

SPADES
-
Appunti sparsi

- work in progress -

Realizzato da:
Marco Pracucci

1. L'ALGORITMO DI SIMULAZIONE

L'algoritmo del motore di simulazione di SPADES (simulation engine) è basato su un generico algoritmo per simulatori ad eventi discreti. Questo algoritmo è stato modificato dall' autore di SPADES (Patrick Riley) principalmente per garantire buone prestazioni di calcolo parallelo, durante le fasi di thinking degli agenti.

1.1 Gli agenti

SPADES vede un agente come un'entità computazionale che riceve dei messaggi *sense*, effettua una serie di calcoli (*think*) e restituisce delle azioni (*act*). Un agente deve effettuare i suoi calcoli solamente quando riceve una *sense*, ovvero si trova nella fase di thinking; questo vincolo, tuttavia, viene rilassato dai *time notify*. Più precisamente, siccome SPADES vincola un agente ad inviare azioni solamente in risposta ad una *sense*, il sistema fornisce all'agente una speciale azione chiamata *request time notify*. Un *time notify* è essenzialmente una sensazione vuota alla quale l'agente può rispondere con delle azioni, come avviene con ogni altro tipo di *sense*. Pertanto, un *time notify* può far iniziare una fase di thinking.

1.2 L' approccio con lista di eventi centralizzata

Astrattamente, quello che SPADES deve fare è gestire un insieme di eventi che compaiono in diversi tempi. Per esempio, un evento può essere una *sense* (SenseEvent) oppure una *action* (ActEvent).

Esistono diversi approcci alla gestione delle liste di eventi, che si distinguono principalmente per dinamicità e complessità degli algoritmi. Al fine di semplificare l'implementazione, SPADES utilizza un approccio di tipo centralizzato. In questo approccio, vi è una singola lista di eventi gestita da un processo *master* (nel nostro caso, il simulation engine), il quale è responsabile di schedulare e gestire gli eventi per tutti gli altri processori (nel nostro caso, gli agenti). Ogni processo che vuole schedulare un evento futuro deve notificarlo al processo master per far sì che questo venga schedolato. Il processo master ha, quindi, una conoscenza completa (in ogni momento) di tutti gli *eventi in attesa* (pending events) ed è in grado di decidere indipendentemente quali tra questi eventi processare, senza che vengano violati vincoli di causalità.

1.2.1 Svantaggi

Uno svantaggio dell'approccio con lista di eventi centralizzata è che ogni processo deve notificare allo scheduler centrale che ha finito di processare un evento ed è pronto per processarne altri. Tuttavia, gli agenti sono stati progettati secondo il paradigma sense-think-act che mitiga questo svantaggio, in quanto tutti gli agenti producono un' azione in risposta agli eventi di *sense*. Queste azioni permettono anche di notificare allo scheduler che il calcolo del precedente evento è stato completato.

Un ulteriore svantaggio di questo approccio è dato dall' efficienza e scalabilità, in quanto vi è solo un singolo processo che coordina le attività per tutti gli agenti. Questo unico punto di coordinamento potrebbe essere il collo di bottiglia dell'intera simulazione. Tuttavia, si evince dal manuale di SPADES che questo sistema è stato progettato per simulazioni con un numero di agenti relativamente basso e che su sistemi di tali dimensioni non si notano colli di bottiglia causati dallo scheduler degli eventi.

1.3 Il ciclo sense-think-act

In questo paragrafo viene fornita una spiegazione degli eventi che accadono durante un normale ciclo sense-think-act di un agente. La figura 1.1 mostra la natura di questo ciclo.

