Extension de la plateforme de modélisation/simulation DIESE :
le composant OPEN DIESE dédié au développement de modules de simulation en position de serveur

Jean-Pierre RELLIER - INRA/MIA UBIA Toulouse
Août 2009

1) Présentation générale de OPEN DIESE

DIESE est une bibliothèque de classes C++, orientée vers le développement de simulateurs.

- BASIC DIESE est son composant qui fournit un ensemble de classes génériques pour modéliser un système dans une approche "objet", et simuler son fonctionnement dynamique. CONTROL DIESE est son composant orienté vers la modélisation du pilotage des systèmes. Les classes qu'il propose sont toutes des spécialisations des classes fournies par le composant BASIC DIESE.
Un simulateur développé sur les couches BASIC DIESE et CONTROL DIESE est un programme exécutable autonome, lancé par une commande directement adressée par l'utilisateur au système d'exploitation. Il gère ses entrées/sorties par des instructions que le programmeur a inséré dans le code interne du simulateur. L'arrêt du programme est géré de la même manière. L'arrêt "normal" est en général dicté par la donnée (en position d'entrée) de la date de fin de la période de simulation. Le programme peut s'arrêter avant cette date dans un contexte spécifié par le code du simulateur (par exemple, dans une circonstance particulière, il est impossible de continuer l'exécution du plan de pilotage du système).

- OPEN DIESE est un composant complémentaire qui permet de développer, non pas un simulateur autonome, mais un module de simulation exploité par un autre logiciel qui requiert en entrée des informations temporisées sur l'état d'un système. C'est typiquement le cas lorsque le logiciel est lui aussi un simulateur d'un volet de la dynamique temporelle d'un système, et qu'il souhaite déléguer, de manière synchronisée, un autre volet au module de simulation développé sous BASIC DIESE ou CONTROL DIESE. De manière générale, le module de simulation DIESE en position de "serveur" d'information vis-à-vis du logiciel, qui est, lui, en position de "maître". Cette position de "maître" n'exclut pas de fournir des données (d'entrée ou de contrôle) au module de simulation sous DIESE.

Les classes que propose la couche OPEN DIESE sont toutes des spécialisations des classes fournies par les composants BASIC DIESE et CONTROL DIESE.

L'exécution d'un simulateur développé sur la couche OPEN DIESE est donc commandée, non pas directement par l'utilisateur du simulateur, mais indirectement *via* le programme maître. C'est le programme maître qui est géré par l'utilisateur. Il en résulte que la période simulée d'une part, et les données d'entrée d'autre part sont spécifiées et notifiées au simulateur par le programme maître. De même, c'est au programme maître (et non plus à l'utilisateur) que le simulateur passe ses données en sortie.
Le programme maître est nécessairement un exécutable généré par la compilation d'un programme écrit en C++. Lors de l'étape d'édition de liens, on prend soin de faire connaître à cet exécutable toutes les références présentes dans le simulateur développé sous DIESE, et assemblées dans une bibliothèque logicielle.

2) Impact de l'ajout du composant OPEN DIESE

Le fonctionnement décrit ci-dessus passe par une modification très localisée du moteur de simulation de BASIC DIESE : la méthode Run() de la classe Simulation.

- BASIC DIESE et CONTROL DIESE
Pour l'essentiel, la méthode Run() s'applique à un agenda d'événements pré-rempli par l'utilisateur, et enchaîne sans discontinuer autant de dépilements de l'événement de tête qu'il en faut pour vider totalement d'agenda. C'est l'utilisateur qui a invoqué le Run() (très généralement dans la fonction 'main()')

- OPEN DIESE
La méthode RunOpenSimulation() (qu'on continue à abréger en Run() dans la suite) s'applique à un agenda d'événements pré-rempli par le programme maître, et enchaîne sans discontinuer des dépilements de l'événement de tête tant que celui-ci possède une date d'occurrence inférieure à un seuil fixé par le programme maître. C'est le programme maître qui a invoqué le Run(), en lui passant en paramètres, entre autres choses, la date seuil.
Le programme maître peut enchaîner autant d'invocations de la méthode Run() qu'il lui est utile. C'est typiquement le cas lorsque le programme maître délègue la simulation de la dynamique d'état du système, parce qu'il ne sait pas le faire lui-même, et parce qu'il n'est intéressé que par un certain nombre de "photographies" (généralement partielles) de l'état du système prises à des moments particuliers. Alors, le programme maître arrête la simulation à chacun de ces moments, exploite la photographie, puis relance la simulation jusqu'au prochain moment d'intérêt.

