# Chapitre V- La simulation.

**Objectifs du chapitre :**

- Présenter le mécanisme de simulation à événements discrets mis en place.
- Montrer que la sémantique des composants et des connecteurs est respectée.
- Montrer comment les cycles sont résolus et comment un comportement déterministe est obtenu.

Le chapitre II établit une sémantique temporelle des composants et des connecteurs. Les chapitres III et IV montrent les mécanismes mis en œuvre pour améliorer la lisibilité, la réutilisation et le pouvoir d'expression des modèles. Des modèles complexes peuvent être réalisés par agencement de composants, de connecteurs et par utilisation des mécanismes proposés. En effet, il est facile par composition de services de réaliser des comportements cycliques ou non déterministes.

Dans SEP, le comportement composite est calculé en simulation par des évaluations successives des différents services mis en jeu. Le risque est alors, dans le cas de systèmes cycliques non stables, de ne pas obtenir de résultat et de perdre le contrôle du simulateur, ce qui n'est pas acceptable. Un autre risque vient du fait que les comportements de services concurrents ne peuvent être obtenus que par une évaluation séquentielle des services en concurrence. Le résultat peut alors dépendre de la « sérialisation » des services concurrents, c'est-à-dire de l'ordre dans lequel les services sont exécutés pour obtenir le comportement concurrent.

Ce chapitre montre comment la sémantique temporelle du modèle générique est réalisée en simulation. Il montre comment les résultats sont toujours obtenus en temps fini et borné quelle que soit la configuration et même lorsqu'elle est cyclique. De plus, il montre que les résultats obtenus ne dépendent pas de la *sérialisation* des services concurrents.

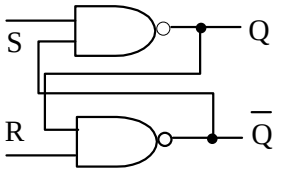


**Figure 1 – Bascule SR.**

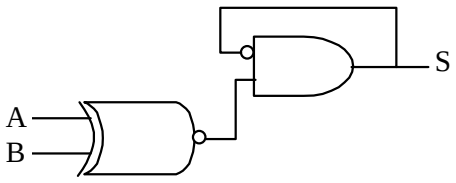


**Figure 2 – Exemple d'un circuit potentiellement instable.**

Le cas de la bascule SR (cf. Figure 1) montre le cas typique d'un système composé de composants combinatoires et dont le comportement est non déterministe selon les scénarios[1]. Le cas présenté par la Figure 2 montre une configuration de systèmes combinatoires où les

[1] Le cas de non déterminisme est le cas où les deux entrées passent simultanément de l'état 0 à l'état 1. Les deux situations de sortie 0,1 et 1,0 sont stables et cohérentes. Le lecteur peut vérifier qu'avec le mécanisme utilisé dans SEP, le résultat ⊥ est obtenu sur les deux sorties ainsi qu'un message d'avertissement.

états transitoires peuvent être différents suivant l'ordre de prise en compte des entrées. Le comportement obtenu selon les scénarios est cyclique ou stable.

La section 1- décrit le moteur de simulation, c'est-à-dire le rôle de l'ordonnanceur ainsi que la procédure que les différents composants suivent en fonction de la nature des services qu'ils rendent. La section 2- montre les différences d'ordonnancement entre les services combinatoires et les services séquentiels rendues obligatoires par leur sémantique. Enfin, la section 3- présente l'algorithme d'ordonnancement utilisé.