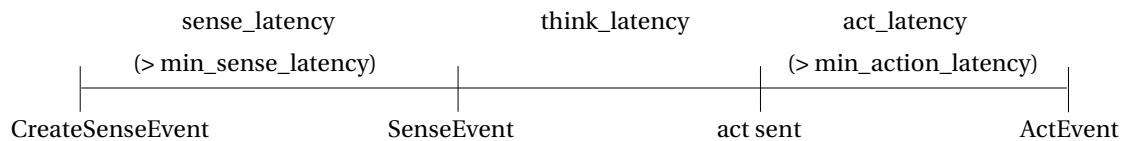


Figura 1.1, eventi nel ciclo sense-think-act di un agente

Il simulation engine vede e gestisce il ciclo sense-think-act nel seguente modo:

- Come prima cosa viene inserito nella coda degli eventi (`pending_event_queue`) un evento del tipo `CreateSenseEvent`, il quale contiene le informazioni necessarie alla creazione di un evento sensazione (`SenseEvent`). Di solito, per realizzare un `CreateSenseEvent`, viene letto lo stato corrente del mondo e convertito in un insieme di informazioni da inviare all'agente. SPADES richiede che il tempo di simulazione intercorso tra `CreateSenseEvent` e `SenseEvent` sia maggiore di `min_sense_latency`, un valore specificato dal progettista della simulazione.
- Le informazioni contenute in un `CreateSenseEvent` vengono successivamente incapsulate in un `SenseEvent`, che verrà a sua volta inserito nella coda degli eventi. Quando questo `SenseEvent` verrà *realizzato*, l'insieme di informazioni saranno inviate all'agente che inizierà così la fase di thinking. È importante notare che la realizzazione di un `SenseEvent` NON richiede la lettura di nessuno stato del mondo (corrente o passato) in quanto le informazioni sono fissate al tempo di realizzazione del `CreateSenseEvent`.
- Quando l'agente riceve la sense, il communication server che gestisce l'agente stesso inizia a *cronometrare* la computazione dell'agente. Quando l'agente ha inviato tutte le azioni, computate nella sua fase di thinking, al communication server, quest'ultimo converte il tempo che l'agente ha speso nella fase di thinking in tempo della simulazione (`think_latency`). Le azioni ed il `think_latency` vengono inviate, dal communication server, al simulation engine (operazione raffigurata in figura 1.1 da *act sent*).
- Ricevute le azioni ed il `think_latency`, il simulation engine aggiunge l'`action_latency` (determinato interrogando il world model) ed inserisce un `ActEvent` nella coda degli eventi. SPADES richiede che il tempo di simulazione intercorso tra `act sent` e `ActEvent` sia maggiore di `min_action_latency`.
- La realizzazione dell'`ActEvent` è ciò che attualmente genera gli effetti delle azioni dell'agente sul mondo.

1.4 Lookahead property

Un concetto associato riguardo i *conservative parallel discrete event simulator* è la necessità, che questi hanno, di una proprietà lookahead non-zero per ottenere buone prestazioni di calcolo parallelo. In pratica, il valore lookahead è un vincolo di limite inferiore sul tempo di simulazione che intercorre tra la generazione di un evento sul processore A e la realizzazione dello stesso sul processore B. Elevati valori di lookahead danno origine a buone prestazioni.

In SPADES, la proprietà lookahead è stata implementata attraverso le latenze minime: **min_sense_latency** e **min_action_latency**. Più precisamente, le latenze su sensazioni ed azioni forniscono un valore lookahead per gli agenti e permettono questi di calcolare le azioni (fase di thinking) in parallelo. Per esempio, quando viene realizzato un SenseEvent per l' agente *i*, quest'ultimo non può generare nessuno evento con tempo di comparsa minore del tempo corrente (come lui lo percepisce) più **min_action_latency**.

1.5 La percezione del tempo da parte di un agente

Un agente non conosce il tempo corrente della simulazione come lo conosce il simulation engine. Più precisamente, un agente non viene notificato ogni volta che il world model avanza il tempo della simulazione, pertanto l' agente non è in grado di conoscere (in ogni istante) il tempo corrente. Tuttavia, per come è stato progettato SPADES, questa caratteristica non rappresenta un problema. Infatti, un agente percepisce il tempo corrente sulla base dell'ultima comunicazione con il simulation engine: una sensazione ricevuta oppure una azione inviata.

