT :	55
PROCEED OPERATION EVENT :	59 (d'abord les opérations permanentes …)
PROCEED OPERATION EVENT :	60 (…puis les autres)

Le modélisateur d'une application particulière est libre d'affecter aux EVENEMENTS propres à cette application des *degrés-de-priorité* inférieurs ou supérieurs aux valeurs ci-dessus (dans la limite du domaine [0 99]).

Les EVENEMENTS portant sur les OPERATIONS ont un *degré-de-priorité* par défaut (voir le tableau ci-dessus). Cependant, si une *priorité-d-exécution* est spécifiée pour une OPERATION particulière, et que ce n'est pas la valeur par défaut de l'ATTRIBUT, alors cette *priorité-d-exécution* devient le *degré-de-priorité* des EVENEMENTS d'initialisation et de poursuite de l'exécution de l'OPERATION.

Index

classes d'entités

Annexe 1 : check-sons-if {-waiting() | -open() | -closed() }
check-sons-if {-waiting(*a*) | -open(*a*) | -closed(*a*) }

Préconditions aux changements de situation d'une activité A, portant sur la situation des composantes a_i, selon la classe de A
Si *a* est passé en argument, les préconditions complémentaires ne sont pas testées si elles portent sur *a*.
Les composantes de A sont : a_0, a_1, ... a_i, ... a_{n-1}

	Lorsque A . validate-{waiting,open, closed} est invoqué, la nouvelle valeur candidate pour la situation de A *est selon le cas :*		
	waiting	**open**	**closed**
	Alors il faut que soit respectée la précondition générale suivante :		
précondition générale →	A sleeping	a_i waiting ou a_i . validate-waiting(from_up | from_down, ...)	A open
classe de A ↓	*De plus, et selon la classe de A, les préconditions complémentaires au changement sont les suivantes :*		
primitive	sans objet	sans objet	sans objet
before **unordered before**	a_0 . validate-waiting(from_up, A)	a_0 . opening-predicate() = 1 et a_0 . validate-open(from_up, A)	a_{n-1} . closing-predicate() = 1 et a_{n-1} . validate-closed(from_up, A) ou pas de composantes et A. opening-predicate() != false
meeting	a_0 . validate-waiting(from_up, A)	a_0 . opening-predicate() = 1 et a_0 . validate-open(from_up, A)	a_{n-1} . closing-predicate() = 1 et a_{n-1} . validate-closed(from_up, A) ou pas de composantes et A. opening-predicate() != false
overlapping	a_0 . validate-waiting(from_up, A)	a_0 . opening-predicate() = 1 et a_0 . validate-open(from_up, A)	a_{n-1} . closing-predicate() = 1 et a_{n-1} . validate-closed(from_up, A) ou pas de composantes et A. opening-predicate() != false
inclusion	a_0 . validate-waiting(from_up, A)	a_0 . opening-predicate() = 1 et a_0 . validate-open(from_up, A)	a_0 . closing-predicate() = 1 et a_0 . validate-closed(from_up, A) ou pas de composantes et A. opening-predicate() != false
costarting	$\forall k, a_k$. validate-waiting(from_up, A)	$\forall k, a_k$. opening-predicate() = 1 et a_k . validate-open(from_up, A)	$\forall k, a_k$ closed ou a_k . closing-predicate() = 1 et a_k . validate-closed(from_up, A)
coending	$\exists k$ tq a_k . validate-waiting(from_up, A)	$\exists k$ tq a_k . opening-predicate() = 1 et a_k . validate-open(from_up, A)	$\forall k, a_k$. closing-predicate() = 1 et a_k . validate-closed(from_up, A) toutes dates de fermeture égales
equality	$\forall k, a_k$. validate-waiting(from_up, A)	$\forall k, a_k$. opening-predicate() = 1 et a_k . validate-open(from_up, A)	$\forall k, a_k$. closing-predicate() = 1 et a_k . validate-closed(from_up, A) toutes dates de fermeture égales
disjunction	$\exists k$ tq a_k . validate-waiting(from_up, A)	$\exists k$ tq a_k . opening-predicate() = 1 et a_k . validate-open(from_up, A)	$\forall k, a_k$ cancelled ou a_k closed ou a_k . closing-predicate() = 1 et a_k . validate-closed(from_up, A)
conjunction	$\exists k$ tq a_k . validate-waiting(from_up, A)	$\exists k$ tq a_k . opening-predicate() = 1 et a_k . validate-open(from_up, A)	$\forall k, a_k$ closed ou a_k . closing-predicate() = 1 et a_k . validate-closed(from_up, A)
iteration	/	/	/
optional	a_0 . validate-waiting(from_up, A)	a_0 . opening-predicate() = 1 et a_0 . validate-open(from_up, A)	a_0 . closing-predicate() = 1 et a_0 . validate-closed(from_up, A)
constrained optional	a_0 . validate-waiting(from_up, A)	a_0 . opening-predicate() = 1 et a_0 . validate-open(from_up, A)	a_0 . closing-predicate() = 1 et a_0 . validate-closed(from_up, A)