## *1- Le moteur de simulation.*

Le moteur de simulation à événements discrets de SEP fonctionne par cycles. L'objectif de l'ordonnanceur, lors d'un cycle, est de calculer la valeur des signaux de sortie de tous les services au temps de simulation courant. En fait, seule la sortie des services dont l'exécution est programmée pendant le cycle sera évaluée. Un cycle commence lorsqu'un ou plusieurs événements sont produits par l'environnement, c'est-à-dire par un composant d'entrée (*InputComponent*) ou un composant de temps (*TimeComponent*). Rappelons que pour SEP un événement associé à un signal est une paire (v, t) où $v \in$ *Value* est la valeur du signal à la date $t \in \Re$.

cycle

Dès que la date d'un événement porté par un signal est égale à la date de simulation courante, cet événement est diffusé vers les ports de destination du signal. Les fonctions de commande associées aux ports déterminent les services dont l'exécution doit être programmée. Remarquons que certains événements ne provoqueront l'exécution d'aucun service. En effet, l'évaluation des fonctions de commandes des composants détermine la sensibilité d'un composant à une série d'événements. Chaque composant définit sa propre sensibilité aux événements qui lui sont fournis.

Les services dont l'exécution est programmée sont classés dans une file d'attente. Le travail de l'ordonnanceur sera d'exécuter les services dans un ordre adapté afin de garantir le déterminisme du comportement. Au début du cycle, un certain nombre de services à exécuter sont programmés, le comportement doit être identique quel que soit l'ordre relatif de programmation des services. Cependant, l'exécution d'un service peut provoquer l'émission d'événements, les événements dont la date est égale à la date de simulation sont propagés dans le même cycle et peuvent ainsi provoquer la programmation de nouveaux services.

A la fin du cycle, la file d'attente est vide, il n'y a plus de service dont l'exécution est programmée.

Trois types de composants fournissent des services.

Tout d'abord, les composants de type *InputComponent* programment l'exécution d'un service lorsqu'un événement d'entrée est produit par l'environnement graphique, c'est-à-dire par exemple lors d'un clic sur un bouton de la souris. Ces événements sont le résultat d'une interaction asynchrone[2] avec l'utilisateur par l'intermédiaire du système d'exploitation et ne peuvent pas être datés de façon déterministe par rapport aux événements en cours de traitement. C'est pourquoi, pour les services de type *InputComponent*, seuls ceux programmés avant le début du cycle sont exécutés pendant le cycle.

En fait, les services des *InputComponent* sont programmés dans une file d'attente séparée. Aucun n'est exécuté pendant le cycle. Lorsqu'il ne reste plus aucun événement non traité, ces services sont mis dans la file d'attente principale.

[2] Le mot asynchrone est utilisé ici pour désigner le fait que l'utilisateur peut vouloir interagir à n'importe quel moment, et ce indépendamment des événements du système.

Intuitivement, cela revient à considérer que tous les événements de ce type se produisent entre la fin d'un cycle et le début du cycle suivant. En fait, ce type de service est assez exceptionnel et c'est ce qui explique la très faible priorité qui lui est associée ; des services de ce type sont programmés pour amorcer la simulation, ou lorsque la simulation se fait en mode pas à pas afin de provoquer le début du cycle suivant.

Ensuite, les composants de type *TimeComponent* programment des services au fur et à mesure que le temps de simulation s'écoule. Ces services permettent de modéliser les comportements périodiques comme, en particulier, celui des horloges. Le système de simulation proposé n'est pas réactif, c'est-à-dire que le temps de simulation évolue de façon non linéaire en fonction de la quantité d'informations à traiter. En pratique, cela signifie que le temps de simulation n'évolue qu'entre la fin d'un cycle et le début du cycle suivant, la durée réelle d'un cycle n'est pas connue à l'avance, l'ordonnanceur fait en sorte qu'elle soit bornée. La gestion des différents services par l'ordonnanceur est détaillée dans la section 2-.

