A Single Thread Discrete Event Simulation Toolkit for Java: STSimJ

Wenguang Chen, Dingxing Wang, and Weimin Zheng

Tsinghua University, Beijing 100084, China

Abstract. Discrete event simulation is widely used in simulating complex systems. SimJava [6]is a popular java toolkit for discrete event simulation. However, SimJava employs multiple threads for the simulation process. The disadvantage of the multi-threaded approach is that the result of the simulation is not repeatable, because of the uncertainty introduced by multi-threads. In this paper, we propose a single thread discrete event simulation toolkit for Java, whose result is always repeatable.

1 Introduction

Discrete event simulation is one of the key method to validate and evaluate computer and telecom systems.

SIMULA[1] is a language designed for simulation. Hase++[4,3] is a simulation library for C++ which provides discrete process based simulation similar to SIMULA's class and libraries. In order to enable web browser based simulation, Java is also used for simulation. SimJava[6] is a popular java toolkit for discrete event simulation.

However, both Hase++ and SimJava used multiple threads or processes for the simulation process. The disadvantage of the multi-threaded approach is that the result of the simulation is not repeatable, because of the uncertainty introduced by multi-threads.

There're also other single thread simulation package, such as SMPL[5]. But SMPL is written in C difficult to be extended.

In this paper, we propose a single thread discrete event simulation toolkit for Java(STSimJ), which would always produce the same output for the same input.

2 STSimJ Overview

2.1 STSimJ Concepts

In this section, we illustrate some concepts used in STSimJ for system modelling..

- Entity: The concept of entity in STSimJ is similar to the entity in SimJava. A system consists of several entities.

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3038, pp. 1131–1137, 2004.

- State: At any given time, each entity must be in a certain state. The entity may have several possible states, and may change from one state to the other because of incoming events or internal logic.
- Transition: An entity may change its state from one to another. This process is called "Transition".
- Event: Events are the main reasons for entities to change their states. Events can be fired by either the entity itself or other entities.

2.2 STSimJ Java Library Overview

In STSimJ, there are mainly 6 classes in its library:

- `STSEvent`: A class for holding a generic event. Programmers can inherit this class to define their own event classes.
- `STSEventList`: A helper class to maintain the event queue of each entity.
- `STSTransition`: A class to record the information related to state transition.
- `STSEntityState`: It contains a method named `eventHandler()`, which is supposed to be overridden by real state classes to describe the entities behavior on the state.
- `STSEntity`: It contains member variables and methods for a generic entity, which include state management, transition management etc. The `initialize()` method of class `STSEntity` is supposed to be overridden by real entity classes to describe the events, states and transition information of the entity.
- `STSSystem`: It controls all entities, maintains and advances the simulation time.

In the next section, we are going to describe the usage of these classes in detail.

3 How to Simulate a System with STSimJ

In this section, we describe how to simulate a system with STSimJ.

3.1 Modelling the System with State Chart

Let's demonstrate the process with a simple example:

A system contains a `sender` and a `receiver`. The sender would send a message to the `receiver`, then waits for the `receiver`'s response. After getting the response, it holds for 10 seconds and send a message to the `receiver` again. If it has sent 100 messages, it will exit.

The `receiver` is in idle state initially, after receiving a message from the `sender`, it holds itself for 1.234 seconds, then send a response message to the `sender`. If it received 100 messages, it will exit.

It's easy to draw the state chart[2] of the system. The Fig. 1 shows the state chart of the system. There are 2 entities in the system: `sender` and `receiver`. Each entity has 3 states: `idle`, `send` and `receive`. And there are 3 kinds of state transition illustrated in Fig. 1:

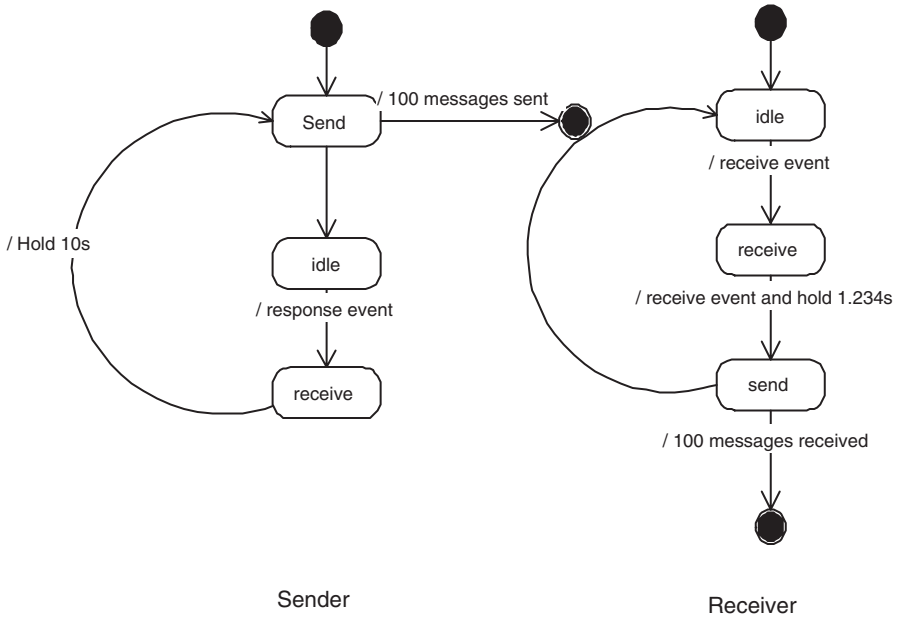

Fig. 1. State chart of the sender/receiver system

- From `SENDER_SEND` to `SENDER_IDLE`
  This is a immediate state transition, which does not involve any external events and time delay. The `sender` would enter SEND_IDLE states right after it sends out the message to the `receiver`. We call this kind of transition ITAA(Immediate Transition After Action).
- From `SENDER_IDLE` to `SENDER_RECEIVE`
  This is a blocked state transition. The transition would not happen until the `sender` received a message from the `receiver`. We call this kind of transition TBBE(Transition Blocked By Event).
- From `SENDER_RECEIVE` to `SENDER_SEND`
  This is a delayed state transition. The entity is held for the specified period and would not process any event. When the period passes, the entity would transit to the target state. We call this kind of transition DT(Delayed Transition).

3.2 Define the Behavior of Entities

Subclassing STSEntity to Define Entity Behavior
In order to define the behavior of a entity, we need to subclass the `STSEntity` and define states, events and transition by overriding its method `initialize()`. The definition of class `Sender` is as following:

```
public class Sender extends STSEntity{
  public Sender(String name) {
    super( name );
  }

  public void intialize() {
    //1. Define states
    STSEntityState senderIDLE =
                        new STSEntityState(this, "SENDER_IDLE");
    STSEntityState senderSEND =
                        new STSEntityState(this, "SENDER_SEND");
    STSEntityState senderRECEIVE =
                        new STSEntityState(this, "SENDER_RECEIVE");
    addState(senderIDLE);
    addState(senderSEND);
    addState(senderRECEIVE);

    //2. Add state transition table to entities
    STSEvent messageEvent =
                        new STSEvent(0.0, "MESSAGE", null, null);
    addStateTransition(senderIDLE, senderRECEIVE, messageEvent);
    addHoldStateTransition(senderRECEIVE, senderSEND, 10.0);

    //3. Set initial state
    setInitialState(senderSEND);
  }
}
```

The state definition is a simple step and does not need to be explained more. However, the definition of events and state transition is a little more tricky.

The class `STSEntity` have two methods to describe the state transition:

- addStateTransition()
  This method is used to specify the transition of TBBE type. In this sample, the statement `addStateTransition(senderIDLE,senderRECEIVE, messageEvent)` describes this transition:if the current state is `senderIDLE`, then it will wait for a MESSAGE event and change the entity state to `senderRECEIVE`.
- `addHoldStateTransition()`
  This method is used to specify the transition of DT type. In this sample, the statement `addHoldStateTransition(senderRECEIVE,senderSEND, 10.0)` describes this transition:if the current state is `senderRECEIVE`, then hold 10.0 seconds and enter the state `senderSEND`.

For the state transition type ITAA, we did not specify it here. Instead, it is specified in the `eventHanlder()` of state `SENDER_SEND`. See the next section.