A . check-sons-if-waiting() (resp. A . check-sons-if-open(), A . check-sons-if-closed()) renvoie vrai ssi les préconditions complémentaires au passage de A à `waiting` (resp. `open`, `closed`) sont vérifiées.

a_i . opening-predicate() renvoie : 1 si les conditions d'ouverture (resp. de fermeture) de a_i sont vérifiées,
(resp. a_i . closing-predicate()) 0 sinon

Les conditions d'ouverture de a_i sont vérifiées ssi :

- la date courante est comprise entre *début-au-plus-tôt* et *début-au-plus-tard*,
- le PREDICAT *condition-ouverture* renvoie *vrai*, ou bien n'est pas spécifié.

Les *début-au-plus-tôt* et *début-au-plus-tard* sont automatiquement ajustés en fonction de *délais-o-o* (resp. *délais-f-o*), par rapport à l'ouverture (resp. la fermeture) de la composante précédant a_i.

x . validate-open(from, y) renvoie vrai (*idem* pour x . validate-waiting(from, y), x . validate-closed(from, y)) si x est `open` ou si :

la précondition générale au passage de x à `open` est vérifiée et si
x . check-sons-if-open() si from = from_up ou x . check-sons-if-open(y) si from = from_down renvoie vrai et si
m_{ik} . check-if-son-open(x) renvoie vrai pour toutes les activités m_{ik} dont x est composante (excepté y si from = from_up)

Annexe 2 : check-if-son {-waiting(a_i) | -open(a_i) | -closed(a_i) }

Préconditions aux changements de situation d'une activité a_i, selon la classe de l'activité A dont elle est composante.
Les composantes de A sont : a_0, a_1, ... a_i, ... a_{n-1}

	Lorsque a_i . validate-{-waiting,-open, -closed} est invoqué, ou dans UpdateStatus() quand a_i .-openningPredicate() = 1 ou a_i .ClosingPredicate() = 1, la nouvelle valeur candidate pour la situation de a_i est selon le cas :		
	waiting	**open**	**closed**
	Alors il faut que soit respectée la précondition générale suivante :		
précondition générale →	a_i sleeping	a_i waiting ou a_i . validate-waiting(from_up | from_down, A)	a_i open
classe de A ↓	*De plus, et selon la classe de A, les préconditions complémentaires au changement sont les suivantes :*		
primitive	sans objet	sans objet	sans objet
before **unordered before**	si i = 0 : A.validate-waiting(from_down, a_i) si i > 0 : a_{i-1} closed ou { a_{i-1} . ClosingPredicate() = 1 et a_{i-1} . validate-closed(from_up, A)}	si i = 0 : A.validate-open(from_down, a_i) si i > 0 : current-date > a_{i-1} . end-date()	si i = n-1 : A.validate-closed(from_down, a_i)
meeting	si i = 0 : A.validate-waiting(from_down, a_i) si i > 0 : a_{i-1} . ClosingPredicate() = 1 et a_{i-1} . validate-closed(from_up, A)	si i = 0 : A.validate-open(from_down, a_i) si i > 0 : a_{i-1} . ClosingPredicate() = 1 et a_{i-1} . validate-closed(from_up, A)	si i = n-1 : A.validate-closed(from_down, a_i) si i < n-1 : a_{i+1} . -openningPredicate() = 1 et a_{i+1} . validate-open(from_up, A)
overlapping	si i = 0 : A.validate-waiting(from_down, a_i) si i > 0 : a_{i-1} open ou { a_{i-1} . -openningPredicate() = 1 et a_{i-1} . validate-open(from_up, A) }	si i = 0 : A.validate-open(from_down, a_i) si i > 0 : current-date > a_{i-1} . beg-date()	si i = n-1 : A.validate-closed(from_down, a_i) si i < n-1 : a_{i+1} open si i > 0 : current-date > a_{i-1} . end-date()
inclusion	si i = 0 : A.validate-waiting(from_down, a_i) si i > 0 : a_{i-1} open ou { a_{i-1} . -openningPredicate() = 1 et a_{i-1} . validate-open(from_up, A) }	si i = 0 : A.validate-open(from_down, a_i) si i > 0 : current-date > a_{i-1} . beg-date()	si i =0 : A.validate-closed(from_down, a_i) si i > 0 : a_{i-1} open si i < n-1 : current-date > a_{i+1} . end-date()
costarting	A.validate-waiting(from_down, a_i)	A.validate-open(from_down, a_i)	/
coending	A open ou A.validate-waiting(from_down, a_i)	A open ou A.validate-open(from_down, a_i)	si $\forall k \neq i$, a_k . closed alors A.validate-closed(from_down, a_i) sinon faux
equality	A.validate-waiting(from_down, a_i)	A.validate-open(from_down, a_i)	si $\forall k \neq i$, a_k . closed alors A.validate-closed(from_down, a_i) sinon faux
disjunction	A.validate-waiting(from_down, a_i)	A.validate-open(from_down, a_i)	A.validate-closed(from_down, a_i)
conjunction	A.validate-waiting(from_down, a_i)	A.validate-open(from_down, a_i)	vrai
iteration	A open ou A waiting	A open	A open
optional	A.validate-waiting(from_down, a_i)	A.validate-open(from_down, a_i)	A.validate-closed(from_down, a_i)
constrained optional	A.validate-waiting(from_down, a_i)	A.validate-open(from_down, a_i)	A.validate-closed(from_down, a_i)