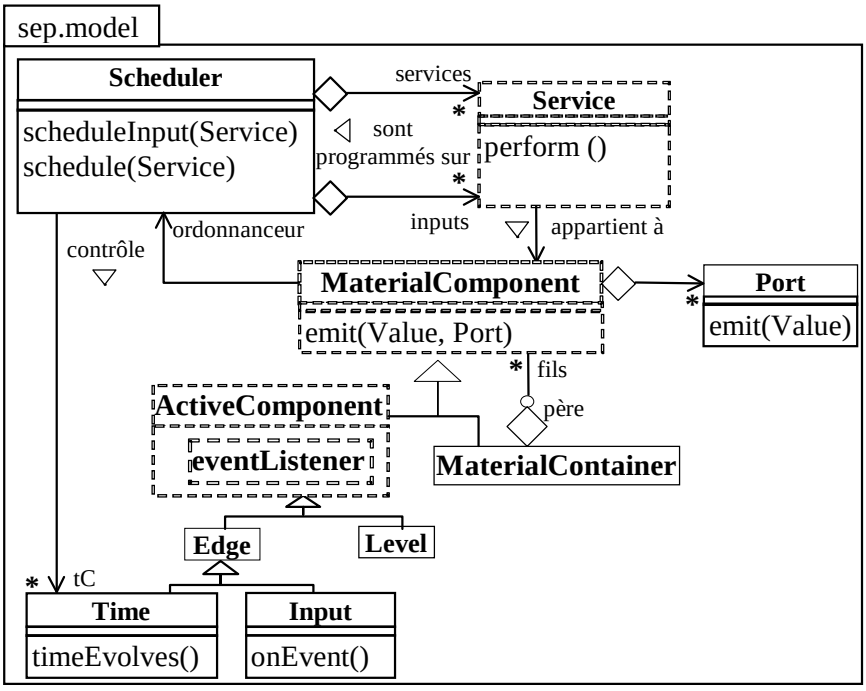


**Figure 3 – Modèle statique d'ordonnancement des services par le Scheduler.**

L'évolution du temps de simulation correspond alors à la plus petite évolution nécessaire pour provoquer la programmation d'un service par un *TimeComponent* ou à la prise en compte d'un événement dont la diffusion a été retardée (cf. chapitre II- section 2-3.). Si aucun *TimeComponent* n'est susceptible de programmer un service et qu'aucun événement retardé n'est présent sur un signal, le temps de simulation n'évolue pas.

Notons que c'est alors le seul cas où un service peut être programmé par un *InputComponent*. Intuitivement, quand tous les événements prévus ont été interprétés et qu'aucun événement retardé n'est présent, alors seulement, l'ordonnanceur prend en compte les événements produits par l'utilisateur.

Enfin, les autres composants programment l'exécution de services en fonction des événements qui leur parviennent.

Un ordonnanceur (*Scheduler*) unique est chargé d'enregistrer la programmation de tous les services, il ordonne l'exécution des différents services afin d'assurer un déterminisme de comportement. La Figure 3 illustre, par un diagramme statique UML, les liens entre le

Scheduler et les composants. La Figure 4 illustre, par un diagramme de séquences UML, les séquences de programmation des services en fonction du type des composants.

