HPTS++

Manuel utilisateur

26 mai 2003

Table des matières

Introduction

HPTS++ (Hierarchical Parallel Transition System) est un langage accompagné d'une API C++ permettant de décrire des systèmes multi agents. Dans ce système, chaque agent est décrit par l'intermédiaire d'un automate. Initialement, ce langage a été développé dans le cadre de l'animation comportementale pour permettre la description de la partie réactive du comportement d'un humanoïde virtuel. Le but était de fournir un langage permettant de décrire des comportements (sous forme d'automates) mais aussi d'offrir des mécanismes permettant une synchronisation automatique de ces derniers tout en exploitant, si possible, des possibilités d'adaptation des comportements lorsqu'ils s'exécutent en même temps (répartition de l'attention visuelle, concurrence sur les mains pour l'interaction avec des objets...). Pour ce faire, un certain nombre de concepts ont été inclus :

- **Ressources**. Le système est doté d'une synchronisation par sémaphores pour permettre l'exclusion mutuelle des agents sur des ressources. Dans ce cadre, un mécanisme de détection d'inter-blocages a été mis en oeuvre.
- **Priorités**. Chaque agent du système possède une fonction de priorité dynamique permettant de décrire l'importance de l'agent dans un contexte donné.
- **Adaptation**. Un mécanisme de description des différentes possibilités d'adaptation du déroulement d'un automate en fonction de la disponibilité des ressources ou des besoins est offert.

Ces différents concepts sont utilisés par le système d'ordonnancement, fourni avec HPTS++. Ce dernier adapte automatiquement l'exécution des différents automates en cours d'exécution, en fonction de leur priorité respective et des disponibilités et besoins en terme de ressources.

Le langage fourni pour décrire les automates consiste en une encapsulation de code C++ à l'intérieur de blocs prédéfinis permettant de décrire des automates. Chaque automate, une fois compilé, est transformé en une classe C++ manipulable en tant que telle. De cette manière, tous les concepts orientés objet de C++ sont accessibles via le langage (héritage, polymorphisme...). Cette propriété rend l'interfaçage d'HPTS++ avec des applications C++ immédiat, en offrant au programmeur la possibilité d'inclure et manipuler ses propres classes directement dans les automates.

Chapitre 1

Approche générale

Cette section a pour but de donner un certain nombre de bases du système HPTS++. Elle n'est donc pas spécialisée mais expose un certain nombre de points permettant de se familiariser à HPTS++.

1.1 Architecture globale

HPTS++ est fourni avec un langage de description d'automates basé sur C++ et un environnement d'exécution de ces automates.

1.1.1 Processus de compilation

Les automates décrits par l'intermédiaire du langage sont dans un premier temps compilés via HPTScompile, le compilateur d'automate. Ce processus de compilation génère deux fichiers C++, le fichier entête (.h) et le fichier source (.cxx) portant le nom de la classe d'automate décrite dans le fichier source d'automate (Cf. fig. 1.1).

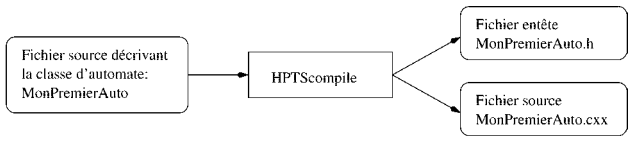

FIG. 1.1 – *Chaîne de compilation*

Les fichiers sources ainsi générés peuvent ensuite être compilés comme tout fichier C++, en ajoutant dans les chemins d'inclusion les répertoires de HPTS++. Les fichiers générés par le compilateur contiennent les informations sur les numéros de ligne du fichier source en langage HPTS. Les erreurs détectées par le compilateur sont donc indiquées dans le fichier HPTS. Chaque automate est généré sous forme d'une classe C++ utilisable comme telle à l'intérieur du programme.

1.1.2 Architecture d'exécution

HPTS++ offre un environnement d'exécution d'automates configurable. Il est possible d'exécuter plusieurs automates parallèles sous forme de hiérarchie ou non, avec une possibilité de mélange. L'exécution d'un ensemble d'automates est gérée par l'intermédiaire d'un contrôleur (Cf. fig 1.2).

Rôle du contrôleur
Les rôles assumés par un contrôleur sont les suivants :

1. Il gère l'exécution de tous les automates qui lui sont associés en les ordonnançant. C'est donc lui qui permet de demander l'exécution d'un nouvel automate.

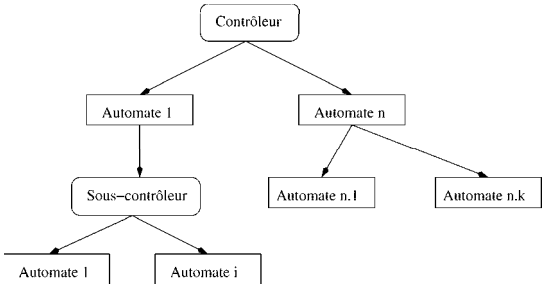

FIG. 1.2 – *Une vue globale de l'architecture d'exécution des automates.*

2. Il fournit des instances de ressources à ses automates associés sur demande. Ces ressources sont des sémaphores, deux automates ne peuvent donc pas posséder la même ressource au même pas de temps.
3. Il se charge de la gestion des signaux envoyés aux automates. Ces signaux peuvent être de trois types :
   - **kill** : signal servant à tuer l'automate. Ce dernier stoppe son exécution qui ne sera jamais reprise.
   - **suspend** : signal de suspension d'exécution. L'automate stoppe son exécution temporairement en attendant d'être redémarré via un signal resume.
   - **resume** : signal de reprise d'exécution d'un automate ayant préalablement reçu un signal suspend.

Il est possible d'instancier des sous-contrôleurs locaux à un automate. De cette manière, un automate peut se charger de la supervision de l'exécution de sous-automates parallèles via le contrôleur local, permettant ainsi de décrire une concurrence locale entre les automates associés au sous-contrôleur (Cf. fig. 1.2).

Notion de hiérarchie d'automates

La notion de hiérarchie d'automates sert à décrire un automate sous la forme du produit de plusieurs sous-automates fonctionnant en parallèle. Dans ce cas, l'automate père de la hiérarchie peut être vu comme un filtre, synthétisant un résultat à partir des propositions envoyées par ses fils.

