Conceptos y Biblioteca EOSimulator

Mg. Ariel Gonzalez
Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto
agonzalez(AT)dc.exa.unrc.edu.ar

Temas de la Clase: Continuación de *Conceptos de Simulación y Uso de la Biblioteca EOSimulator*

Entidades y Recursos

- Entidades:
  - Temporales
  - Permanentes

- Recursos: “restricciones”
  - Renovables
  - No renovables

Eventos

- eventos ligado o planificado (bound event).

- evento condicional (conditional event).

- Ejemplo de un hospital simple.

Diagramas de Actividad

- Describe la vida de las actividades en el sistema.
  - Colas: círculos grandes
  - Actividades: rectángulos
  - Recursos: círculos pequeños
  - Flujo de las entidades y comunicación con los recursos: Líneas con flechas.

- Ejemplo de un hospital simple.

Métodos de Estructuración de una Simulación

- Método de tres fases: *(avanzar reloj, ejecutar eventos planificados, ejecutar eventos condicionados "true")*

- *Eventos ligados y condicionados se implementan por separado.*

- Método de dos fases: *(avanzar reloj, ejecutar eventos planificados)*

- *Los eventos condicionados incluidos en los planificados.*

Pseudocódigo

Una vez realizado el diagrama de actividades, el próximo paso en el diseño del programa de la simulación, es escribir un pseudocódigo de sus eventos.

Usaremos algo similar a lo utilizado en análisis de sistemas, con una sintaxis similar a Pascal.

Pseudocódigo

Usaremos una notación similar al inglés estructurado descrito por De Marco [1] y usado en análisis de sistemas. Es un subconjunto del lenguaje Inglés escrito en formato rígido sin calificativos. El lenguaje es menos preciso y más transparente que un programa en Pascal pero es fácil de traducir a cualquier lenguaje de programación estructurado.

Dado que el texto de Davies y O'Keefe usa Pascal, el pseudocódigo que presenta está escrito e indentado en forma similar a dicho lenguaje de programación, usando palabras reservadas como ser `begin`, `end`, `while`. Las oraciones que no son propias de Pascal se delimitan con llaves ($\{\}$) y paréntesis angulados ($<>$).

[1] DeMarco, T., Structured Analysis and System Specification, Prentice Hall, 1979.

Pseudocódigo

Las Figuras 2.4 y 2.5 de Davies y O'Keefe muestran un pseudocódigo del sistema del hospital simple usando el método de las tres fases y el método de eventos. Los apéndices 2.A y 2.B muestran pseudocódigos para los otros casos de estudio.

La Tabla 2.1 muestra una instancia particular del hospital simple (con 4 camas disponibles), conteniendo el día de arribo y el largo de la estadía de 13 pacientes. La Tabla 2.2 muestra los estados da la cola de admisión y de las camas del hospital, para los datos de la Tabla 2.1.

Resumen

Un modelo de simulación avanza desde un punto discreto del eje de tiempo (*golpe de reloj*) al siguiente.

SED maneja *actividades* en las que están implicadas entidades individuales. Estas *entidades* se encuentran en distintos estados a medida que la simulación se desarrolla (ocupadas, ociosas o en cola).

Generalmente las entidades necesitan de *recursos* para poder realizar alguna actividad. La disponibilidad de un recurso puede ser una condición para que suceda un evento, el que marca el comienzo de una actividad (*evento condicionado*).

Resumen

Un *evento fijo* marca el fin de una actividad y es agendado a ejecutarse en una determinada hora. Un *alimentador* (o feeder), es un evento fijo que genera entidades temporarias.

Un *diagrama de actividades* muestra el flujo o el ciclo de vida de las entidades. Mediante un *pseudocódigo* se describen las actividades en lenguaje estructurado.

Existen tres formas (*vistas del mundo*) de estructurar y controlar el flujo del tiempo en SED: a) tres fases, b) dos fases, c) a procesos; a) y b) son enfoques basados en eventos y se diferencian por el manejo de los eventos condicionados. El método de las tres fases considera a los eventos condicionados como procedimientos separados. En el método de las dos fases, se integran en los eventos fijos.