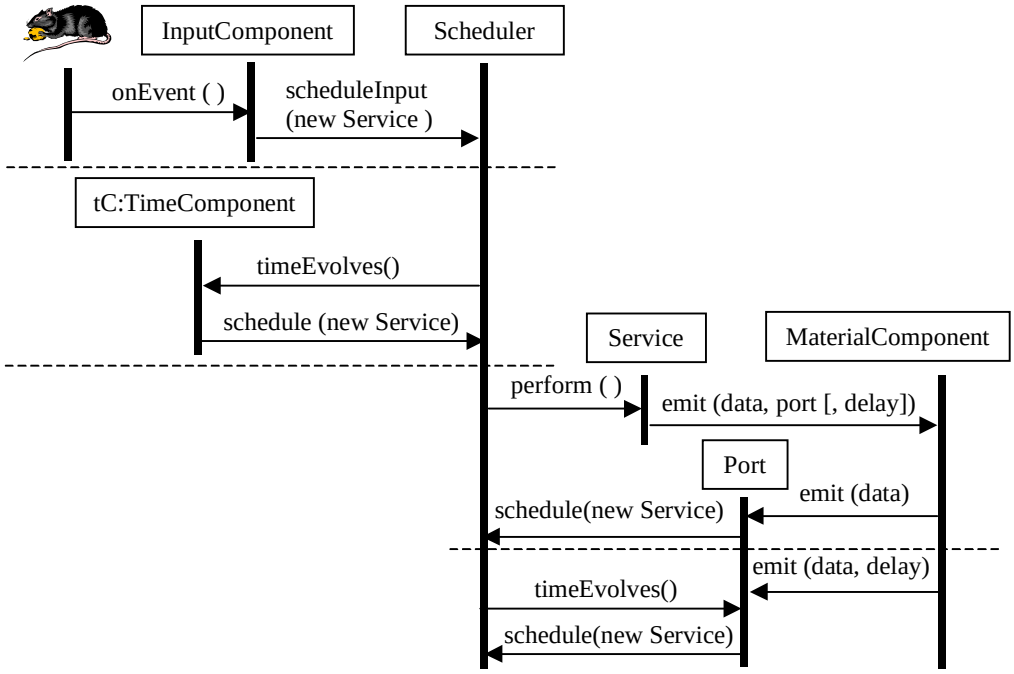


**Figure 4 – Diagramme de séquences UML : programmation des services par les différents types de composants.**

### *2- L'ordonnancement de l'exécution des services.*

En définitive, l'ordonnanceur doit traiter les services programmés, ils sont de deux types : les services séquentiels programmés par les composants *EdgeComponent*, cette classe inclut les services *TimeComponent* et *InputComponent* (cf. Figure 3) ; les services combinatoires programmés par les composants *LevelComponent*.

Au début d'un cycle où la date de simulation est *t*, l'ordonnanceur dispose des valeurs de tous les signaux à des dates strictement inférieures à *t*. De plus, les cycles de simulation précédents ont éventuellement conduit à l'émission d'événements marqués *t*, des services correspondants ont été programmés suivant la fonction de commande associée. Ce sont ces événements et la programmation de ces services qui ont déclenché le début du cycle, et ce sont ces services que l'ordonnanceur doit exécuter pendant ce cycle de simulation. L'objectif est alors de calculer une sortie unique pour tous les services déjà programmés ou qui seront programmés pendant la durée du cycle. La date des événements de sortie produits sera une date supérieure ou égale à *t*. Seuls les événements de sortie datés *t* conduiront à la programmation immédiate de nouveaux services à traiter dans le même cycle de simulation.

#### 2-1. La notion d'instant.

Au début du cycle, l'exécution d'un certain nombre de services est programmée. L'exécution de ces services peut provoquer la programmation de nouveaux services à exécuter dans le même cycle de simulation. Ces phases successives d'exécution de services
instant sont appelés les instants de simulation. L'utilisation d'instants successifs pour la réalisation du comportement, contrairement, à la compilation directe d'un résultat, peut poser des problèmes. Comme le montre la section 2-2, cela ne pose aucun problème pour l'exécution

des services séquentiels. En revanche, cela pose des problèmes pour le traitement des services combinatoires. En effet, les décisions prises en fonction des entrées disponibles peuvent être remises en cause dans l'instant suivant puisque l'état de ces entrées a pu être modifié. La section 2-3 montre comment ces problèmes sont résolus.

## 2-2. Le cas des services séquentiels.

Considérons le cas des services séquentiels. Ils produisent des sorties *o* synchrones avec leur signal de commande c mais à une date strictement supérieure à la date à laquelle les valeurs s d'entrée, utilisées pour calculer la sortie, sont échantillonnées :

si $\mathcal{F}_c(t) \neq \perp$, $\mathcal{F}_O(t) = H(\mathcal{F}_s(t - \varepsilon), \mathcal{F}_e(t - \varepsilon))$, ε est une durée non nulle, aussi petite que l'on veut, pendant laquelle les entrées ne varient pas (cf. sémantique temporelle des services séquentiels énoncée au chapitre II).