La différence entre des automates parallèles et des automates hiérarchiques se situe sur deux points :

1. Les automates fils d'un automate s'exécutent avant l'automate père, pour permettre à l'automate père de synthétiser de l'information à partir des résultats du calcul des automates fils.
2. Lorsque l'automate père reçoit un signal, ce dernier se propage au niveau des automates fils.

La manipulation d'une hiérarchie d'automates se résume donc à la manipulation de l'automate père de la hiérarchie. Cette notion permet, notamment, de décrire implicitement un automate comme le produit de plusieurs automates.

1.1.3 Etat d'exécution des automates

Un automate lorsqu'il a été créé et associé à un contrôleur en charge de son exécution, peut se trouver dans quatre états:

- **running** : l'automate est actuellement en cours d'exécution.
- **suspended** : l'automate est actuellement suspendu dans son exécution. Il n'est donc plus utilisé mais en attente de reprise d'exécution.
- **killed** : l'automate a été tué. Il n'est donc plus exécuté.
- **ended** : l'automate s'est terminé en arrivant dans un état de terminaison. Il n'est donc plus exécuté.

1.2 Ordonnancement des automates

A l'origine, le système d'ordonnancement fourni avec HPTS++ a été conçu dans le but de permettre le mélange et la synchronisation automatique de comportements en fonction de leur importance respective et des ressources dont ils ont besoin. Chaque comportement étant décrit sous la forme d'automates, plusieurs

concepts ont été ajoutés dans leur description : les ressources, les degrés de préférence et les priorités. Les ressources permettent de décrire les exclusions mutuelles entre les différents automates alors que les degrés de préférence sont utilisés pour décrire différentes possibilités d'adaptation en fonction de la disponibilité ou des besoins en terme de ressources. En utilisant cette information, il devient possible de créer un système d'ordonnancement permettant de synchroniser automatiquement les automates en fonction de leur priorité respective.

1.2.1 Ressources

Pour chaque automate exécuté par HPTS++, un ensemble de ressources est défini. Ces ressources sont des sémaphores utiles pour l'exclusion mutuelle. Lors de la description d'un automate, l'utilisateur peut associer un ensemble de ressources à chaque état de l'automate. Les ressources étant des sémaphores, leur disponibilité conditionne les transitions dans cet état. Leur allocation, de par le mode de description, est automatique :

- L'entrée dans un état provoque la prise des ressources qui lui sont associées.
- La sortie de l'état provoque la relâche des ressources associées.
- Les ressources sont conservées si une transition connecte deux états utilisant ces ressources.

Alors que plusieurs automates sont en concurrence sur le même ensemble de ressources, des inter-blocages peuvent arriver. Pour faciliter la description des automates et ne pas avoir à gérer ce problème, un système de calcul de dépendances de ressources est fourni. Ce dernier permet de détecter un éventuel inter-blocage avant que ce dernier arrive, l'entrée dans un état est donc aussi conditionnée par la possibilité d'inter-blocage.

1.2.2 Degrés de préférence

La notion de degré de préférence a été ajoutée pour permettre de décrire différentes possibilités d'adaptation de l'exécution d'un automate en fonction des disponibilités des ressources. Un degré de préférence est un coefficient réel associé à chaque transition d'un automate. Il décrit la propension d'un automate à utiliser cette transition. En fonction de sa valeur, il a différentes significations, notons p le degré de préférence :

- $p > 0$: cette transition favorise la réalisation de l'action associée à l'automate. Par défaut, la transition possédant le plus grand degré de préférence devrait être choisie.
- $p < 0$: cette transition ne favorise pas la réalisation de l'action associée à l'automate. Ces transitions sont utilisées pour décrire une façon cohérente d'adapter l'exécution de l'automate en relâchant des ressources. Par défaut, cette transition n'est pas franchie.
- $p = 0$: le franchissement de cette transition n'influe pas sur le déroulement de l'automate.

Ce coefficient, associé à chaque transition, permet de concentrer à l'intérieur d'un seul automate toute l'information sur ses différentes possibilités d'adaptation en fonction de disponibilités ou demandes en terme de ressources.

1.2.3 Fonction de priorité

La notion de priorité a été introduite pour permettre de spécifier l'importance d'un automate à chaque pas de temps. Une fonction de priorité est donc associée à chaque automate. Cette fonction renvoie une valeur réelle représentant l'importance de l'automate dans un contexte donné. En fonction de son signe, le résultat de cette fonction a plusieurs significations :

- priorité> 0 : cet automate doit être exécuté et est adapté au contexte. Cette valeur peut être interprétée comme un coefficient d'adéquation entre le contexte et l'automate.
- priorité< 0 : cet automate est inhibé, cette valeur peut être interprétée comme un coefficient d'inadéquation entre l'automate et le contexte.

1.2.4 Ordonnancement

Les notions précédentes sont utilisées pour créer un système d'ordonnancement des automates permettant de favoriser l'exécution des automates de plus forte priorité en adaptant automatiquement l'exécution des autres automates si cela est possible.

A chaque pas de temps, l'ordonnanceur collecte des informations sur tous les automates. Cette information est composée de toutes les possibilités de transition d'un automate ; l'état courant ainsi que tous les états extrémité des transitions franchissables depuis l'état courant. Pour chacune de ces propositions de transition l'ordonnanceur calcule un poids, notons *prio* la priorité associée à l'automate pour ce pas de temps et p le degré de préférence associé à la transition permettant d'atteindre un état. Ce poids (W) est calculé de la manière suivante :

$$W = prio \times p \tag{1.1}$$

Considérons un poids W associé à une proposition, ce poids a différentes significations :

- $W > 0$: l'automate est enclin à transiter dans l'état correspondant à cette proposition. Deux cas sont à distinguer:
  - $(prio > 0) \wedge (p > 0)$: le fait de transiter dans cet état favorise la réalisation de la tâche associée à l'automate.
  - $(prio < 0) \wedge (p < 0)$: l'automate est inhibé, les transitions conduisant à un arrêt cohérent de l'automate doivent être favorisées.
- $W < 0$: l'automate n'est pas enclin à transiter dans l'état associé à la transition. Deux cas sont à distinguer:
  - $(prio > 0) \wedge (p < 0)$: le fait de transiter dans l'état associé à cette proposition ne favorise pas la réalisation de la tâche associée à l'automate. Ce cas est utilisé pour proposer des possibilités d'adaptation du déroulement de l'automate en relâchant des ressources. Cette proposition peut être utilisée si un automate possédant une priorité plus forte a besoin des ressources libérées par l'exploitation de cette proposition.
  - $(prio < 0) \wedge (p > 0)$: l'automate est inhibé, ce type de proposition peut être utilisé pour proposer d'autres moyens de stopper l'exécution du comportement en utilisant moins de ressources.
- $W = 0$: l'automate est indifférent au fait d'exploiter cette proposition.