Indichiamo con **agent_i.currenttime** il tempo corrente così come è percepito dall' agente *i*. Questo valore viene memorizzato dal simulation engine (per ogni agente), il quale lo utilizza per calcolare il *minimum agent time*.

1.6 Come processare solamente gli eventi safe

L'approccio *conservative* di SPADES fa sì che il motore di simulazione processi solamente gli eventi che possono essere definiti *safe*. Per esempio, vengono processati solo gli eventi che non possono violare vincoli di causalità.

Per ottenere questo, SPADES determina l' arco di tempo massimo nel quale gli eventi contenuti si possono considerare safe. Questo valore equivale al minor tempo nel quale uno degli *n* agenti può generare un evento che abbia effetto sugli altri agenti o sul mondo simulato (ad esempio, un' azione). Inoltre un agente si può trovare, in un dato momento, in uno dei seguenti due stati: waiting o thinking. Nel primo caso (waiting) significa che l'agente è in attesa di messaggi da parte del simulation engine, pertanto non può generare azioni. Diversamente, se si trova nello stato di thinking, significa che il simulation engine ha inviato una sense all' agente *i* e questi deve ancora rispondere con delle azioni.

Il motore di simulazione conosce **min_action_latency**, il tempo corrente di ogni agente *i* e lo stato in cui si trova l'agente *i*. Pertanto, il simulation engine è in grado di calcolare il valore di **min_agent_time** (minimum agent time), ovvero il minor tempo (prossimo) nel quale almeno un agente può generare un evento che abbia effetto sulla simulazione. La tabella 1.1 mostra lo pseudo-codice di questo semplice algoritmo.

```

calculateMinAgentTime()
  foreach i in set_of_all_agents
  if (agenti.status = Waiting)
    agent_timei = ∞
  else
    agent_timei = agenti.currenttue + min_action_latency
  return mini agent_timei

```

Tabella 1.1, pseudo-codice per il calcolo di min_agent_time

1.7 Gli eventi e FixedAgentEvent

Gli eventi sono uno dei principali oggetti di SPADES. La simulazione, infatti, procede avanzando il world model ad un particolare tempo e realizzando uno o più eventi con tempo di comparsa uguale al tempo corrente.

SPADES fornisce un insieme di classi dalle quali il progettista della simulazione può ereditare durante lo sviluppo delle proprie classi eventi. La figura 1.2 mostra la gerarchia delle classi eventi, le quali ereditano tutte dalla classe astratta Event.

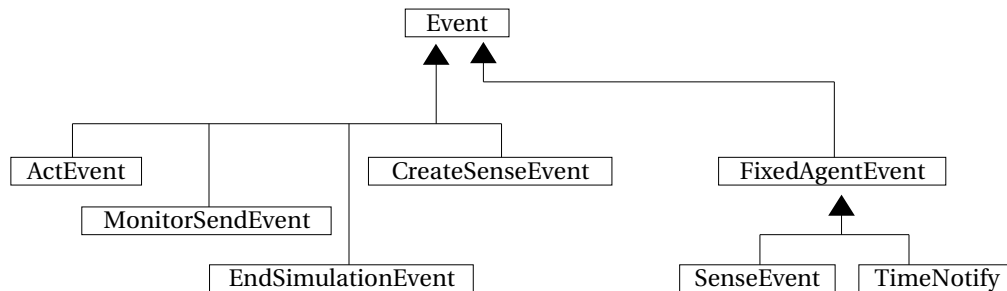


Figura 1.2, gerarchia delle classi eventi

FixedAgentEvent è una sottoclasse speciale di eventi. Questa categoria di eventi ha proprietà particolari ed è stata sviluppata per migliorare le performance di calcolo parallelo in SPADES (una descrizione dettagliata dei motivi la si può trovare nel paragrafo 6.8 del manuale di SPADES). Più precisamente, dato un insieme di FixedAgentEvent destinati a diversi agenti, questi eventi possono venire realizzati dal simulation engine fuori ordine di tempo (anche se l'ordine temporale viene mantenuto per ogni singolo agente). I FixedAgentEvent godono delle seguenti proprietà:

1. Non dipendono dallo stato corrente del mondo.

Verificare questa proprietà per TimeNotify è banale: questo tipo di messaggio non contiene informazioni riguardo lo stato corrente del mondo. Per quanto riguarda SenseEvent, occorre notare che le informazioni sullo stato del mondo vengono prelevate durante la generazione di CreateSenseEvent; la realizzazione di SenseEvent non porta ad accedere allo stato corrente del mondo.

2. Hanno effetti solamente su di un singolo agente

3. SenseEvent e TimeNotify sono gli unici due eventi del tipo FixedAgentEvent

4. I FixedAgentEvent sono gli unici eventi che possono far iniziare all'agente la fase di thinking. È importante notare che non necessariamente un agente entra nella fase di thinking ogni qual volta riceve questo tipo di messaggi.

Nonostante i FixedAgentEvent possano venire realizzati fuori ordine temporale, le garanzie di correttezza, che SPADES fornisce, sono:

1. Tutti gli eventi che NON sono FixedAgentEvent vengono realizzati in ordine di tempo
2. Tutti gli eventi che inviano sensazioni agli agenti sono FixedAgentEvent
3. L'insieme di eventi del tipo FixedAgentEvent per un particolare agente vengono realizzati in ordine di tempo

1.8 Minimum sensation time

Il valore **min_sense_time** (minimum sensation time) è il minor tempo prossimo nel quale una nuova sensazione (diversamente da un TimeNotify) può essere generata ed accodata. Siccome l'implementazione corrente di SPADES richiede che il world model (a partire dal progettista) fornisca il min_sense_latency, il min_sense_time è dato dal tempo corrente della simulazione sommato a min_sense_latency.

1.9 TimeNotify

I TimeNotify sono eventi privilegiati e non sono vincolati dal min_sense_time. Questo perché i TimeNotify non hanno effetto su nessun agente a parte quello che lo ha richiesto.

TODO: la descrizione dei TimeNotify non è particolarmente rilevante ai fini di questa breve analisi sul simulation engine. Tuttavia, potrebbe essere interessante analizzare a fondo questo tipo di evento e studiarne (eventuali) interessanti applicazioni.

1.10 Il motore di simulazione

I concetti fondamentali alla comprensione dell'algoritmo su cui si basa il motore di simulazione di SPADES sono stati illustrati nei paragrafi precedenti. In questo paragrafo vengono illustrati gli ultimi concetti importanti al fine della comprensione ed infine l'algoritmo.

Come introdotto nel paragrafo 1.7, il simengine deve realizzare i FixedAgentEvent di un particolare agente in ordine di tempo, mentre globalmente (tra tutti gli agenti) questi eventi possono essere realizzati anche fuori ordine di tempo. Per assicurare che questi vincoli vengano rispettati, il simengine mantiene una coda di FixedAgentEvent per ogni agente. Un FixedAgentEvent viene inserito nella coda del rispettivo agente solo se il tempo dell'evento è minore del min_sense_time. In tal caso, i FixedAgentEvent vengono accodati prima di essere inviati all'agente.

Ci sono principalmente due funzioni che gestiscono queste code. La prima, **enqueueAgentEvent()** inserisce un FixedAgentEvent nella coda del rispettivo agente. La seconda, **doneThinking()** viene chiamata ogni volta che l'agente finisce la fase di thinking. Entrambe le funzioni fanno uso di una terza, **checkForReadyEvents()**. Lo pseudo-codice di tutte e tre le funzioni è mostrato in tabella 1.2.

```

enqueueAgentEvent(e:Event)
  a = e.agent
  agenta.pending_agent_events.insert(e)
  checkForReadyEvents(a)

```

```

doneThinking(a:Agent, t:Time)
  agenta.currenttime = t
  checkForReadyEvents(a)