Le système de simulation doit permettre de lire la valeur d'un signal *s* à la date courante *t*, ainsi que la valeur précédente à *t* - ε. Ces deux valeurs ne peuvent être différentes que si un événement a été émis sur le signal *s* à la date *t*.

Il apparaît que dans un même instant, l'ordre relatif d'exécution des services séquentiels, les uns par rapport aux autres, n'est pas important. En effet, l'événement de sortie produit par un service quelconque lors d'un instant ne peut être utilisé par un service séquentiel exécuté au cours du même instant, la sortie n'est pas synchrone avec les entrées.

En revanche, le signal de sortie d'un service séquentiel *s1* exécuté à l'instant δ (date t+δ) peut être utilisé comme commande d'un service séquentiel *s2*. Le service séquentiel *s2* sera alors exécuté à l'instant suivant δ+1, c'est-à-dire à une date logique postérieure t+δ+1. Le service s2, peut utiliser les données produites lors de l'instant δ.

Il peut donc exister des boucles de causalité dans les chemins de commande des services séquentiels : l'exécution d'un service séquentiel d'un certain composant provoque la programmation et donc l'exécution d'un service séquentiel, éventuellement du même composant. Ces boucles de causalité sont résolues en interdisant plusieurs exécutions des services séquentiels pendant un cycle de simulation. Si des scénarios conduisent à plusieurs exécutions d'un même service séquentiel, sa sortie est définitivement affectée dans ce cycle de simulation à la valeur ⊥, un message d'avertissement est émis. Notons en outre, que le nombre de services séquentiels est fini et connu au début de la simulation (cf. chapitre III section 2).

Au cours d'un instant, plusieurs services séquentiels du même composant peuvent être programmés, seuls ceux avec la priorité la plus élevée sont effectivement exécutés, les autres sont annulés et un message d'avertissement prévient l'utilisateur. Si plusieurs services ont la même priorité, ils sont exécutés. S'ils émettent sur le même port un événement avec la même date, une fonction de résolution est utilisée pour résoudre le conflit écriture-écriture.

Cependant, au cours d'un même cycle, à des instants différents, des services de priorités différentes peuvent être exécutés.

## 2-3. Le cas des services combinatoires.

Reprenons la sémantique temporelle, énoncée au chapitre II, de l'exécution d'un service combinatoire qui dépend du vecteur d'entrées s et produit la sortie o : $\mathcal{F}_o(t) = H(\mathcal{F}_s(t))$. Les composants combinatoires peuvent donc produire des événements sur un port de sortie

connecté directement[3] ou indirectement à une de leurs entrées et ainsi réaliser des cycles dans les chemins de données. Une fonction cyclique ne réalise pas nécessairement un système instable. L'ordonnanceur a la charge de résoudre les cycles éventuels et de rejeter les configurations où aucun comportement déterministe ne peut être calculé.

Etant donné qu'au temps de simulation *t*, à l'instant $\delta$, la sortie d'un service combinatoire peut être utilisée comme commande d'un service séquentiel, les services combinatoires sont toujours exécutés avant les services séquentiels.

Plus précisément, au début d'un cycle de simulation, un certain nombre de services séquentiels et concurrents sont programmés. Les services combinatoires sont exécutés, leur exécution peut provoquer la programmation de nouveaux services combinatoires ou séquentiels. Seuls les nouveaux services combinatoires sont alors exécutés jusqu'à l'obtention de la stabilité (cf. relation sur les événements marqués).