A . check-if-son-waiting(a_i) (resp. A . check-if-son-open(a_i), A . check-if-son-closed(a_i)) renvoie vrai ssi
les préconditions complémentaires au passage de a_i à `waiting` (resp. `open`, `closed`) sont vérifiées.

a_i . opening-predicate() renvoie : 1 si les conditions d'ouverture (resp. de fermeture) de a_i sont vérifiées,
(resp. a_i . closing-predicate()) 0 sinon

Les conditions d'ouverture de a_i sont vérifiées ssi :

- la date courante est comprise entre *début-au-plus-tôt* et *début-au-plus-tard*,
- le PREDICAT *condition-ouverture* renvoie *vrai*, ou bien n'est pas spécifié.

Les *début-au-plus-tôt* et *début-au-plus-tard* sont automatiquement ajustés en fonction de *délais-o-o* (resp. *délais-f-o*), par rapport à l'ouverture (resp. la fermeture) de la composante précédant a_i.

x . validate-open(from, y) renvoie vrai (*idem* pour x . validate-waiting(from, y), x . validate-closed(from, y)) si x est `open` ou si :

la précondition générale au passage de x à `open` est vérifiée et si
x . check-sons-if-open() si from = from_up ou x . check-sons-if-open(y) si from = from_down renvoie vrai et si
m_{ik} . check-if-son-open(x) renvoie vrai pour toutes les activités m_{ik} dont x est composante (excepté y si from = from_up)

Annexe 3 : propagate {-waiting | -open | -closed}-to-sons(a_i)

Propagation aux composantes de A d'un changement de situation de A
Les composantes de A sont : a_0, a_1, ... a_i, ... a_{n-1}

A . propagate{-waiting -open -closed}-to-sons(a_i) : si le changement de situation de A a été provoqué par un changement de situation de sa composante a_i, la propagation ne porte pas sur a_i.

	La nouvelle valeur pour la situation de A est devenue selon le cas : ...		
	waiting	**open**	**closed**
	... lorsque A est une des composantes de B et que B . propagate{-waiting,-open,-closed}-to-sons(b) a été invoqué, avec b ≠ A, ou bien lorsque A .UpdateIfSon{-waiting,-open, -closed}(a_i) a été invoqué, ou bien lorsque A .TurnTo{-open, -closed} a été invoqué dans UpdateStatus().		
classe de A ↓	*Alors, l'invocation de A . propagate{-waiting,-open, -closed}-to-sons(a_i) provoque :*		
primitive	sans objet	sans objet	sans objet
before	a_0 waiting	a_0 open	a_{n-1} closed
unordered before	a_0 waiting	a_0 open	a_0 et a_1 closed
meeting	a_0 waiting	a_0 open	a_{n-1} closed
overlapping	a_0 waiting	a_0 open a_1 waiting ssi a_1.validate-waiting(from_up, A)	a_{n-1} closed
inclusion	a_0 waiting	a_0 open a_1 waiting ssi a_1.validate-waiting(from_up, A)	a_0 closed
costarting	∀k, a_k waiting	∀k, a_k open	∀k, a_k closed ssi a_k open
coending	∀k, a_k waiting ssi a_k.validate-waiting(from_up, A)	∀k, a_k open ssi a_k.validate-open(from_up, A)	∀k, a_k closed
equality	∀k, a_k waiting	∀k, a_k open	∀k, a_k closed
disjunction	∀k, a_k waiting ssi a_k.validate-waiting(from_up, A)	∀k, a_k open ssi a_k.validate-open(from_up, A)	∀k, a_k closed ssi a_k open
conjunction	∀k, a_k waiting ssi a_k.validate-waiting(from_up, A)	∀k, a_k open ssi a_k.validate-open(from_up, A)	∀k, a_k closed ssi a_k open
iteration	/	a_0 waiting ssi a_0 . validate-waiting(from_up, A)	a_0 closed ssi a_0 sleeping ou waiting
optional	a_0 waiting	a_0 open	a_0 closed
constrained optional	a_0 waiting	a_0 open	a_0 closed

Annexe 4 : update-if-son{-waiting | -open | -closed}(a_i)

Propagation à A et aux composantes de A d'un changement de situation sur une composante a_i de A
Les composantes de A sont : a_0, a_1, ... a_i, ... a_{n-1}

A . update-if-son{-waiting -open -closed}(a_i) : a_i est la composante dont la situation a changé.