```

```

checkForReadyEvents(a:Agent)
  while(true)
    if (agenta.status = thinking)
      return
    if (agenta.pending_agent_events.empty())
      return
    next_event = agenta.pending_agent_events.pop()
    realizeEvent(next_event)

```

Tabella 1.2, pseudo-codice delle funzioni enqueueAgentEvent(), doneThinking() e checkForReadyEvents()

La tabella 1.3 mostra lo pseudo-codice dell' algoritmo del motore di simulazione di SPADES. Più precisamente, la tabella mostra la struttura del metodo *mainLoop()* della classe SimEngine, che implementa il motore di simulazione.

```

repeat forever
  receive messages
  next_event = pending_event_queue.head
  min_agent_time = calculateMinAgentTime()
  while (next_event.time < min_agent_time)
    advanceWorldTime (next_event.time)
    pending_event_queue.remove (next_event)
    if (next_event is a FixedAgentEvent)
      enqueueAgentEvent (next_event)
    else
      realizeEvent (next_event)
  next_event = pending_event_queue.head
  min_agent_time = calculateMinAgentTime()
  min_sense_time = current_time + min_sense_latency
  foreach e (pending_event_queue) /* in time order */
    if (e.time > min_sense_time)
      break
    if (e is a FixedAgentEvent)
      pending_event_queue.remove(e)
      enqueueAgentEvent(e)

```

Tabella 1.3, pseudo-codice del mainLoop()

APPENDICE A

A.1 La porta di ingresso del simengine

Il “punto di ingresso” del simulation engine è la funzione *SimulationEngineMain()* definita nel file *enginemain.cpp*. I parametri da passare a questa funzione sono: un riferimento agli argomenti passati a linea di comando ed un puntatore ad un' istanza del WorldModel. La tabella A.1 mostra le principali operazioni eseguite da *SimulationEngineMain()*.

<pre>int SimulationEngineMain (const int argc, const char *const *argv, WorldModel pWorldModel)</pre> <ul style="list-style-type: none">- Parsing degli argomenti a linea di comando e dei parametri nei file di configurazione- Inizializzazione del logger- Creazione di una istanza della classe SimEngine (pSE), passandole come argomento un puntatore al WorldModel (pWorldModel)- Esecuzione di pSE->mainLoop()