A un instant donné, les fonctions de commande des services séquentiels ne sont exécutées qu'après l'obtention de la stabilité, c'est-à-dire lorsqu'il n'y a plus de service combinatoire programmé. Tous les services séquentiels programmés sont alors exécutés (cf. section 2-2). Cette exécution peut provoquer la programmation de nouveaux services aussi bien combinatoires que séquentiels, ces services seront exécutés lors de l'instant suivant. L'instant se termine après l'exécution de tous les services séquentiels ce qui marque le début de l'instant suivant. Le cycle se termine lorsque tous les services programmés ont été exécutés.

micro-instant

Les itérations successives nécessaires à l'exécution des services combinatoires définissent des micro-instants. Lors d'un micro-instant, un même service ne peut être programmé qu'une seule fois. La donnée émise par un service combinatoire est un événement marqué. C'est-à-dire un événement – valeur associée à une estampille temporelle qui contient la date et l'instant de simulation – et une marque booléenne { *comb*, *seq* } qui détermine la nature de l'événement.

La marque *comb* signifie que la valeur pourra être remise en cause au cours du cycle, la marque *seq* signifie que la valeur du signal ne sera pas remise en cause.

Un service séquentiel émet l'événement marqué *seq* ; en effet, il ne peut être exécuté qu'une seule et unique fois au cours du cycle de simulation. Les services combinatoires avec retard émettent des événements marqués *seq* ; en effet, leur résultat sera disponible à une date de simulation strictement postérieure, les effets ne sont pas pris en compte dans le même cycle de simulation. Enfin, un service combinatoire sans retard n'émet un événement marqué *seq* que si toutes les événements d'entrée du service sont marqués *seq*. L'événement de sortie est marqué *comb*, si et seulement si au moins un événement d'entrée est marqué *comb*,.

Au cours d'un instant t+$\delta$, un service combinatoire peut émettre plusieurs événements marqués, potentiellement un par micro-instant. Les événements marqués émis par un service combinatoire respectent la relation suivante qui garantit l'obtention de la stabilité à chaque instant :

- soit $( e_i )_{i \in \mathcal{N}}$, la liste des événements marqués émis par un service combinatoire, $e_i = ((v_i, t_i), m_i) \in \mathcal{S} \times \{ comb, seq \}$ ;
- on a alors, $\forall i \in \mathcal{N}$ $t_i < t_{i+1}$ ou ( $t_i = t_{i+1}$ et ($v_i = v_{i+1}$ ou $m_{i+1} = seq$ ou $v_{i+1} = \perp$)).

[3] La cycle peut être indirect si l'événement traverse plusieurs services combinatoires sans retard avant de reboucler sur l'entrée du composant qui l'a émis.

Cela signifie que deux événements consécutifs qui portent une date différente peuvent avoir n'importe quelle valeur, ils sont nécessairement à des instants différents. S'ils sont émis au cours du même instant – à la même date – alors un des cas suivant se produit :

- la valeur est la même, cet événement ne produira aucune modification sur les composants combinatoires qui le reçoivent, seuls les services séquentiels réagiront ;
- la nouvelle marque émise est *seq*, la nouvelle valeur peut donc être quelconque, mais la stabilité sera nécessairement atteinte lorsque tous les services séquentiels auront réagis, puisqu'ils ne peuvent réagir qu'une seule fois par cycle de simulation ;
- aucune des conditions précédentes n'est réunie, la valeur de l'événement est alors forcée à $\perp$, la stabilité est atteinte, un message d'avertissement est émis pour signaler la présence d'un cycle combinatoire non résolu.

La convergence est assurée puisque le nombre d'événements non combinatoires est fini et connu au début de l'instant. Le nombre d'événements marqués émis par un service combinatoire au cours d'un instant est donc borné par N + 2, où N est le nombre de services combinatoires avec retard plus le nombre de services séquentiels. Le temps d'exécution d'un comportement constitué par la composition de plusieurs services combinatoires, éventuellement dans une configuration cyclique, est donc borné et fini.

Enfin, au cours d'un micro-instant, tous les services combinatoires programmés sont exécutés, puis leurs sorties ne sont propagées qu'à la fin du micro-instant. Ainsi, le comportement obtenu ne dépend pas de l'ordre d'exécution des services pendant un micro-instant.