	La nouvelle valeur pour la situation de a_i est devenue selon le cas : ...		
	waiting	**open**	**closed**
	... lorsque a_i est une des composantes de B et que B . Propagate{-waiting,-open, -closed}ToSons(b) a été invoqué, avec b ≠ a_i, ou bien lorsque a_i .update-if-son{-waiting, -open, -closed}(b) a été invoqué, ou bien lorque a_i .TurnTo{-open, -closed} a été invoqué dans UpdateStatus().		
classe de A ↓	alors, l'invocation de A . update-if-son{-waiting, -open, -closed}(a_i) provoque :		
primitive	sans objet	sans objet	sans objet
before **unordered before**	si i = 0 : A waiting si i > 0 : a_{i-1} closed ssi a_{i-1} open	si i = 0 : A open	si i = n-1 : A closed ssi A.validate-closed(from_down, a_i) si i < n-1 : a_{i+1} waiting ssi a_{i+1} . validate-waiting(from_up, A)
meeting	si i = 0 : A waiting si i > 0 : a_{i-1} closed	si i = 0 : A open si i > 0 : a_{i-1} closed	si i = n-1 : A closed ssi A.validate-closed(from_down, a_i) si i < n-1 : a_{i+1} waiting puis open
overlapping	si i = 0 : A waiting si i > 0 : a_{i-1} open ssi a_{i-1} waiting	si i = 0 : A open si i < n-1 : a_{i+1} waiting ssi a_{i+1} . validate-waiting(from_up, A)	si i = n-1 : A closed
inclusion	si i = 0 : A waiting si i > 0 : a_{i-1} open ssi a_{i-1} waiting	si i = 0 : A open si i < n-1 : a_{i+1} waiting ssi a_{i+1} . validate-waiting(from_up, A)	si i = 0 : A closed
costarting	A waiting	A open	A closed, ssi ∀k, a_k closed ou a_k . validate-closed(from_up, A)
coending	A waiting	A open	A closed[61]
equality	A waiting	A open	A closed
disjunction	A waiting	A open[62]	A closed, ssi a_k closed ou cancelled ∀k
conjunction	A waiting	A open	A closed, ssi ∀k, a_k closed ou a_k . validate-closed(from_up, A)
iteration	/	/	a_0 waiting ssi A . ClosingPredicate() ≠ 1
optional	A waiting	A open	A closed
constrained optional	A waiting	A open	A closed

[61] On arrête les opérations dont le degré de progression a été mis à jour dans ce pas ; on prépare les autres à subir cette dernière mise à jour avant de les arrêter. Même remarque pour les EQUALITY ACTIVITY SPECIFICATIONS.

[62] Les autres composantes ne deviendront `cancelled` que dès que l'OPERATION en jeu dans a_i deviendra `executing`.

Annexe 5 : la détermination des ressources requises par une activité

Le résultat de la DETERMINATION d'une PRIMITIVE ACTIVITY SPECIFICATION est une liste Q, de la forme :

Q : $\{q_1, ..., q_{nq}\}$, ou q_k, k=1,n_q, est une alternative de réquisition compatible avec la spécification.

Chaque alternative fera l'objet d'une tentative d'allocation, qui réussira ou échouera en fonction de la disponibilité des RESOURCES, puis sera confrontée aux différents types de contraintes. Les alternatives (aucune, une seule ou plusieurs) qui auront passé ces deux filtres constitueront autant de manières de mettre en œuvre la même PRIMITIVE ACTIVITY SPECIFICATION, différenciées par les RESOURCES mobilisées. Ces versions, s'il en a bien plusieurs, seront en concurrence dans le mécanisme SELECTING INSTRUCTION LIST.

La liste Q est construite selon la PROCEDURE suivante.

1. DEVELOPPEMENT de la PERFORMER SPECIFICATION :

L'EXPANSION de la SPECIFICATION D'ENSEMBLE D'ENTITES, puis la FONCTION de sélection renvoient une liste $_pQ$:
$_pQ$: { $_pQ^1$, .. $_pQ^{pm}$}
avec $_pm$: nombre d'alternatives de réquisition de PERFORMERS
$_pQ^j$: (p^j_1, .. p^j_{pn})
avec $_pn$: nombre (commun) de PERFORMERS dans toutes les alternatives $_pQ^j$
et soit w_j la *puissance* globale des PERFORMERS de $_pQ^j$

2. DEVELOPPEMENT de l'OPERATION SPECIFICATION :

L'INSTANCIATION de l'OPERATION réalise le DEVELOPPEMENT de ses n_s OPERATION RESOURCE SPECIFICATIONS. Ce DEVELOPPEMENT prend en compte, en particulier, la *puissance* de l'ensemble des PERFORMERS requis par l'activité. Il convient donc de réaliser autant de DEVELOPPEMENTS des OPERATION RESOURCE SPECIFICATIONS que d'alternatives de réquisition de PERFORMERS établies à la phase 2, c'est-à-dire $_pm$.