Tabella A.1, operazioni eseguite dalla funzione *SimulationEngineMain()*

A.2 Codice sorgente commentato del metodo SimEngine::mainLoop()

```
bool
spades::SimEngine::mainLoop ()
{
    /* This is a back pointer so that AgentInfo can realize events */
    AgentInfo::setSimEngine(this);

    if (inShutdownMode())
    {
        errorlog << "I ran into an error on initialization!" << ende;
        return false;
    }

    TimeVal tv_pause(EngineParam::instance()->getPauseModeWaitSec(),
                    EngineParam::instance()->getPauseModeWaitUsec());
    TimeVal tv_normal(EngineParam::instance()->getWaitSec(),
                    EngineParam::instance()->getWaitUsec());

    tv_limited_rate_next = tv_normal;

    // Inizializza il WorldModel
    if (!pWorldModel->initialize (this))
    {
        errorlog << "WorldModel failed to initialize!" << ende;
        return false;
    }

    // Inizializza il MonitorManager; questa operazione occorre eseguirla dopo
    // l'inizializzazione del WorldModel in quanto devono essere disponibili
    // le informazioni del monitor header
    if (!monitor_manager.initialize())
    {
        errorlog << "Error in monitor manager initialize!" << ende;
        changeSimulationMode(SM_Shutdown);
    }

    if (EngineParam::instance()->getUseTextEventLog())
    // Manteniamo un file testuale degli eventi processati (utile per fare debug)
    {
        StringReplacer strrep;
        strrep.addReplacement("%D", SharedParam::instance()->getLogfileDir());
        std::string fn(strrep.performReplacements(EngineParam::instance()->
```

```

        getTextEventLogFn()));
pctxt_event_log = new ofstream( fn.c_str() );
if (!(*pctxt_event_log))
{
    warninglog(10) << "Error opening text event log: "
        << EngineParam::instance()->getTextEventLogFn()
        << ende;
    delete pctxt_event_log;
    pctxt_event_log = NULL;
}
}

// Esegue le ultime inizializzazioni del commengine (dopo che tutte le altre
// inizializzazioni sono state effettuate)
if (!commengine.initFinal())
{
    warninglog(10) << "CommEngine failed to do its final initialization, exiting"
        << ende;
    changeSimulationMode(SM_Shutdown);
}

tv_curr.setFromTimeOfDay();
tv_pause_mode = tv_curr;
tv_last_event_time = tv_curr;

if (!inShutdownMode())
    cout << "Initialization completed..." << endl;

while (!inShutdownMode())
{
    checkForRebaseLimitedRate();

    // Riceve i messaggi; tv_this_wait e' il timeout sulla select, chiamata
    // in checkReceivedData()
    int nummess;
    TimeVal tv_this_wait;
    if (inPauseMode())
        tv_this_wait = tv_pause;
    else if (sim_mode == SM_RunNormal)
        tv_this_wait = tv_normal;
    else if (sim_mode == SM_RunLimitedRate)
        tv_this_wait = tv_limited_rate_next;
    else
        errorlog << "I don't know what to do with the wait time for this mode "
            << sim_mode << ende;
    nummess = commengine.checkReceivedData(tv_this_wait);

    // Processa i messaggi ricevuti da commengine
    nummess = processMessages();

    // Imposta il timeval corrente, utilizzato da diverse funzioni nel seguito
    tv_curr.setFromTimeOfDay();

    printStatusUpdate(false);

    // Controlla se ci sono degli agenti da migrare. Questa operazione viene
    // effettuata solo ora, in quanto tutti i messaggi ricevuti dai CS
    // devono essere stati processati
    agent_alloc_manager.checkForMigrations();

    if (inPauseMode())
        // Simulazione in pausa
        {
            actlog (50, ELC->eventman())
                << "In pause mode, calling back to world model" << ende;

            commengine.resetNumRecentlyQueuedMessages();
        }
}

```

```

pWorldModel->pauseModeCallback ();
actlog(130, ELC->eventman())
  << "pauseModeCallback caused "
  << commengine.getNumRecentlyQueuedMessages()
  << " to be queued" << ende;

// Controlla se e' stato superato il limite massimo (in sec) sul
// tempo che la simulazione puo restare in pausa
if (EngineParam::instance()->getMaxPauseModeSeconds() > 0 &&
(tv_curr - tv_pause_mode).toSeconds() >
EngineParam::instance()->getMaxPauseModeSeconds())
{
  warninglog(10) << "Too long in pause mode ("
    << (tv_curr - tv_pause_mode)
    << ">" << EngineParam::instance()->getMaxPauseModeSeconds()
    << ") with no messages."
    << " cs: " << commengine.getNumCommServers()
    << " a: " << getNumAgents()
    << ende;

  // Shutdown della simulazione
  initiateShutdown();
}
}
else
// Simulazione NON in pausa
{
  #ifdef TEST_SEGFAULT
    if (sim_time >= 500)
    {
      strcpy(NULL, "I'm a little seg fault, short and stout");
      errorlog << "I tried to seg fault and failed!" << ende;
    }
  #endif

  // Calcola il min_agent_time, il simulation time piu' "vicino" nel quale
  // un agente puo causare un evento che ha effetto sugli altri agenti
  // o sul WorldModel
  // Come (giustamente) Patrick fa notare: we should really not have
  // to recalculate each time, but update as messages and events are processed
  SimTime min_agent_time = calculateMinAgentTime ();
  if (min_agent_time != SIMTIME_INVALID && min_agent_time < sim_time)
    errorlog << "How is the min_agent_time less than sim_time? "
      << min_agent_time << " " << sim_time << ende;

  int events_processed = 0;

  actlog (10, ELC->eventman())
    << "Now processing events from queue"
    << " events: " << pending_events.size()
    << " agenttime: " << min_agent_time << ende;

  // reset this now in case we run out of events or something like that
  tv_limited_rate_next = tv_normal;

  SimTime head_event_time;
  while (!inShutdownMode() &&
    !pending_events.empty() &&
    ( head_event_time = (*pending_events.begin())->getTime ()
      < min_agent_time || min_agent_time == SIMTIME_INVALID))
  // Cicla per ogni evento nella coda pending_events che ha tempo di
  // comparsa < min_agent_time
  {
    actlog (20, ELC->eventman())
      << "Examining event for time "
      << head_event_time
      << ", agenttime: " << min_agent_time
      // Insure was reporting that this was giving a READ DANGLING error
      // sometimes. I can't figure out how that could occur, but we don't
      // really need the event to be logged here