Une fois que les propositions de transition et leurs poids sont calculés, l'ordonnanceur recherche une combinaison de propositions respectant les contraintes suivantes :

- Pas de conflit de ressources, la combinaison ne contient pas deux états utilisant des ressources en commun.
- Pas de possibilité d'inter-blocage.
- La somme des poids associés aux propositions constituant la combinaison est maximale.

Finalement, les automates possédant la plus forte priorité sont favorisés dans leur exécution alors que les automates ayant des priorités moins fortes vont relâcher leurs ressources de façon cohérente. La cohérence de l'ordonnancement est assurée car l'ordonnanceur ne peut utiliser que des propositions de transition venant des automates et qui sont supposées être cohérentes.

Ce système permet de décrire des automates de façon indépendante tout en assurant leur bon déroulement, et ce, grâce à la gestion de l'exclusion mutuelle via les ressources et au mécanisme de détection des inter-blocages potentiels. De part la description des degrés de préférence associés aux transitions l'adaptation du déroulement des automates à la disponibilité ou aux besoins en terme de ressources est automatique.

1.3 Description des automates

Pour offrir un maximum de liberté dans la description des automates et leur utilisation, un certain nombre de points-clef dans l'exécution ont été repérés. A chacun de ces points clefs, HPTS++ offre la possibilité d'ajouter du code pour permettre d'effectuer les actions voulues par l'utilisateur. En plus de ces points clef, un certain nombre d'autres définitions ont été ajoutées pour permettre la gestion de la concurrence des

automates. La liste suivante a pour but de donner une vision générale des points accessibles à l'utilisateur lors de la description des automates:

1.3.1 Les informations associées aux automates

- **Fonction de priorité**. Il s'agit d'une fonction numérique pouvant évoluer au cours du temps et permettant de stipuler l'importance d'un automate à chaque pas de temps.
- **Méthodes de réaction aux signaux**. Il s'agit de méthodes permettant de réagir au niveau de l'automate à la réception d'un signal tel que kill/suspend/resume.
- **Code à exécuter avant un pas de temps**. Il s'agit d'un code associé à l'automate qui est exécuté en début de chaque pas de calcul.
- **Code à exécuter après un pas de temps**. Il s'agit d'un code associé à l'automate qui est exécuté en fin chaque pas de calcul.
- **Ensemble d'états**. Il s'agit des différents états associés à l'automate.
- **Ensemble de transitions**. Il s'agit des différentes transitions associées à l'automate.
- **Etat initial**. Il s'agit de l'état de départ de l'automate.

1.3.2 Les informations associées aux états

- **Ressources utilisées**. Ensemble des ressources utilisées dans cet état. Si toutes ces ressources ne sont pas libres ou libérables, l'automate ne pourra pas transiter dans cet état. Il est à noter que les ressources utilisées par un état sont évaluées uniquement lors du lancement de l'automate. Il n'est donc pas possible de changer les ensembles de ressources utilisées par un état de l'automate alors que ce dernier est en cours d'exécution.
- **Code d'entrée dans l'état**. Code à exécuter lorsqu'une transition provoque l'entrée dans l'état décrit.
- **Code à exécuter en restant dans l'état**. Code à exécuter tant que l'on reste dans l'état ; c'est-à-dire tant qu'aucune transition possédant cet état comme origine n'est franchie.
- **Code de sortie dans l'état**. Code à exécuter lorsqu'une transition provoque la sortie de l'état décrit.
- **Méthodes de réaction aux signaux**. Code permettant de réagir au niveau de l'état à la réception d'un signal de type kill/suspend/resume.

1.3.3 Les informations associées aux transitions

- **Etat d'origine**. Etat depuis lequel la transition peut être considérée comme franchissable.
- **Etat d'extrémité**. Etat dans lequel se trouvera l'automate si la transition décrite est franchie.
- **Préférence**. Expression renvoyant un nombre flottant stipulant le degré de préférence associé à cette transition.
- **Condition**. Expression booléenne conditionnant le franchissement de la transition. Si cette expression est fausse, la transition ne pourra pas être franchie.
- **Action**. Code à exécuter lorsque la transition est franchie.

Chapitre 2

Description d'un automate avec HPTS++

2.1 Classe d'automate

Lorsque l'automate est compilé, une classe C++ correspondant à ce dernier est générée. Pour des raisons de simplicité, elle porte le nom associé à l'automate lors de sa description.

2.1.1 Notion d'héritage

Dans l'exemple ci-dessous, on stipule que l'on va créer une classe d'automate *ClasseAutomate*.

```
state machine ClasseAutomate
{
    // Corps de l'automate
}
```

D'autre part, il existe une notion d'héritage à l'intérieur du langage. Cet héritage se décline en deux types :

- l'héritage d'automate.
- l'héritage de classe C++.

Héritage d'automate L'exemple ci-dessous montre l'automate *ClasseAutomate* précédent héritant maintenant de l'automate *AutreAutomate*.

```
state machine ClasseAutomate : state machine AutreAutomate
{
    // Corps de l'automate
}
```

Héritage de classe L'exemple ci dessous montre l'automate *ClasseAutomate* précédent héritant maintenant de la classe *ClasseCpp*

```
state machine ClasseAutomate : class ClasseCpp
{
    // Corps de l'automate
```

```
}
```

Il est aussi possible de faire de l'héritage multiple avec des automates et des classes C++. Dans ce cas chaque automate/classe est séparé par une virgule dans l'entête de l'automate. En reprenant les deux exemples précédents pour créer l'automate *ClasseAutomate* héritant de l'automate *AutreAutomate* et de la classe *ClasseCpp*, la syntaxe est la suivante :

```
state machine ClasseAutomate : state machine AutreAutomate, class ClasseCpp
{
    // Corps de l'automate
}
```

2.1.2 Données membres et méthodes

Un automate est traduit par une classe C++, il est donc possible de définir des données membres associées à ce dernier ainsi que des méthodes, des constructeurs et destructeurs. Ces définitions se font dans une section appelée *variables* avec la syntaxe de C++, l'exemple ci-dessous montre l'ajout d'un constructeur par défaut et d'un destructeur (virtuel) à l'automate *ClasseAutomate* :

```
state machine ClasseAutomate
{
    variables
    {{
    public:
        ClasseAutomate() ; // Constructeur
        virtual ~ ClasseAutomate() ; // Destructeur (nécessairement virtuel)
    }}
// Autres définitions liées à l'automate
}
```

Relation avec l'API C++
Lors de la compilation, le code décrit dans la partie **variables** de l'automate est regénéré tel quel dans le fichier header associé à la classe d'automate.