2.1 DEVELOPPEMENT des OPERATION RESOURCE SPECIFICATIONS pour une alternative de réquisition de PERFORMERS

Pour l'alternative $_pQ^k$, on construit la liste ^k_rQ en faisant d'abord le produit cartésien des résultats $^k_r\xi_j$, i=1,n_s des DEVELOPPEMENTS des n_s OPERATION RESOURCE SPECIFICATIONS, puis en concaténant les listes dans chaque élément du produit.

Ce qui donne, pour la i-ème OPERATION RESOURCE SPECIFICATION :
$^k_r\xi_j$ = { ($^ir^1_1$, .. $^ir^1_{rni}$), ..., ($^ir^{mi,k}_1$, .. $^ir^{rmi,k}_{rni}$) }
avec $_rm_{i,k}$: nombre d'alternatives de réquisition des RESOURCES par la i-ème spécification, pour la k-ème alternative de réquisition de PERFORMERS
$_rn_j$: nombre (commun) d'INSTANCES de RESOURCES dans toutes les alternatives

Pour l'ensemble des OPERATION RESOURCE SPECIFICATIONS :
$^k_rQ = {}^k_r\xi_1 \otimes ... \otimes {}^k_r\xi_{ns}$, c'est-à-dire :
{ ($^1r^1_1$, .. $^1r^1_{rn1,k}$), ..., ($^1r^{rm1,k}_1$, .. $^1r^{rm1,k}_{rn1,k}$) } ⊗ ... ⊗ { ($^{ns}r^1_1$, .. $^{ns}r^1_{rnns,k}$), ..., ($^{ns}r^{rmns,k}_1$, .. $^{ns}r^{rmns,k}_{rnns,k}$) }
ce qui s'écrit :
{ (($^1r^1_1$, .. $^1r^1_{rn1,k}$) ... ($^{ns}r^1_1$, .. $^{ns}r^1_{rnns,k}$)), ..
(($^1r^1_1$, .. $^1r^1_{rn1,k}$) ... ($^{ns}r^{rmns,k}_1$, .. $^{ns}r^{rmns,k}_{rnns,k}$)), ..
(($^1r^{rm1,k}_1$, .. $^1r^{rm1,k}_{rn1,k}$) ... ($^{ns}r^1_1$, .. $^{ns}r^1_{rnns,k}$)), ..
(($^1r^{rm1,k}_1$, .. $^1r^{rm1,k}_{rn1,k}$) ... ($^{ns}r^{rmns,k}_1$, .. $^{ns}r^{rmns,k}_{rnns,k}$)) }

⊗ (par concaténation des listes dans chaque élément)
{ ($^1r^1_1$, .. $^1r^1_{rn1,k}$, .. $^{ns}r^1_1$, .. $^{ns}r^1_{rnns,k}$), ..
($^1r^1_1$, .. $^1r^1_{rn1,k}$, .. $^{ns}r^{rmns,k}_1$, .. $^{ns}r^{rmns,k}_{rnns,k}$), ..
($^1r^{rm1,k}_1$, .. $^1r^{rm1,k}_{rn1,k}$, .. $^{ns}r^1_1$, .. $^{ns}r^1_{rnns,k}$), ..
($^1r^{rm1,k}_1$, .. $^1r^{rm1,k}_{rn1,k}$, .. $^{ns}r^{rmns,k}_1$, .. $^{ns}r^{rmns,k}_{rnns,k}$) }
Ainsi,
^k_rQ : { $^k_rQ^1$, .. $^k_rQ^{rmk}$}
avec $_rm_k$: nombre d'alternatives de réquisition de l'ensemble des RESOURCES pour la k-ème alternative de réquisition de PERFORMERS
$_rm_k = {}_1m_{1,k}$ x... x $_1m_{ns,k}$

2.2 Prise en compte de toutes les alternatives de réquisition de PERFORMERS

Après les DEVELOPPEMENTS conjoints de la PERFORMER SPECIFICATION et de l'OPERATION SPECIFICATION, on dispose d'une liste :
$_{pr}Q$: { $_pQ^1$ ⊗ ^1_rQ, ..., $_pQ^{pm}$ ⊗ $^{pm}_rQ$ }
où (rappel) $_pm$ est le nombre d'alternatives de réquisition de PERFORMERS

et ⊗ est l'opération de concaténation (les deux listes –de PERFORMERS et de REOURCES- sont remplacées par une seule liste dont les éléments sont tous les éléments des deux listes).