Subclassing STSEntityState to Define State Event Handler
If the entity should perform some operations other than jump to another state or waiting for a event in a state, the operations should be specified in the simulation code. In this case, we should subclass `STSEntityState` and override its method `eventHandler()`. Besides, the transition of type ITAA should also be specified in the `eventHandler()` method.

```
public class StateSenderSend extends STSEntityState {
  int msgCnt = 0;
  public StateSenderSend(STSEntity ofEntity, String stateName) {
    super(ofEntity, stateName);
  }
  public void eventHandler( ISTSEvent event) {
    if ( msgCnt ++ >= 100 ) {
      entity.setCurrentState("EXIT", null);
      return;
    }
    STSEvent evt = new STSEvent(STSSystem.getGlobalTime(),
                          "MESSAGE", entity.getName(), "hello" );
    STSSystem.sendEvent( "receiver", 0.5, evt );
    entity.setCurrentState("SENDER_IDLE", null);
  }
}
```

In this example, the operation performed is to send a MESSAGE event to the entity `receiver` by using the method `STSSystem.sendEvent()` . And the entity `sender` enter the state `SENDER_IDLE` right after the `sendEvent()` by calling `STSEntity`'s method `setCurrentState()`.

3.3 Put It All Together to Simulate!

Now we have defined the entities and their behavior. It's time to get them together:

```
public class SampleSimulation {
  public static void main( String[] args ) {
    STSSystem.initialize();

    // Add sender entity into system
    STSEntity sender = new STSEntity( "sender" );
    STSEntity receiver = new STSEntity( "receiver" );
    STSSystem.add(sender);
    STSSystem.add(receiver);

    // Start to simulate
    STSSystem.run();
  }
}
```

Simply create and add them to the class `STSSystem`, it is ready to go. The `STSSystem.run()` is a static method of the class `STSSystem` that would start the simulation.

4 The Single Thread Simulation Kernel

The main feature of STSimJ is that it just use a single thread simulation kernel. All the situations that would require blocking, i.e. waiting for a event to arrive and holding the system for a period etc., are defined as state transition in STSimJ, and thus there's no need to block any simulation code in STSimJ. The simulation kernel is like following:

```
public class STSSsytem {
  ...
  public static void run() {
    ...
    // Simulate
    boolean hasEvents = true;
    globalTime = advanceToNextEventTime();
    while ( hasEvents && globalTime < terminateTime ) {
      hasEvents = false;
      for (int i = 0; i < entityList.size(); i++) {
        STSEntity entity = (STSEntity) entityList.elementAt(i);
        hasEvents |= entity.run();
      }
    }
  }
}
```

The method `advanceToNextEventTime()` find the next event time from the current event lists of all entities. `globalTime` is the global clock that can be read by all entities. The `STSEntity::run()` method would handle the event occurred at the time and/or transit states.

It should be noted that all `entity.run()`s are executed one by one and there's no chance for them to execute simultaneously.

5 Conclusion and Future Work

In this paper, we proposed a single thread discrete event simulation toolkit for Java:STSimJ. To use STSimJ, one can use the state chart to describe the system to be simulated, then map the state chart to STSimJ code. We have illustrated that with the frameworks and libraries provided by STSimJ, it is quite easy to construct the simulation code.

In STSimJ, there's no blocked thread in the simulation code. All simulation is done with synchronous call to entity method and event handler method. So

the whole simulation is completed within the single thread. Thus, it avoids the uncertainty of multi-thread simulation and is always repeatable.

In the future, we will integrate more features into STSimJ, which include statistics library and GUI libraries.

References

1. Birtwhistle, G.M., Dahl, O-J. Myhrhaug B. and Nygaard K.,Simula Begin, Academic Press, 1973
2. Fowler M. and Scott K., UML Distilled: A Brief Guide to the Standard Object Modeling Language(2^{nd}ed), Addison Wesley Longman, 2000
3. Howell, F.W. Hase++:a discrete event simulation library for C++. http://www.dcs.ed.ac.uk/home/fwh/hase++/hase++.html
4. Ibbett, R.N.,Heywood,P.E. and Howell,F.W., "Hase : a flexible toolset for computer architects." The Computer Journal, 2000
5. MacDougall M.H. Simulating computer systems:techniques and tools. The MIT Press, 1987
6. McNab, R. and Howell, F.W. Using Java for Discrete Event Simulation, Proceedings of Twelfth UK Computer and Telecommunications Performance Engineering Workshop (UKPEW),Edinburgh, 219-228,1996 38(10):755-764.