2.1.3 Factorisation de code au niveau de l'automate

HPTS++ offre à l'utilisateur le moyen de décrire des actions à exécuter systématiquement avant un pas de temps ou après un pas de temps. Ceci se fait par l'intermédiaire des mots-clefs **before step** pour le code à exécuter avant un pas de temps et **after step** pour le code à exécuter après un pas de temps. La syntaxe est donc la suivante :

```
state machine ClasseAutomate
{
    variables
    {{
    }}
    before step
    {{
        // Code C++ à exécuter avant le pas de temps
    }}
    after step
    {{
        // Code C++ à exécuter après le pas de temps
```

```
    }}
// Autres définitions liées à l'automate
}
```

Relation avec l'API C++
Lors de la compilation, deux méthodes correspondant au code à exécuter avant/après un pas de temps sont générées:

virtual void **beforeStep**()
virtual void **afterStep**()

2.1.4 Priorité

La priorité d'un automate est décrite par l'intermédiaire d'une expression C++ retournant un résultat de type float. Le mot-clef permettant de décrire la priorité est **priority**. La syntaxe est la suivante:

```
state machine ClasseAutomate
{
    variables
    {{
    }}
    priority
    {{
        // Expression C++ renvoyant un résultat de type float
    }}
}
```

Relation avec l'API C++
A la génération de chaque automate, la priorité est encapsulée dans une méthode de la classe d'automate. Son interface est la suivante:

virtual float **priority**()

2.1.5 Etat initial et terminaison de l'automate

Etat initial

L'état initial de l'automate se décrit à l'aide du mot-clef **initial state**. L'expression associée est une expression C++ renvoyant un résultat de type **HPTS::State ***.

La syntaxe est la suivante:

```
state machine ClasseAutomate
{
    variables
    {{
    }}
    initial state
    {{
        // Expression C++ renvoyant un résultat de type HPTS::State *
    }}
}
```

Relation avec l'API C++
L'état initial d'un automate peut être ensuite récupéré par l'intermédiaire de la méthode suivante:

virtual HPTS::State * entry()

Etat final

La notion d'état final n'a pas été introduite dans la grammaire d'HPTS++. En effet, l'héritage d'automate étant possible, il devient donc aussi possible d'ajouter des transitions à l'intérieur des automates. La notion d'état final devient donc difficile à gérer dans ce cas, car un état peut être considéré comme final dans un cas et pas dans l'autre. Le choix a été de considérer qu'un état est un état final lorsqu'il ne possède pas de transition sortante.

2.2 Etats et transitions

2.2.1 Description des états

Un état comporte plusieurs parties paramétrables correspondant à plusieurs moments dans l'exécution et à l'utilisation des ressources dont voici la liste:

- **Resources utilisées**. Il s'agit de l'ensemble des ressources nécessaires lors du passage dans cet état. Elles se décrivent par l'intermédiaire du mot-clef **uses**.
- **Entrée dans l'état**. Il s'agit du code à exécuter lorsqu'une transition provoque l'entrée dans l'état décrit. Ce code n'est exécuté qu'à ce moment. Il se décrit par l'intermédiaire du mot-clef **entry**.
- **En restant dans l'état**. Il s'agit du code à exécuter à chaque pas de temps tant que l'automate reste dans cet état, c'est-à-dire tant qu'aucune transition sortante de l'état n'est sélectionnée. Il se décrit par l'intermédiaire du mot-clef **during**.
- **Sortie de l'état**. Il s'agit du code à exécuter lorsque l'automate sort de l'état décrit, donc lorsqu'une transition sortant de l'état est sélectionnée pour être franchie. Il se décrit par l'intermédiaire du mot clef **exit**.

Chacune de ces parties est optionnelle dans la définition de l'état. Si une partie n'est pas spécifiée, par défaut elle ne fait rien. La définition d'un état se fait par l'intermédiaire de la syntaxe suivante:

```
state identifiantEtat
{
    uses
    {{
    // Expression de type std::list<HPTS::Resource*>  ou HPTS::Resource*
    }} ( ,
    {{
    // Expression de type std::list<HPTS::Resource*>  ou HPTS::Resource*
    }} )*;
    entry
    {{
        // Code C++ exécuté lors de l'entrée dans l'état
    }}
    during
    {{
        // Code C++ exécuté en restant dans l'état
    }}
    exit
    {{
        // Code C++ exécuté lors de la sortie de l'état
    }}
```

}

Il est à noter que la syntaxe pour l'utilisation des ressources est un peu particulière. Cette dernière peut être une liste d'expressions C++ retournant des valeurs de type **std::list<HPTS::Resource*>** ou bien de type **HPTS::Resource***, le symbole "**;**" servant de marqueur de fin de liste.

Relation avec l'API C++.
Pour chaque état décrit dans l'automate, le compilateur génère une méthode dont l'entête est le suivant :

virtual HPTS::State * **identifiantEtat**()

Cette méthode permet de récupérer et de manipuler l'état à l'intérieur de l'automate. La classe **HPTS::State** possède des méthodes permettant d'appeler le code associé à ces parties de l'état.

virtual std::list<HPTS::Resource*> HPTS::State::**resources**()
virtual void HPTS::State::**entry**()
virtual void HPTS::State::**during**()
virtual void HPTS::State::**exit**()

2.2.2 Description des transitions

Les transitions d'un automate comportent plusieurs parties paramétrables :
- **Origine**. Il s'agit de l'état d'origine de la transition.
- **Extrémité**. Il s'agit de l'état d'extrémité de la transition.
- **Préférence**. Il s'agit d'une expression C++ renvoyant un "float" et stipulant la préférence associée à cette transition.
- **Condition**. Il s'agit d'une expression C++ renvoyant un "bool" conditionnant le franchissement de la transition. Si cette expression est fausse, la transition est infranchissable.
- **Action**. Il s'agit de code C++ décrivant l'action associée au franchissement de cette transition.