3. DEVELOPPEMENT de l'OPERATED OBJECT SPECIFICATION :

L'EXPANSION de la SPECIFICATION D'ENSEMBLE D'ENTITES, puis la FONCTION de sélection renvoient une liste ${}_oQ$:
${}_oQ : \{ {}_oQ^1, .. {}_oQ^{om} \}$
avec ${}_om$: nombre d'alternatives de réquisition d'OPERATED OBJECTS
${}_oQ^i : ({}_oi_1, .. {}_oi_{on})$
avec ${}_on$: nombre (commun) d'OPERATED OBJECTS dans toutes les alternatives ${}_oQ^i$

4. Assemblage terminal des trois DEVELOPPEMENTS

On réalise enfin la connexion entre les DEVELOPPEMENTS des trois COMPOSANTS de l'activité primitive, en calculant d'abord le produit cartésien de deux listes :

- celle issue des DEVELOPPEMENTS conjoints de la PERFORMER SPECIFICATION et de l'OPERATION SPECIFICATION,
- celle issue du DEVELOPPEMENT de l'OPERATED OBJECT SPECIFICATION.

$Q' = {}_oQ \otimes {}_{pr}Q$

... puis en en concaténant les listes dans chaque élément du produit. On obtient alors :

$Q : \{q_1, ..., q_{nq}\}$, ou q_k, $k=1,n_q$ est une alternative de réquisition compatible avec la spécification de l'activité primitive.

Annexe 6 : schéma général des mécanismes de simulation d'un système piloté

Légende : MECANISME : mécanisme — MECANISME :: def : définition du mécanisme
EVENEMENT : événement
⇒ PROCESSUS : directive d'exécution d'un processus
→ < action > : action associée à la directive d'exécution d'un processus
dont < sous-action > : partie du contenu d'une action
ENTITE . *méthode*(arguments) : invocation de la méthode d'une entité

SIMULATION ::
CONTROLLED SYSTEM cs = new CONTROLLED SYSTEM() ;
OPERATING SYSTEM os = new OPERATING SYSTEM() ;
STRATEGY s = new STRATEGY() ;
MANAGER ds = new MANAGER(s) ;
PRODUCTION SYSTEM ps = new PRODUCTION SYSTEM (cs, os, ds) ; // construction du système piloté
SIMULATION simulation = new SIMULATION() ; // installation du moteur de simulation
ensemble d'INSTALLATIONS d'ALARMES sur cs
ensemble de PROGRAMMATIONS de

EVENEMENTS sur le PRODUCTION SYSTEM // plusieurs processus aux effets divers
⇒ PROCESSUS CONTINUS sur le PRODUCTION SYSTEM
→ PROCESSUS . *procédure-initialisation*() {// initialisation de PROPRIETES de ps}
→ PROCESSUS . *procédure poursuite*() {// premier changement de VALEUR de ces PROPRIETES}
⇒ PROCESSUS PONCTUELS sur le PRODUCTION SYSTEM
→ PROCESSUS . *procédure-exécution*() {// changement de VALEURS de PROPRIETES de ps }
// y compris mécanismes sur les RESOURCES : IMMOBILISATION, MOBILISATION, RECHARGEMENT

CHECK REACTIVE TRAJECTORY EVENT // *cf.* annexe 7

UPDATING SITUATION EVENT // *cf.* annexe 7

MAKING INSTRUCTION LIST EVENT // *cf.* annexe 7

SIMULATION . *run-simulation*() {
ensemble de PROGRAMMATIONS / OCCURRENCES / AUTOGENERATIONS de

EVENEMENTS sur le PRODUCTION SYSTEM // processus déjà initialisés
⇒ PROCESSUS CONTINUS sur le PRODUCTION SYSTEM
→ PROCESSUS . *procédure poursuite*() {// changement de VALEUR des PROPRIETES
(*) si ALARME sur PROPRIETES
alors INVOCATION REFLEXE éventuelle ::
PROGRAMMATION }

EVENEMENTS sur le PRODUCTION SYSTEM // plusieurs processus aux effets divers
⇒ PROCESSUS CONTINUS sur le PRODUCTION SYSTEM
→ PROCESSUS . *procédure-initialisation*() {// initialisation de PROPRIETES de ps}
→ PROCESSUS . *procédure poursuite*() {// 1er changement de VALEUR de ces PROPRIETES (*) }
⇒ PROCESSUS PONCTUELS sur le PRODUCTION SYSTEM
→ PROCESSUS . *procédure-exécution*() {// changement de VALEURS de PROPRIETES (*) de ps}