Preguntas

1. Identificar las similaridades y diferencias entre los pseudocódigos de las Figuras 2.4 (tres fases) y 2.5 (dos fases) para el mismo sistema del hospital.

2. Ejecutar manualmente las acciones de los eventos de las Figuras 2.4 y 2.5, con los datos de la Tabla 2.1 (para los primeros días de la simulación).

Introducción

Utilizamos EOSimulator (basado en C++), desarrollado por el Departamento de Investigación Operativa del Instituto de Computación.

Alternativamente se podrá utilizar Pascal_SIM (basado en Pascal y utilizado en el texto de Davies y O'Keefe), desarrollado por la Universidad de Southampton.

Ambas bibliotecas permiten la programación de modelos basados en los enfoques orientados a eventos de dos o tres fases.

Conceptos importantes

- Modelo
- Evento (fijo y condicionado)
- Entidad
- Recurso
- Calendario
- Ejecutivo

EOSimulator

- Biblioteca para ser utilizada desde un programa en C++.
- Conjunto de clases que brindan servicios y requieren la definición de operaciones prefijadas.
- Código fuente disponible (no se recomienda modificarlo).

Implementación en EOSimulator

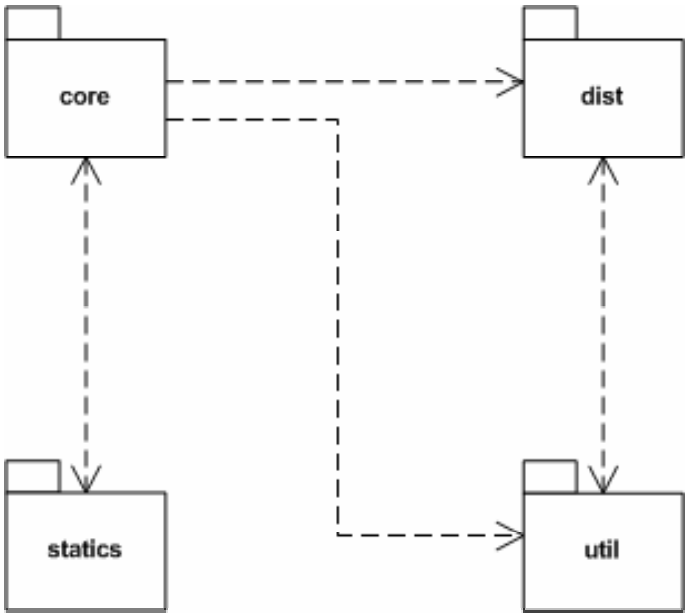
- *Entidades*: Se pueden utilizar las brindadas por defecto, o se pueden crear nuevas subclases.

- *Recursos*: Brindados por la biblioteca.

- *Eventos*: Son clases abstractas, con operaciones cuyo método debe definirse en clases derivadas.

- *Modelo*: Deriva de una clase abstracta y debe contener todos los atributos relevantes del sistema que se modela.

Arquitectura de EOSimulator



- `core`: Estructura de la simulación.
- `dist`: Números aleatorios y distribuciones.
- `statics`: Recolección de datos.
- `utils`: Utilidades varias.

Entidades

Poseen atributos para indicar su próximo evento fijo y su instante de ocurrencia. Deben crearse de forma *dinámica* (utilizando `new`). Pueden crearse entidades específicas, derivadas de `Entity`.

```
class Entity {
private:
    BEvent* bEv;
    double clock;
public:
    Entity ();
    virtual ~Entity ();
    void setBEvent (BEvent* bEv_);
    void setClock (double clock_);
    double getClock ();
    void processEvent ();
};
```

Recursos

Clase base abstracta `Bin`; clases concretas `Renewable` y `NonRenewable`.

```
class Bin {
public:
    virtual ~Bin();
    void acquire (double amount_);
    bool isAvailable (double amount_);
};
```