3) La relation « client-serveur » : généralités

D'une manière ou d'une autre, le programme maître doit assurer ou déléguer les tâches suivantes :

- L'initialisation de la simulation
  - Il s'agit essentiellement d'instancier la classe OpenSimulation, cette spécialisation de Simulation dotée d'une méthode Run() adaptée. La génération par Solfege du code du simulateur a produit la fonction InitOpenSimulation, qui réalise cette instanciation (c'est la variable globale pCurrentSim) ainsi que les tâches annexes d'installation de la hiérarchie des classes et de réalisation des moniteurs (cf. documentation BASIC DIESE). Le programme maître doit donc invoquer cette fonction avant le premier appel à Run().
  - En complément, certaines propriétes de la simulation doivent être établies, telles notamment la valeur du pas de l'horloge, et la date calendaire réelle correspondant à la valeur zéro de l'horloge. Sous BASIC DIESE et CONTROL DIESE, ces choses-là sont couramment spécifiées dans le 'fichier des paramètres de la simulation' placé en entrée de la simulation. Dans ce cas, il faut appeler la fonction qui interprète ce fichier : ParseSimulationFile(), argumentée par le chemin complet du fichier en question.
  - Comme le 'fichier des paramètres de la simulation' peut exploiter des paramètres définis dans le 'fichier des paramètres du système' (cf. documentation BASIC DIESE), il convient d'invoquer préalablement la fonction dédiée ParseParameterFile().

- L'initialisation de l'état du système

Cette initialisation est réalisée (comme sous BASIC DIESE et CONTROL DIESE) de deux manières complémentaires : (i) l'interprétation des fichiers placés en entrée de la simulation (et notamment le 'fichier de la structure du système'), et (ii) les méthodes d'initialisation des processus lancés dès le début de la période simulée. La sous-tâche (i) est réalisée avant l'invocation de Run(). La sous-tâche (ii) est réalisée lors des dépilements des événements datés en t=0. Noter que certaines parties de l'état du système peuvent être initialisées plus tard, si leur dynamique de démarre pas en t=0.

- Pour la sous-tâche (i), le programme maître invoque la fonction dédiée ParseStructureFile(), après l'initialisation de la simulation et avant le premier appel à Run().
- Pour la sous-tâche (ii), le programme maître instancie les processus à initialiser, puis les encapsule dans un événenent, munis d'une directive INIT. Après avoir donné une date d'occurrence (typiquement t=0) et une priorité à l'événement, il l'insère dans l'agenda de pCurrentSim. Pour déclencher ces événements, et ainsi appliquer les directives d'initialisation, il faut invoquer la méthode Run(), en lui passant en argument la valeur de l'horloge à la date d'initialisation la plus tardive, augmentée de 1 (typiquement la valeur 1 si toutes les initialisations se font en t=0).

L'initialisation de l'état du système peut être jointe à l'initialisation de la simulation dans la fonction InitOpenSimulation. C'est cette option qui est prise par Solfege, qui enchaîne donc ces deux tâches dans une version standard, laquelle peut encore être éditée, pour modification ou augmentation. Le programme maître n'a donc plus qu'à invoquer la fonction InitOpenSimulation, avant le passer à la tâche ci-dessous.