CHECK REACTIVE TRAJECTORY EVENT
⇒ CHECK REACTIVE TRAJECTORY PROCESS
→ MANAGER . *check-reactive-trajectory*() {
CHECKING REACTIVE TRAJECTORY :: {
// modification de la STRATEGY s en fonction de l'ETAT du PRODUCTION SYSTEM }

CHECK REACTIVE TRAJECTORY EVENT // *cf.* annexe 7

UPDATING SITUATION EVENT // *cf.* annexe 7

MAKING INSTRUCTION LIST EVENT // *cf.* annexe 7

PROCEED OPERATION EVENT
⇒ PROCEED OPERATION PROCESS(OPERATION ot)
→ ot . *procedure-transition-état*(.) {
// changement de VALEURS de PROPRIETES (*) de ps }
}

Annexe 7 : schéma général des mécanismes de mise en œuvre de la stratégie

Légende : MECANISME : mécanisme MECANISME :: def : définition du mécanisme
EVENEMENT : événement
⇒ PROCESSUS : directive d'exécution d'un processus
→ < action > : action associée à la directive d'exécution d'un processus
dont < sous-action > : partie du contenu d'une action
ENTITE . *méthode*(arguments) : invocation de la méthode d'une entité

SIMULATION :: ensemble de PROGRAMMATIONS / OCCURRENCES / AUTOGENERATIONS de

CHECK REACTIVE TRAJECTORY EVENT
⇒ CHECK REACTIVE TRAJECTORY PROCESS
→ MANAGER . *check-reactive-trajectory*() {
CHECKING REACTIVE TRAJECTORY
}

UPDATING SITUATION EVENT
⇒ UPDATING SITUATION PROCESS
→ MANAGER . *updating-situation*() {
UPDATING SITUATION ::
pour tout block dans STRATEGY faire {
block . NOMINAL PLAN . *update-status*()
dont PROPAGATION
PROGRAMMATION MAKING INSTRUCTION LIST EVENT } }

MAKING INSTRUCTION LIST EVENT
⇒ MAKING INSTRUCTION LIST PROCESS
→ OPERATING SYSTEM . *make-instruction-list*() {
pour tout block dans STRATEGY faire {
MAKING INSTRUCTION LIST ::
NOMINAL PLAN . *expand-to-disjunction*()
dont EXPLICITATION d'une IMPLICIT CONJUNCTION ACTIVITY SPECIFICATION
OPERATING SYSTEM . *set-all-feasible-alternatives*(S)
pour tout jeu inconsistant faire
ALLOCATING RESOURCES :: jeu . *pre-allocate*(.) {
pour toute activité faire
DETERMINATION :: activité . *get-specimens*(.)
dont DEVELOPPEMENTS spec's
dont EXPANSION :: spec. *expand*()
dont INSTANCIATION :: new OPERATION() }
SELECTING INSTRUCTION LIST :: OPERATING SYSTEM . *select-best-alternative*() }
→ PROGRAMMATION ACTING INSTRUCTION LIST EVENT

ACTING INSTRUCTION LIST EVENT
⇒ ACTING INSTRUCTION LIST PROCESS
→ ACTING INSTRUCTION LIST ::
pour tout PACK OF EXECUTABLE INSTRUCTIONS pack faire
OPERATING SYSTEM . *act-instruction-list*(pack)
pour toute OPERATION ot dans pack faire
PROGRAMMATION PROCEED OPERATION EVENT(ot)

PROCEED OPERATION EVENT
⇒ PROCEED OPERATION PROCESS(OPERATION ot)
→ PROCEED OPERATION PROCESS . *procédure-initialisation*() {
dont ot . *degré-de-progression* = 0
dont REQUISITION }
→ PROCEED OPERATION PROCESS . *procédure poursuite*() {
ot . *procedure-transition-état*(.)
si ot . *degré-de-progression* = 1 alors
INSTRUCTION . *turn-to-closed*()
dont LIBÉRATION ::
PROGRAMMATION MAKING INSTRUCTION LIST EVENT
dont PROPAGATION
dont ITERATION si ITERATION ACTIVITY SPECIFICATION }

Annexe 8 : Relations portant sur les dates, entre activités et activités composantes