Recursos

```
class Renewable: public Bin {
public:
    Renewable (double quantity_, double max_);
    ~Renewable ();
    void returnBin (double amount_);
};

class NonRenewable: public Bin {
public:
    NonRenewable (double quantity_);
    ~NonRenewable ();
    void addBin (double amount_);
};
```

Colas

Interfaz `EntityQueue`, con implementaciones de colas fifo, lifo y de prioridad (dada por un comparador).

```
class EntityQueue {
public:
    EntityQueue() {};
    virtual ~EntityQueue() {};
    virtual void push(core::Entity* ent_) = 0;
    virtual core::Entity* pop() = 0;
    virtual void remove(unsigned int i_) = 0;
    virtual bool empty() = 0;
    virtual core::Entity* operator[] (unsigned int i_) = 0;
    virtual unsigned int size() = 0;
};
```

Eventos

- Clases abstractas `BEvent` y `CEvent`.
- Un evento pertenece a un modelo y tiene una referencia al mismo (`owner`).
- Para eventos concretos, debe implementarse la operación `eventRoutine`.

Eventos fijos

```
class BEvent {
protected:
     Model& owner;
public:
     BEvent (std::string name_, Model& owner_);
     virtual ~BEvent ();
     std::string getName();
     virtual void eventRoutine (Entity* who_) = 0;
};
```

El nombre de los `BEvent` es el identificador utilizado para agendar las entidades. Es de tipo `std::string`.

Eventos condicionados

```
class CEvent {
protected:
     Model& owner;
public:
     CEvent (Model& owner_);
     virtual ~CEvent ();
     virtual void eventRoutine () = 0;
};
```

Modelo

Es una clase abstracta. Define el modelo del sistema que se quiere simular. Posee como atributos: eventos (B y C), recursos, entidades globales, colas, histogramas y distribuciones. En un modelo concreto se deben implementar las operaciones de inicialización `init` y `doInitialSchedules`.

```
class Model {
private:
    Experiment* exp;
    utils::BEventMap bEvs;
public:
    Model();
    virtual ~Model();
    virtual void init () = 0;
    virtual void doInitialSchedules () = 0;
    void connectToExp (Experiment *exp_);
    void registerBEvent (BEvent* bEv_);
    void registerCEvent (CEvent* cEv_);
    void registerDist (dist::Distribution* dist_);
    void registerHistogram (statics::Histogram* hist_);
    void schedule (double offset_, Entity* who_, std::string what_);
    double getSimTime();
};
```

Calendario

Dos tipos: eventos B (entidades agendadas a un evento fijo) y C (eventos condicionados registrados).

```
class BCalendar {
private:
     double simTime, endSim;
     utils::EntityQueueOrdered ents;
public:
     BCalendar ();
     ~BCalendar ();
     void bPhase ();
     bool isStopped();
     void schedule (double offset_, Entity* who_);
     void setEndTime (double when_);
     double getSimTime();
};
```

Calendario

```
class CCalendar {
private:
     utils::CEventVector cEvs;
public:
     CCalendar ();
     ~CCalendar ();
     void cPhase ();
     void registerCEvent (CEvent* cEv_);
};
```

Ejecutivo

- Clase `Experiment` contiene los calendarios B y C.
- Las fases A y B usan `BCalendar`; la fase C usa `CCalendar`.
- Para correr la simulación, se conecta la instancia de `Model` con la instancia de `Experiment` y luego se ejecuta mediante la operación `run`.

Ejecutivo

```
class Experiment {
private:
    bool running;
    dist::DistManager distMan;
    BCalendar bCal;
    CCalendar cCal;
    Model* currModel;
public:
    Experiment ();
    ~Experiment();
    void run (double simTime_);
    void setModel (Model* model_);
    void schedule (double offset_, Entity* who_);
    void setSeed (unsigned long seed_);
    void registerDist (dist::Distribution* dist_);
    void registerCEvent (CEvent* cEv_);
    double getSimTime();
};
```