- La gestion de la simulation de la dynamique du système

Elle consiste, pour le programme maître, à provoquer une pause dans la série des dépilements d'événements par Run() à la première prochaine date à laquelle il souhaite demander des informations au simulateur sur l'état du système. Cet instant de pause est spécifié par une valeur passée en argument de la méthode Run().
Ainsi, la gestion de la simulation consiste en l'itération de la séquence suivante :
- détermination du prochain instant de pause t
- invocation de Run(..., t, ...)
- l'inspection et exploitation de l'état du système
Il est important de noter que l'instant de la pause doit être connu dès l'appel à Run(), c'est-à-dire que le programme maître ne peut pas le découvrir pendant la série de dépilements engagée par cet appel. Cela parce que le programme maître a "perdu la main" dès l'appel et qu'il ne la reprendra qu'au "retour" de la méthode Run()
Le Run() ne doit être invoqué que si la simulation n'est pas terminée. Lorsque cela survient, c'est le Run(), l'ayant lui-même détecté, qui en informe le programme maître.

- L'échange d'information avec le simulateur

L'option prise dans la conception de OPEN DIESE est que le programme client sait à l'avance quelles parties du système il souhaite "regarder" au prochain instant de pause, et que cet ensemble de choses est dénombrable. Une autre option aurait été de laisser le programme client naviguer librement, lors de la pause, dans l'ensemble du système simulé à la recherche des variables d'état ciblées.

- L'ensemble des parties du système que le programme client va regarder est représenté par une liste de pointeurs sur des instances de la classe InterfaceDataItem. Cette liste doit être créée par le programme client, puis passée en argument de la méthode Run(), au côté de l'instant de pause. Chacune de ces instances indique (i) l'entité qui sera regardée, (ii) le descripteur qui sera regardé pour cette entité, et (iii) l'instant auquel le simulateur doit capturer la valeur de ce descripteur. Couramment, cet instant sera justement l'instant de pause, mais il peut lui être antérieur, c'est-à-dire compris entre la valeur de l'horloge de simulation lors de l'appel de Run et l'instant de pause. Il ne reste plus à la méthode Run qu'à compléter cette structure avec la valeur qu'aura prise ce descripteur à la date demandée. C'est cette valeur que récupèrera le programme client au "retour" du Run().
- Le programme maître peut aussi influer sur la dynamique du système simulé, et établissant ou en modifiant la valeur de variables d'état ou de décision. Ce passage d'information vers le simulateur utilise aussi une liste de pointeurs sur des instances de la classe InterfaceDataItem. Cette liste doit être créée par le programme client, puis passée en argument de la méthode Run(),. Chacune de ces instances indique (i) l'entité qui sera touchée, (ii) le descripteur qui sera valué pour cette entité, et (iii) l'instant auquel le simulateur doit attribuer la valeur au descripteur. Couramment, cet instant sera justement l'instant de l'appel du Run, mais il peut lui être postérieur, c'est-à-dire compris entre cet appel et l'instant de la prochaine pause. Il ne reste plus à la méthode Run qu'à effectuer les affectations de valeurs.

4) La relation « client-serveur » : aspects pratiques

D'un point de vue pratique, le programmeur du code "maître" doit provoquer la séquence suivante d'opérations, de manière adaptée à son contexte logiciel, son application, ... et à son style de codage.

➢ L'initialisation de la simulation

```
ParseParameterFile("/home/durand/.../sim1.par");

pCurrentSim = new OpenSimulation();
//---------------------------------------------------------------------------
pCurrentSim->ClockTimeUnit(DAY);          // ou bien :
pCurrentSim->ClockTimeUnitQuantity(1);    // ParseSimulationFile(
pCurrentSim->SetBeginningDate(1, 9, 2009); //     "/home/durand/.../sim1.sim");
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
ContinuousProcess pCP = new ProcessusA();      // en complement ou
int processInitClock = 0;                      // en remplacement de :
                                               // ParseStructureFile("/home/durand/.../sim1.str");
pCP->PostInitialisationEvent(processInitClock);  ...
//---------------------------------------------------------------------------
```

Solfege génère une version standard de cette initialisation : c'est la fonction InitOpenSimulation, argumentée avec les trois fichiers évoqués ('paramètres', 'simulation', 'structure'). On note au passage qu'on ignore les deux autres fichiers ('spécifications de sortie' et 'directives') qu'on place en entrée d'un simulateur OPEN DIESE ou CONTROL DIESE. En effet, dans un simulateur OPEN DIESE, les spécifications de sorties sont remplacées par les instances d'InterfaceDataItem, et le fichier des directives est remplacé par le contrôle opéré par le programme maître.
Cette fonction est partiellement modifiable par le développeur pour y ajouter, notamment, des initialisations du système par processus.