Les activités d'un PLAN sont organisées en un réseau. Les arcs de ce réseau sont les *relations ensemblistes* (voir § 3.1) entre une activité agrégée (voir § 4.4) et ses activités composantes. La *relation ensembliste* prend un sens spécifique (séquence, conjonction, etc.) selon la *particularisation* de l'activité agrégée.
Les activités sont spécifiées, en particulier, par les ATTRIBUTS suivants, qui la positionnent dans le temps : *début-au-plus-tôt, début-au-plus-tard, fin-au-plus-tôt, fin-au-plus-tard*. La présente annexe précise les règles par lesquelles la valeur de ces ATTRIBUTS pour une activité A est calculée à partir des spécifications fournies au niveau de A, mais aussi au niveau de ses composantes a_0, a_1, ... a_i, ... a_{n-1}, et des activités $m_k$dont A est composante.
De manière générale à toutes les *particularisations*, une date (ici pour exemple le *début-au-plus-tôt*), est calculée comme suit :

```
int ACTIVITY . get-min-beg-date() {
    local = ACTIVITY . début-au-plus-tôt ;
    from_up = ACTIVITY . get-min-beg-date-from-up() ;
    from_down = ACTIVITY . get-min-beg-date-from-down() ; // spécifique à chaque particularisation
    return max(local, from_up, from_down) ;
}
```

```
int ACTIVITY . get-min-beg-date-from-up() {
    result = -9 ;
    POURTOUT activity m_k dont this est composante FAIRE {
        minBeg = m_k . propagate-min-beg-date-to-son(this) ;
        result = max(result, minBeg) ;
    }
    return result ;
}
```

Propagation vers les composantes (*propagate-...-date-to-son*) **:**

	Si la valeur de l'attribut pour A est x (valeur issue de la spécification sur A et des spécifications sur les activités dont A est composante) et si cet attribut n'est pas spécifié pour les composantes de A, ...			
	... cette valeur x est ainsi propagée aux composantes de A :			
classe de A ↓	**min-beg**	**max-beg**	**min-end**	**max-end**
before **unordered before** **meeting** **overlapping** **inclusion**	a_0 : x	a_0 : x	a_{n-1} : x	a_{n-1} : x
costarting **coending** **equality** **conjunction**	$\forall a_i$: x $\forall a_i$: x $\forall a_i$: x $\forall a_i$: x	$\forall a_i$: x / $\forall a_i$: x /	/ $\forall a_i$: x $\forall a_i$: x /x	$\forall a_i$: x $\forall a_i$: x $\forall a_i$: x $\forall a_i$: x
disjunction	$\forall a_i$ non cancelled: x	$\forall a_i$ non cancelled: x	$\forall a_i$ non cancelled: x	$\forall a_i$ non cancelled: x
iteration	a_0 : x	a_0 : x	/	/
optional **const[ed] optional**	a_0 : x	a_0 : x	a_0 : x	a_0 : x

Exploitation de la valeur donnée aux composantes (*get-...-date-from-down*) **:**

	Si la valeur de l'attribut pour la composante a_i est x (valeur issue de la spécification sur a_i et des spécifications sur ses activités composantes) et si cet attribut n'est pas spécifié pour A, ...			
	... la valeur pour A est ainsi calculée :			
classe de A ↓	**min-beg**	**max-beg**	**min-end**	**max-end**
before **unordered before** **meeting** **overlapping**	a_0 . min-beg	a_0 . max-beg	a_{n-1} . min-end	a_{n-1} . max-end
inclusion	a_0 . min-beg	a_0 . max-beg	a_0 . min-end	a_0 . max-end
costarting	max(a_i . min-beg)	min(a_i . max-beg)	max(a_i . min-end)	si $\exists$k tq a_k.max-end= -9 alors ∞ sinon max(a_i.max-end)
coending	si $\exists$k tq a_k.min-beg = -9 alors -∞ sinon min(a_i . min-beg)	min(a_i . max-beg)	max(a_i . min-end)	min(a_i . max-end)
equality	max(a_i . min-beg)	min(a_i . max-beg)	max(a_i . min-end)	min(a_i . max-end)
conjunction	si $\exists$k tq a_k.min-beg = -9 alors -∞ sinon min(a_i . min-beg)	min(a_i . max-beg)	max(a_i . min-end)	si $\exists$k tq a_k.max-end= -9 alors ∞ sinon max(a_i.max-end)
disjunction	min(a_i . min-beg) sur les a_i non cancelled	min(a_i . max-beg) sur les a_i non cancelled	max(a_i . min-end) sur les a_i non cancelled	max(a_i . max-end) sur les a_i non cancelled
iteration	/	/	/	/
optional **const^{ed} optional**	a_0 . min-beg	a_0 . max-beg	a_0 . min-end	a_0 . max-end

Cas d'inconsistance du quadruplet {min-beg, max-beg, min-end, max-end} :

Les inconsistances sont détectées lors de l'affectation d'une valeur à un quelconque des quatre ATTRIBUTS. Lorsqu'une inconsistance est détectée, le mécanisme de SIMULATION est arrêté.
Ci-dessous, chaque paragraphe correspond à l'ATTRIBUT auquel on affecte une valeur.

min-beg :
. min-beg > max-beg
. min-beg > max-end
. SI min-beg > min-end ET min-beg ≤ max-beg ALORS min-end = min-beg

max-beg :
. max-beg < min-beg
. max-beg > max-end

min-end :
. min-end > max-end
. min-end < min-beg

max-end :
. max-end < min-end
. max-end < min-beg
. SI max-end < max-beg ET max-end ≥ min-beg ALORS max-beg = max-end