Les transitions sont décrites par l'intermédiaire de la syntaxe suivante:

transition identifiantTransition
{
origin
{{
*// état d'origine de la transition, expression C++ de type HPTS::State **
}}
extremity
{{
*// état d'extrémité de la transition, expression C++ de type HPTS::State **
}}
preference
{{
// préférence associée à la transition, epression C++ de type float
}}
condition
{{
// condition associée à la transition, expression C++ de type bool
}}
action
{{

```
        // action associée à la transition, Code C++
    }}
}
```

Relation avec l'API C++.
Pour chaque transition, le compilateur génère une méthode dont l'entête est le suivant :

virtual HPTS::Transition * **identifiantTransition**()

Cette méthode permet de récupérer et de manipuler une transition à l'intérieur de l'automate. La classe **Transition** possède des méthodes permettant de récupérer les informations associées à une transition :

virtual HPTS::State * HPTS::Transition::**origin**()
virtual HPTS::State * HPTS::Transition::**extremity**()
virtual float HPTS::Transition::**preference**()
virtual bool HPTS::Transition::**condition**()
virtual void HPTS::Transtion::**action**()

2.2.3 Redéfinition des transitions et des états

Lors de la compilation, un ensemble de méthodes virtuelles portant le nom des états et des transitions sont générées (Cf. 2.2.1, 2.2.2). De ce fait, il devient possible de redéfinir les états et les transitions associés à un automate via le mécanisme d'héritage. Pour ce faire, il suffit de redéclarer un état/transition portant le même nom qu'un état/transition d'un automate ancêtre.

Le redéfinition des états et transitions est transparente. Dans le cas d'une transition, si les états d'origine et/ou d'extrémité ont été redéfinis, ces transitions les utiliseront automatiquement. De la même manière, lorsqu'une transition est redéfinie, seule sa dernière définition est prise en compte lors de l'exécution.

2.3 Réaction aux signaux

HPTS++ offre la possibilité de contrôler l'exécution des automates par l'intermédiaire de signaux. il existe trois signaux :

- *kill*, ce signal correspond à une destruction de l'automate en cours d'exécution.
- *suspend*, ce signal correspond à une suspension d'exécution de l'automate.
- *resume*, ce signal correspond à la reprise de l'exécution de l'automate après une suspension (signal *suspend*)

L'envoi de ces signaux est géré via le contrôleur (voir section correspondante). Dans la mesure où ces signaux peuvent intervenir n'importe quand lors de l'exécution d'un automate, des codes de réaction à ces derniers sont répartissables dans la description de l'automate. Il est possible de réagir à deux niveaux (combinables) lors de la réception d'un signal, au niveau de l'état actuel de l'automate et au niveau de l'automate.

2.3.1 Réaction au niveau de l'automate

.
La syntaxe est la suivante :

```
state machine ClasseAutomate
{
    variables
    {{
        // Données membres et méthodes associées aux automates.
```

```
    }}

    kill {{Code de réaction au signal kill}}
    suspend {{Code de réaction au signal suspend }}
    resume {{Code de réaction au signal resume }}

    // Autres définitions liées à l'automate
}
```

Il est à noter que ces codes de réaction aux signaux sont facultatifs, s'ils ne sont pas définis rien ne se passera, s'ils sont définis dans une classe automate mère de l'automate courant, le code de réaction de la classe mère sera appelé.

Relation avec l'API C++
A l'intérieur de la classe correspondant à chaque automate, des méthodes associées aux réactions par rapport aux signaux sont présentes :

virtual void **kill**()
virtual void **suspend**()
virtual void **resume**()

2.3.2 Réaction au niveau de l'état

.

Un état possède lui aussi trois champs permettant de décrire une réaction à un signal particulier. La syntaxe au niveau de l'état est la suivante :

```
state identifiantEtat
{
    entry {{ }}
    during {{ }}
    exit {{ }}
    kill
    {{
        // Code C++ exécuté lors de la reception d'un signal kill
    }}
    suspend
    {{
        // Code C++ exécuté lors de la reception d'un signal suspend
    }}
    resume
    {{
        // Code C++ exécuté lors de la reception d'un signal resume
    }}
}
```

Relation avec l'API C++
Comme décrit au paragraphe 2.2.1 la méthode suivante est générée pour la récupération des états :

virtual HPTS::State * **identifiantEtat**()

En plus des méthodes de la classe **HPTS::State** décrites au paragraphe 2.2.1, les méthodes suivantes permettant d'appeler le code associé aux signaux :

virtual void HPTS::State::kill()

virtual void HPTS::State::suspend()
virtual void HPTS::State::resume()

2.4 Structure globale d'un automate

La structure globale d'un automate est la suivante :

header
{{
// Include de fichiers et définitions globales à la mode C++
}}

state machine Identifiant **: state machine** Identifiant, **class** Identifiant
{

variables
{{
// Attributs de la classe d'automate
// Code C++ comprenant le constructeur de l'automate
// Code C++ comprenant le destructeur(virtuel de l'automate)
// Code C++ comprenant les méthodes associées aux automates
}}

// Priorité de l'automate (facultatif)
priority {{Expression renvoyant un double }}
// Code a exécuter avant le pas de temps (facultatif)
before step {{Code }}
// Code a exécuter après le pas de temps (facultatif)
after step {{Code }}

// Réaction au signal kill(facultatif)
kill {{Code }}
// Réaction au signal suspend(facultatif)
suspend {{Code }}
// Réaction au signal resume(facultatif)
resume {{Code }}

// Etat de départ de l'automate
initial state {{état initial de l'automate }}

// Liste des états
state Identifiant
{
entry {{Code d'entrée dans l'état }}
during {{Code en cours d'état }}
exit {{Code de sortie de l'état }}
kill {{Code correspondant au signal kill }}
suspend {{Code correspondant au signal suspend }}
resume {{Code correspondant au signal resume }}
}

```
    // Liste des transitions
    transition Identifiant
    {
        origin {{état d'origine de la transition }}
        extremity {{état d'extrémité de la transition }}
        preference {{préférence associée à la transition }}
        condition {{Expression booléenne correspondant à la condition de transition }}
        action {{Code C++ correspondant à l'action associée à la transition }}
    }

}
```

Chapitre 3

Moteur d'exécution

Par la suite, toute les classes qui vont être décrites sont déclarées dans le namespace **HPTS**, ce namespace ne sera pas rappelé dans les déclarations pour des raisons de lisibilité.