Il reste, le cas échéant, à provoquer lés dépilements d'événements qui gèrent les processus d'initialisation, par :

```
//---------------------------------------------------------------------------
int iterationCounter = 0;
int pauseClock = processInitClock +1; // hypothese : un unique ProcessusA
StructTmToStringISO(ClockValueToTm(pauseClock));
((OpenSimulation*)pCurrentSim)->RunOpenSimulationToDate(iterationCounter, gTimeString);
//---------------------------------------------------------------------------
```

➢ La gestion de la simulation et la communication avec le simulateur :

//

```
// Definition de l'interface d'entree
//
pEntityTab* IN_tab = new pEntityTab();
Entity* pIN_1 = new InterfaceDataItem();
pIN_1->SetIntConstValue(TYPE_ATTRIBUTE, FLOAT);
pIN_1->SetStringConstValue(ENTITY_NAME_ATTRIBUTE, "parcelle_1");
pIN_1->SetIntConstValue(DESCRIPTOR_ID_ATTRIBUTE, QUANTITE_EAU);
IN_tab->push_back(pIN_1);
//
// Definition de l'interface de sortie
//
pEntityTab* OUT_tab = new pEntityTab();
Entity* pOUT_1 = new InterfaceDataItem();
pOUT_1->SetIntConstValue(TYPE_ATTRIBUTE, FLOAT);
pOUT_1->SetStringConstValue(ENTITY_NAME_ATTRIBUTE, "culture_1");
pOUT_1->SetIntConstValue(DESCRIPTOR_ID_ATTRIBUTE, QUANTITE_MS);
OUT_tab->push_back(pOUT_1);
//
// Le contrôle de la simulation
//
Boolean simulationIsOver = FALSE;
for(int j=0;j<100 && !simulationIsOver;j=j+10) {
  //
  // valeurs datées transmises au simulateur   (par défaut : date de l'appel)
  //
  float quantiteEau_j = <...>; // à la discrétion du programme maître
  pIN_1->SetInterfaceValue(quantiteEau_j); // valeur transmise au simulateur
  //
  // dates pour les valeurs reçues             (par défaut : date de retour)
  //
  // date prochaine pause
  //
  char* nextPause = <...>; // date ISO 8601 à la discrétion du programme maître

  simulationIsOver = pCurrentSim->RunOpenSimulationToDate(iterationCounter,
                                          IN_tab,
                                          nextPause,
                                          OUT_tab);
  float quantiteMS_j
     = pOUT_1->GetFloatInterfaceValue();  // valeur émise par le simulateur
  // utilisation de quantiteMS_j à la discrétion du programme maître
```

A un endroit ou à un autre, ce type de code doit figurer, globalement dans cet ordre, dans le programme maître, lequel peut accéder à toutes les références DIESE (fonctions, méthodes, classes, ...) dans la bibliothèque attachée lors de l'édition de liens.

- Cas particulier : un simulateur OPEN DIESE autonome

Un simulateur développé avec OPEN DIESE peut être exploité de manière autonome, c'est-à-dire hors du contrôle d'un programme maître. C'est notamment le cas en phase de développement et, conjointement, de test de ses propres fonctionnalités de simulation.

Le code doit alors avoir la forme, non seulement d'une bibliothèque, mais aussi d'un exécutable. On sait que son point d'entrée est alors la fonction main, laquelle doit donc être écrite.

Solfege génère une version standard de cette fonction, identique à celle générée sous BASIC DIESE et CONTROL DIESE. Elle peut être modifiée par le développeur, pour des besoins spécifiques, mais ce n'est généralement pas nécessaire. Cette version invoque, non pas la fonction InitOpenSimulation, mais la fonction InitSimulation. Elle enchaîne la lecture et l'interprétation des cinq fichiers de la liste standard en entrée.