### 2-4. L'action *sleep* du décodeur d'instructions.

Le chapitre IV introduit la possibilité d'associer un comportement complexe à une instruction pseudo-assembleur par l'intermédiaire d'un service séquentiel du décodeur d'instructions et du langage SEP-ISDL. Ce langage propose l'action *sleep* pour séparer logiquement l'exécution de services programmés à une même date physique (cf. chapitre IV section 3-1.).

Cette action correspond exactement à l'instant tel qu'il est défini dans ce chapitre. En fait, elle signifie que l'exécution des services relatifs aux actions qui la suivent est strictement postérieure à l'exécution des services relatifs aux actions qui la précèdent. Le concepteur peut utiliser cette action pour spécifier que deux groupes d'actions doivent être exécutés en séquence, c'est-à-dire à deux instants successifs. Par défaut, les actions sont exécutées en concurrence.

Dans un tel cas, le concepteur ne veut pas préciser la durée physique qui sépare l'exécution des deux ensembles de services afin que le comportement ne dépende pas des dates exprimées dans les composants extérieurs.

Le premier instant du cycle débute avec le début du cycle. Le dernier instant se termine avec la fin du cycle. L'action *sleep* définit le passage d'un instant au suivant. Les services relatifs aux actions qui suivent l'action *sleep* ne seront exécutés que lorsque les services actuellement programmés l'auront été. Ils sont 'logiquement' postérieurs aux services actuellement programmés.

## *3- L'algorithme d'ordonnancement.*

Les choix expliqués en détail dans les sections précédentes sont implémentés par l'algorithme d'ordonnancement énoncé ci-après :

date_de_simulation = 0
**Répéter**
-- Début du cycle

**Répéter tant qu**'il y a des services programmés.
-- Début d'un instant

**Répéter tant qu**'il y a des services combinatoires programmés.
-- Début d'un micro-instant
**Exécuter** tous les services combinatoires programmés.
**Propager** les événements marqués sur les ports de sortie.
**Programmer** les services combinatoires en fonction des événements propagés.
-- Fin d'un micro-instant.
**Fin répéter**.

**Évaluer** les fonctions de commande des composants séquentiels.
**Programmer** en conséquence l'exécution de services séquentiels.
**Exécuter** tous les services séquentiels programmés.
**Propager** les événements sur les ports de sortie.
**Diffuser** sur les signaux vers les ports de destination.
**Programmer** en conséquence l'exécution des services combinatoires et séquentiels.

-- Fin d'un instant.
**Fin répéter**.

-- Fin du cycle de simulation.

**Si** il y a des événements à retard
**Faire évoluer la date de simulation** en fonction du retard le plus faible.
**Propager** les événements à retard correspondants.
**Programmer** les services éventuellement associés.
**Sinon**
**Attendre** qu'il y ait des services programmés par un *InputComponent*.
**Mettre** dans la file d'attente tous les services *InputComponent* programmés au cours du cycle précédent[4].
**Fin si**.

**Tant qu**'il y a des services programmés.

En conclusion, ce chapitre présente la structure d'objets mise en place pour garantir l'autonomie des composants pendant la simulation. Les objets réagissent indépendamment de la configuration à laquelle ils appartiennent, sur ordre d'un ordonnanceur centralisé. La sémantique temporelle des composants définie au chapitre II motive les choix d'implémentation d'un moteur de simulation à événements discrets.

Notamment, la sortie asynchrone des sorties d'un service séquentiel par rapport à ses entrées nous permet un ordonnancement simple. Tandis que l'ordonnancement des services combinatoires, dont les sorties sont synchrones avec les entrées, se fait par une recherche de point fixe par itérations successives. Cette recherche de point fixe conduit au rejet en simulation de configurations dont le simulateur ne parvient pas à donner un comportement en accord avec la sémantique définie. Le concepteur est alors invité à résoudre les cycles combinatoires en utilisant des services combinatoires à retard ou en ajoutant des composants séquentiels.

[4] Tous les services *InputComponent* programmés pendant un cycle sont considérés comme synchrones et sont exécutés lors du cycle suivant.