3.1 Contrôleur

L'exécution des automates est gérée par l'intermédiaire d'un contrôleur. Il existe deux types de contrôleurs, un contrôleur gérant l'ordonnancement des automates par rapport aux ressources, priorités et degrés de préférence associés aux automates et un contrôleur ne faisant pas d'ordonnancement. Le second contrôleur est plus rapide que le premier. Les deux classes de contrôleurs sont les suivantes :

- **StateMachineController**. Il s'agit du contrôleur se chargeant de l'exécution des automates en gérant l'ordonnancement.
- **StateMachineControllerNoResource**. Il s'agit d'un contrôleur gérant l'exécution des automates sans gestion de ressources et ordonnancement.

Les automates gérés par les contrôleurs doivent hériter de la classe **StateMachine**.

3.1.1 Calcul d'un pas de simulation

Le système simule l'exécution en parallèle de plusieurs automates. Pour ce faire, le contrôleur est basé sur la notion de pas de temps de simulation. En un pas de temps, une seule transition peut être effectuée au niveau des automates ainsi qu'une seule exécution des instructions qui leur sont associées. Pour gérer la cohérence au cours du calcul d'un pas de temps de simulation, ce calcul s'effectue dans un ordre bien particulier, en utilisant des barrières de synchronisation. A chaque pas de temps, l'ordre d'exécution est le suivant :

1. Gestion des signaux, envoi aux différents automates et changement d'état d'exécution.
2. Exécution du code à exécuter en début de pas de temps, pour tous les automates.
3. Collecte des transitions franchissables des automates.
4. Calcul d'ordonnancement des automates (uniquement pour le contrôleur StateMachineController).
5. Exécution du code de sortie des états pour tous les automates passant une transition.
6. Exécution de l'action associée à la transition pour tous les automates passant un transition.
7. Exécution du code d'entrée dans l'état pour tous les automates passant une transition.
8. Exécution du code à exécuter en restant dans l'état pour tous les automates.
9. Exécution du code à exécuter en fin de pas de temps, pour tous les automates.

Ce pas de temps est calculé par l'intermédiaire de la méthode :
StateMachineController::**compute**()

3.1.2 Lancement d'un automate et identifiant

Pour pouvoir lancer l'exécution d'un nouvel automate, le contrôleur dispose de la méthode suivante :

StateMachineController::Identifier StateMachineController::**start**(StateMachine * sm, bool autoDelete=true)

- *sm* est un pointeur sur l'automate à exécuter. Cet automate sera alors associé au contrôleur.
- *StateMachineController::Identifier* est une classe d'identifiant permettant de manipuler l'automate au travers du contrôleur. Elle est notamment utilisée pour l'envoi de signaux, la vérification d'état d'un automate et pour le mécanisme de destruction automatique des automates.
- *autoDelete* est un valeur booléenne permettant de stipuler si l'on veut que le contrôleur gère automatiquement la destruction (delete) d'un automate lorsque son exécution est terminée et qu'il n'est plus référencé.

Le mécanisme de gestion de la destruction automatique des automates est géré par l'intermédiaire des instances de la classe **StateMachineController::Identifier**. Lorsque plus aucune instance de cette classe référençant un automate donné n'existe et que cet automate est terminé, il est automatiquement détruit par le contrôleur.

La classe **StateMachineController::Identifier** dispose de deux méthodes permettant de récupérer le numéro d'identifiant d'un automate, commençant à zéro et évoluant en fonction du nombre d'automates lancés :

int StateMachineController::Identifer::**operator()** () const

ainsi qu'une méthode permettant de récupérer un pointeur sur l'automate associé à cet identifiant :

StateMachine * StateMachineController::Identifier::**value()** const

Outre la récupération de l'identifiant associé à un automate via la méthode **start**, une autre méthode, permettant de récupérer un identifiant à partir d'un pointeur sur l'automate est disponible dans le contrôleur :

StateMachineController::Identifier StateMachineController::**identifier**(StateMachine * sm)

3.1.3 Etat d'exécution des automates

L'état d'exécution des automates est traduit par le type énuméré **StateMachineControler::Status** dont les valeurs sont les suivantes :

- **running** : l'automate est actuellement en cours d'exécution.
- **suspended** : l'automate est actuellement suspendu dans son exécution. Il n'est donc plus utilisé mais en attente de reprise d'exécution.
- **killed** : l'automate a été tué. Il n'est donc plus exécuté.
- **ended** : l'automate s'est terminé en arrivant dans un état de terminaison.Il n'est donc plus exécuté.

L'état d'exécution d'un automate peut être consulté via une méthode de **StateMachineController** :

StateMachineController::Status StateMachineController::**status**(StateMachineController::Identifier id)

Cette consultation se fait donc en utilisant l'identifiant associé à l'automate qui est renvoyé par la méthode **start** ou la méthode **identifier** de la classe StateMachineController (Cf. 3.1.2).

3.1.4 Gestion des signaux

Le contrôleur sait gérer trois types de signaux associés à la gestion de l'exécution d'un automate :

- **kill** : signal servant à tuer l'automate. Ce dernier stoppe son exécution qui ne sera jamais reprise. Dans ce cas, l'automate passe dans l'état **killed**.
- **suspend** : signal de suspension d'exécution. L'automate stoppe son exécution temporairement en attendant d'être redémarré via un signal resume. Dans ce cas, l'automate passe dans l'état **suspended**.
- **resume** : signal de reprise d'exécution d'un automate ayant préalablement reçu un signal suspend. Dans ce cas, l'automate passe de l'état **suspended** à l'état **running**.

L'envoi de ces signaux est géré par l'intermédiaire de trois méthodes de la classe **StateMachineController** :

void StateMachineController::**kill**(StateMachineController::Identifier id)
void StateMachineController::**suspend**(StateMachineController::Identifier id)
void StateMachineController::**resume**(StateMachineController::Identifier id)

Cette gestion des signaux se fait en utilisant l'identifiant associé à l'automate qui est renvoyé par la méthode **start** ou la méthode **identifier** de la classe StateMachineController (Cf. 3.1.2). Il est à noter que ces signaux peuvent être envoyés depuis n'importe quel code associé à l'exécution des automates ; cependant leur prise en compte n'est réelle qu'en début de pas de calcul du contrôleur (Cf. 3.1.1) et ce pour un souci de cohérence.