```

```

// << ": " << (**pending_events.begin())
<< ende;

if (sim_time > head_event_time)
    errorlog << "I pulled out a pending event with time less than "
    << "current time! " << sim_time << " > " << head_event_time << ende;

// Controlla se dobbiamo avanzare lo stato della simulazione,
// ovvero se il tempo corrente della simulazione (sim_time) <
// del tempo dell'evento (head_event_time)
if (sim_time < head_event_time)
{
    // Chiede al WorldModel di avanzare il tempo della simulazione
    // al tempo head_event_time
    SimTime new_sim_time =
        pWorldModel->simToTime(sim_time, head_event_time);

    actlog(60, ELC->eventman())
        << "Advancing world state from " << sim_time
        << "; Wanted " << head_event_time << " got " << new_sim_time
        << ende;

    if (new_sim_time > head_event_time)
        errorlog << "The world went past the head event time!" << ende;

    if (new_sim_time < head_event_time)
    {
        /* We will allow the world model to not advance to the head time
        as long as some progress is being made */
        if (!pending_events.empty ())
        {
            SimTime new_head_event_time =
                (*pending_events.begin ())->getTime();

            if (new_head_event_time > sim_time && new_sim_time == sim_time)
                warninglog(10) << "Time did not advance and no new event "
                << "in the queue!" << ende;

        }

        sim_time = new_sim_time;

        /* The sim_time did not catch up to this event time, so we have
        to go back to the pending events queue */
        continue;
    }
    else
    {
        sim_time = new_sim_time;
    }
}

// we let the simtime advance whether or now we are rate limited.
// Since that can be an expensive computation, we will also check the
// time of day again now
if (sim_mode == SM_RunLimitedRate)
{
    // Limitiamo il "rate" solo per gli eventi che NON sono
    // FixedAgentEvent
    if (!(**pending_events.begin())->isFixedAgentEvent())
    {
        tv_curr.setFromTimeOfDay();
        TimeVal tv_target = limitedRateTimeFor(head_event_time);
        if (tv_target > tv_curr)
        {
            actlog(50, ELC->eventman())
                << "Limited rate: Time has not arrived for event at time "
                << head_event_time
                << "\ttarget=" << tv_target << ", curr=" << tv_curr
                << ende;
        }
    }
}

```

```

        tv_limited_rate_next = tv_target - tv_curr;
        break; // <=====
    }
}

// Rimuoviamo l'evento dalla coda dei pending_events
Event *e = *pending_events.begin ();
pending_events.erase (pending_events.begin ());

// Realizziamo l'evento. Il metodo realizeEvent(e) invoca
// e->realizeEvent(this)
if (!realizeEvent(e))
    errorlog << "Failed to realize event: " << *e << ende;

// Aggiorniamo il contatore degli eventi realizzati
events_processed++;

// Aggiorna il min_agent_time
/* We should really not have to go through the whole calculation each time,
but when there's not too many agents, this should be fine
I'll make this better later */
min_agent_time = calculateMinAgentTime ();
if (min_agent_time != SIMTIME_INVALID && min_agent_time < sim_time)
    errorlog << "How is the min_agent_time less than sim_time? "
        << min_agent_time << " " << sim_time << ende;
} // end while

// Estrae ogni evento agente dalla coda pending_events con tempo <
// calculateMinSenseTime() e lo sposta nella coda agente. Restituisce
// il numero di eventi sposati
int cnt = scanForAgentEvents(calculateMinSenseTime());
if (cnt)
    actlog(60, ELC->eventman()) << "scanForAgentEvents moved "
        << cnt << " events" << ende;

if (events_processed == 0)
// Non sono stati processati eventi
{
    if ((tv_curr - tv_last_event_time).toSeconds() >
        EngineParam::instance()->getTimeoutForEvent() &&
        EngineParam::instance()->getTimeoutForEvent() > 0)
// Scaduto timeout (in sec) sull'attesa di eventi
    {
        warninglog(10)
            << "Timeout while waiting for next event to be processed "
            << (tv_curr - tv_last_event_time).toSeconds() << " > "
            << EngineParam::instance()->getTimeoutForEvent()
            << ende;

        // Shutdown della simulazione
        initiateShutdown();

        errorTrace();
    }
}

#ifdef NO_ACTION_LOG
/* Log some extra useful information */
for ( AgentList::iterator iter = lAgent.begin ();
    iter != lAgent.end ();
    ++iter)
    {
        actlog(100, ELC->agentcontrol())
            << "Agent " << iter->first
            << " status: " << iter->second.getStatus()
            << " time: " << iter->second.getCurrTime()
            << " aqueue: " << iter->second.sizeOfAgentEventQueue()
            << ende;
    }
}

```

```

        #endif
    }
    else
    // Sono stati processati degli eventi
    {
        actlog(50, ELC->eventman()) << "Processed " << events_processed
            << " events" << ende;

        total_events_processed += events_processed;
        tv_last_event_time = tv_curr;
    }
} // end simulazione non in pausa
} // end while (!inShutdownMode())

// --- SIMULAZIONE TERMINATA ---

printStatsUpdate(true);

// Stampa il tempo usato nella simulazione
if (timing)
    stopTiming();
cout << "Real time used for simulation: "
    << elapsed_real_time.toSeconds() << endl;
actlog(10, ELC->eventman()) << "Real time used for simulation: "
    << elapsed_real_time.toSeconds() << ende;
double mean_st_sec = static_cast<int>(getSimulationTime() /
    elapsed_real_time.toSeconds());
cout << "Overall mean simtime/sec: " << mean_st_sec << endl;
actlog(10, ELC->eventman()) << "Overall mean simtime/sec: "
    << mean_st_sec << ende;

// Finalizza il WorldModel
if (!pWorldModel->finalize ())
{
    errorlog << "WorldModel failed to finalize!" << ende;
    return false;
}

return true;
}

```

ANNOTAZIONI

La breve analisi effettuata sul motore di simulazione di SPADES è basata sulla **versione 1.0 di SPADES** stesso e sulla versione 0.91 del manuale.

Equivalenze dei termini:

- Con il termine *tempo*, tranne dove esplicitamente citato, si intende il tempo della simulazione. Pertanto, i termini “tempo” e “tempo della simulazione” avranno lo stesso significato.
- I termini “simulation engine”, “simengine” e “motore della simulazione” hanno tutti lo stesso significato.

BIBLIOGRAFIA

1. www.sourceforge.net/projects/spades-sim

2. <http://bat710.univ-lyon1.fr/~cpham/ParSimul.html>

TODO

- Analizzare il **Limited Rate Run Mode**. Questo è un modo di esecuzione di SPADES nel quale viene fissato un limite superiore alla velocità della simulazione. Questa funzione può risultare utile per far corrispondere la velocità di simulazione con il tempo reale, per esempio per far interagire umani con la simulazione o eseguire operazioni di monitoring su di essa.