3.1.5 Manipulation des ressources

Les ressources utilisées dans la description des automates pour gérer de l'exclusion mutuelle sont allouées par le contrôleur. Chaque contrôleur gère un ensemble de ressources. Pour un souci de simplicité et lisibilité, une ressource est identifiée par une chaîne de caractères puis manipulée via la classe **Resource**. Le contrôleur fournit des instances de ressources par l'intermédiaire de la méthode suivante :

Resource * StateMachineController::**resource**(std::string const & resourceId)

La ressource pointée est allouée dynamiquement par le contrôleur; **sa destruction est à la charge du programmeur**. D'autre part, le contrôleur peut fournir des informations sur l'automate possédant une ressource à un pas de temps donné. Cette information est accessible par l'intermédiaire de la méthode suivante :

StateMachine * StateMachineController::**takenBy**(Resource * res)

Cette méthode retourne un pointeur sur l'automate possédant actuellement la ressource.

3.2 Classes de description des automates

Les automates gérés par HPTS++ et les contrôleurs fournis doivent tous hériter de la classe de base **StateMachine**. Cette classe possède une interface de base qui est la seule manipulée par les contrôleurs de HPTS++.

Outre la classe **StateMachine**, trois autres classes d'automates (héritant de StateMachine) sont fournies au programmeur. Ces classes permettent entre autre de gérer des automates hiérarchiques, des automates possédant un contrôleur local.

3.2.1 Classe de base

La classe de base de tous les automates est **StateMachine**, cette classe possède les méthodes suivantes :

- void StateMachine::**initStateMachine**()
  Cette méthode permet d'initialiser un automate en créant ses états et ses transitions. Tant que cette

méthode n'est pas appelée, l'automate n'est pas utilisable. Il est à noter que la méthode **start** de la classe **StateMachineController** fait automatiquement appel à cette méthode.

- virtual void StateMachine::**beforeStep**()
  Cette méthode correspond au code associé à l'automate, devant être exécutée avant chaque pas de temps (Cf. 2.1.3).
- virtual void StateMachine::**afterStep**()
  Cette méthode correspond au code associé à l'automate, devant être exécutée après chaque pas de temps (Cf. 2.1.3).
- virtual double StateMachine::**priority**()
  Il s'agit de la méthode permettant de récupérer la valeur de la priorité associée à l'automate (Cf. 2.1.4). La priorité par défaut des automates est fixée à 1.
- virtual State * StateMachine::**entry**()
  Il s'agit de la méthode permettant de récupérer l'état initial de l'automate.
- virtual void StateMachine::**kill**()
  Il s'agit de la méthode appelée lors de la réception d'un signal **kill** (Cf. 2.3).
- virtual void StateMachine::**suspend**()
  Il s'agit de la méthode appelée lors de la réception d'un signal **suspend** (Cf. 2.3).
- virtual void StateMachine::**resume**()
  Il s'agit de la méthode appelée lors de la réception d'un signal **resume** (Cf. 2.3).

3.2.2 Automates fournis

Avec HPTS++, trois automates spécifiques sont fournis (Cf. diagramme UML 3.1). Ces automates offrent des fonctionnalités simples telles que l'accès au contrôleur gérant l'exécution de l'automate, la gestion d'une hiérarchie d'automates ou bien la déclaration d'un contrôleur local permettant une concurrence locale de plusieurs automates.

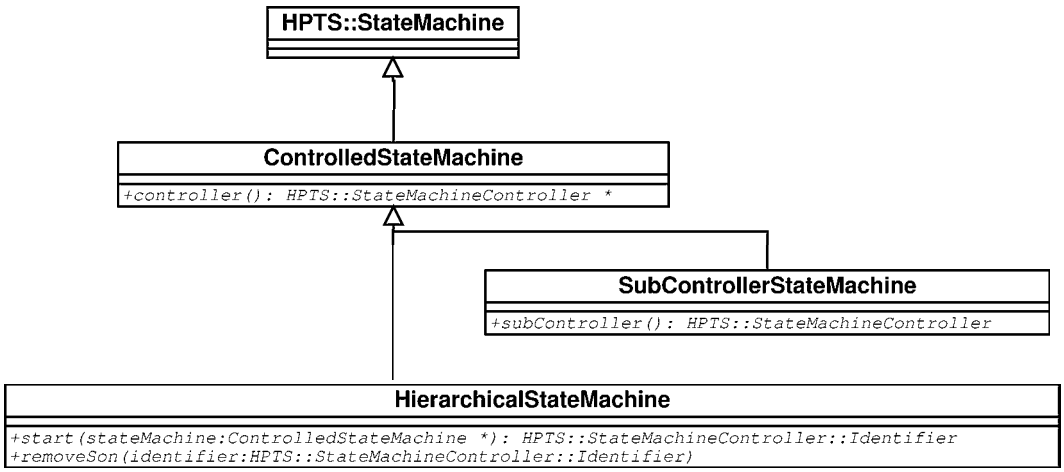

FIG. 3.1 – *La hiérarchie d'héritage des classes d'automate fournies.*

classe d'automate ControlledStateMachine

Cette classe d'automate peut être considérée comme la classe de base des automates fonctionnant sous HPTS++. Lors de sa construction, elle prend en paramètre le contrôleur gérant l'exécution de l'automate :

ControlledStateMachine::**ControlledStateMachine**(HPTS::StateMachineController * ctrl)

Par la suite, il est possible de manipuler le contrôleur par l'intermédiaire de la méthode suivante :

HPTS::StateMachineController & ControlledStateMachine::**controller**() const

classe d'automate HierarchicalStateMachine

Cette classe est conçue pour gérer une hiérarchie d'automates. Autrement dit, un automate héritant de **HierarchicalStateMachine** devient l'automate père d'une hiérarchie ; dans ce cas, tous les signaux envoyés à cet automate sont répercutés sur les automates fils et les automates fils sont exécutés avant l'automate père. Le constructeur de cet automate est le suivant :

HierarchicalStateMachine::**HierarchicalStateMachine**(HPTS::StateMachineController * ctrl)

Pour être inclus dans une hiérarchie d'automate, un automate fils doit être lancé par l'intermédiaire de la méthode suivante :

Controller::Identifier HierarchicalStateMachine::**start**(ControlledStateMachine * sm)

Cette classe d'automate offre aussi la possibilité de détacher un automate fils, c'est à dire que cet automate devient un automate parallèle et que les signaux envoyés à l'automate père ne sont plus répercutés sur cet automate fils. La méthode permettant de détacher l'automate fils par l'intermédiaire de son identifiant est la suivante :

HierarchicalStateMachine::removeSon(Controller::Identifier id)

Attention : la propagation des signaux aux automates fils est faite par l'intermédiaire des méthodes **kill**, **suspend** et **resume**. Donc, si ces méthodes sont ré-implémentées lors de la description d'un automate héritant de cette classe, il faut rappeler ces méthodes pour assurer la cohérence du système.

classe d'automate SubControllerStateMachine

Il s'agit d'une classe d'automate possédant un contrôleur local. Ce contrôleur permet de gérer un ensemble d'automates en concurrence sur des ressources définies dans le contrôleur local, cela permet de restreindre le nombre de ressources gérées par chaque contrôleur et donc de diminuer le coût de calcul de l'ordonnancement. Le constructeur de cette classe d'automate est le suivant :

SubControllerStateMachine::SubControllerStateMachine(Controller * controller)

Par la suite, le sous-contrôleur associé à cet automate peut être manipulé via la méthode suivante :

HPTS::StateMachineController & SubControllerStateMachine::**subController**()

Lors de l'exécution d'un automate héritant de cette classe, les automates associés au sous-contrôleur sont calculés et ordonnancés **avant** l'exécution de l'automate courant.

Attention : l'appel au sous-contrôleur pour le calcul d'un pas de simulation a été implémenté via la méthode virtuelle **beforeStep**(), donc, si une redéfinition de cette méthode est faite lors de la description des automates, il faut ajouter l'appel à **SubControllerStateMachine::beforeStep**() dans le code pour assurer l'exécution du sous-contrôleur et de ses automates associés.

Annexe A

Exemple de d'automate

Voici un exemple de code d'automate. Ce dernier utilise deux ressources, représentées par les variables res1 et res2. Les mots-clef du langage sont écrits en gras, le reste du code est en C++ standard. Le texte associé aux commandes *cout* de c++ décrit .

```
header
{{
    // Inclusion de fichiers a la mode C++      #include <iostream.h>
    #include <HierarchicalStateMachine.h>
    #include <Resource.h>
}}

state machine Test : state machine HierarchicalStateMachine
{
    variables
    {{
        // Donnees membres
        int cpt;
        HPTS::Resource * res;
        HPTS::Resource * res2;

        // Constructeur
        Test(Controller * sController, string const & res, string const & res2)
        : HierarchicalStateMachine(sController), cpt(0)
        {
            res = controller().resource(res);
            res2 = controller().resource(res2);
        }

        // Destructeur
        virtual  Test()
        { delete res; delete res2; }

    // Ici: declaration de toutes les methodes de l'automate.
    }}

    before step
    {{
        cout«"Before step "«endl;
    }}
```

```
after step
{{
    cout«"After step"«endl;
}}

// Definition de l'etat initial
// Note: l'utilisation d'un etat de l'automate se fait par nomEtat()
initial state {{ begin() }}

state begin
{
    entry {{ cout«"Father::begin::entry"«endl; }}
    during {{ cout«"Father::begin::during"«endl; }}
    exit {{ cout«"Father::begin::exit"«endl; }}
}

state inside
{
    // Liste des ressources utilisees
    // Fonctionne aussi avec list<Ressource*>
    uses {{ res }}, {{res2}};
    entry {{ cout«"Father::inside::entry"«endl; }}
    during {{ cout«"Father::inside::during"«endl; cpt++; }}
    exit {{ cout«"Father::inside::exit"«endl; }}
}

state tmp
{
    uses {{ res2 }};
    entry {{ cout«"Father::tmp::entry"«endl; }}
    during {{ cout«"Father::tmp::during"«endl; }}
    exit {{ cout«"Father::tmp::exit"«endl; }}
}

// Note: un etat est considere comme final s'il n'a pas de transition
// extrante
state end
{
    entry {{ cout«"Father::end::entry"«endl; }}
    during {{ cout«"Father::end::during"«endl; }}
    exit {{ cout«"Father::end::exit"«endl; }}
}

transition begin_inside
{
    origin {{ begin() }}
    extremity {{ inside() }}
    condition {{ true }}
    action {{ cout«"Father::begin_inside::action"«endl; }}
}

transition inside_end
```

```
    {
        origin {{ inside() }}
        extremity {{ end() }}
        condition {{ cpt>6 }}
        action {{ cout«"Father::inside_end::action"«endl; }}
    }

    transition inside_tmp
    {
        origin {{ inside() }} // Renvoie un etat
        extremity {{ tmp() }} // Renvoie un etat
        preference {{ -0.5 }} // Renvoie un double
        condition {{ true }} // Renvoie un bool
        action {{ cout«"Father::inside_tmp::action"«endl; }}
    }

    transition tmp_inside
    {
        origin {{ tmp() }}
        extremity {{ inside() }}
        preference {{ 0.5 }}
        condition {{ true }}
        action {{ cout«"Father::tmp_inside::action"«endl; }}
    }
}
```

Annexe B

Grammaire de HPTS++

stateMachineDescription	{ **fileHeader** } **stateMachineHeader** "{" { **variables** } { **priority** } { "before" "step" **cpp** } { "after" "step" **cpp** } { **signalKill** } { **signalSuspend** } { **signalResume** } **initialState** (**state**)* (**transition**)* "}";
fileHeader	"header" **cpp**;
stateMachineHeader	**stateMachineD** { ":" (**stateMachineD** | **classD**) ("," (**stateMachineD** | **classD**))* }
stateMachineD	"state" "machine" **ID**;
classD	"class" **ID**;
priority	"priority" **cpp**;
variables	"variables" **cpp**;
initialState	"initial" "state" **cpp**;
state	"state" **ID** "{" **usesResources** **entryState** **duringState** **exitState** **signalKill** **signalSuspend** **signalResume** "}";
usesResources	"uses" **cpp** ("," **cpp**)* ";";
entryState	"entry" **cpp**;
duringState	"during" **cpp**;
exitState	"exit" **cpp**;
transition	"transition" **ID** "{" "origin" **cpp** "extremity" **cpp** "preference" **cpp** **conditionTransition** **actionTransition** "}";

conditionTransition "condition" **cpp**;
actionTransition "action" **cpp**;
signalStart "start" **cpp**;
signalKill "kill" **cpp**;
signalResume "resume" **cpp**;
signalSuspend "suspend" **cpp**;
cpp “{{“ (**TOKENCPP** | **INTERCPP** | **STRINGVAL**)* “}}”;
listId **ID** ("," **ID**)*;