AN XML BASED SIMULATION LANGUAGE SUPPORTING MODEL KNOWLEDGE REPRESENTATION

CHEN LIU, WEI-PING WANG

System simulation Lab, School of Information System and Management,
National University of Defense Technology, Changsha 410073 , Hunan, China
liuchenchangsha@hotmail.com, luoyingliuchen@vip.sina.com

Abstract:

Knowledge representation of simulation models is the core element to influence simulation development. This paper summarizes the fundamental simulation model formalism from the system point of view. It includes basic model, composite model and scenario model. It also presents rigorous formal specification. The executable simulation language, which supports model knowledge concise representation, is designed based on simulation model formalism. The paper also discusses the corresponding relationship between ESimL and simulation model formalism and details the syntax and semantics of elements in ESimL. Based on simulation model formal specification, the abstract simulation algorithm is given and ESimL virtual machine, which is capable of automatically interpreting and executing simulation model represented by ESimL, is designed. Finally the paper presents a chaos system model, Lorenz system, specified by ESiml, the simulation results can show the validation of the theory and verification of ESimL.

Keywords:

Model knowledge representation; simulation model specification; executable simulation language; simulation virtual machine

1 Introduction

Simulation technology has been widely applied in various fields, such as artificial intelligence, battle simulation, embedded software development, weapon system effectiveness evaluation, product design, etc. Simulation technology has been used for designing, development, evaluation or exploration. This effectively saves the costs of product development, improves product quality and people gain deeper understanding of research systems, problems and routines.

Model is the core in simulation. Simulation is meant to control the model's execution, while models are the cognitive abstract of research systems or problems. Models should include system abstract information and simulation control information in order to be operated by simulator. Model knowledge representation is just to research on how to represent this system abstract information, how to represent participating simulation control information so as to make simulation model representation more concise and complete. Thus simulation model standardization and the ability of fast building system simulation model to conduct design, development and exploration can be achieved.

In the early stage of system simulation development, people use advanced programming language or special simulation language to conduct modeling and simulation. Staff will have to be fluent in complex program syntax, which significantly undermines efficiency. In large-scaled simulation project, modeling and simulation process consists of hundreds of complicated activities facing multi domains. It requires efforts and cooperation from different fields, different locations and different professional staff in order to develop smoothly. Simulation technology development has now moved from single-field oriented application to multi-field collaborative simulation development. The fundamental problem in collaborative simulation lies on models' representation standard, model integration and reusability [1] [2].

The purpose of this research paper is to solve the problem of simulation model knowledge representation and to make simulation model easy to be developed and reused. It uses the DEVS [3] [4] research results and applies system point of view to explore simulation model's basic formalism. The paper also combines XML data exchange standard to research an XML based language, ESimL, to support model knowledge representation. ESimL will declare a group of less quantity but relatively complete elements and elements properties to describe abstract

characteristics, structures, and behavior to support building system model. It can support modeler to maximum extent to build simulation entity with the XML elements. At the same time, ESimL attempts to create a flexible reference standard to represent simulation in order to make simulation model integration and reuse convenient. Different simulation fields can also apply standard XML to describe different simulation methods, such as Petri-Net, Finite state automata and block models.

2 Formalism of simulation model

According to the view of a system, and with the study of DEVS, we categorize simulation model into basic model, composite model and scenario model.

2.1 Basic model formalism

Basic model is one kind of abstract system with the same Attributes and behavior. Basic model is atomic unit, which can not be divided, and can form more complex high level composite model by composition. Basic model can be specified by an 11 tuple.

$BM=<n, pn, A, A', s_0, I, O, \delta, \varepsilon, \tau, \lambda>$

Type constraint：

$n\in\mathbb{NP}, pn\in\mathbb{NP}$;

State constraints：

$A=\{a_i \mid i\in\mathbb{N}\}, A' \subseteq A$;

$S=\{s_i \mid i\in\mathbb{N}\}, s_i=(a_0,a_1\ldots,a_n)$;

$s_0\in S$;

$Q=\{(s, e) \mid s\in S, 0\le e\le\tau(s)\}$;

Interaction constraints:

$I=\{I_i \mid i\in\mathbb{N}\}, X=\{x_i \mid i\in\mathbb{N}\}, \eta_{in}: I\to X$;

$(x,i), x\in \eta_{in}(i), i\in I$;

$O=\{O_i \mid i\in\mathbb{N}\}; Y=\{y_i \mid i\in\mathbb{N}\}, \eta_{out}: O\to Y$;

$(y,o), y\in \eta out(i), o\in O$;

Behavior constraints：

δ： $Q\times X\times I\to S$;

ε： $S\to S$;

τ： $S\to\mathbb{R}^{+}_{0,\infty}$;

λ： $S\to Y\times O$

Among which, n, pn, A, A', s_0, I, O， are the static specification of basic model, δ, ε, τ, λ are the dynamic behavior specification. The meaning of these elements is given as following.

$N\in\mathbb{NP}$, n is the unique name of basic model. $\mathbb{NP}$ is name space.

$Pn\in\mathbb{NP}$, pn is the name of the model's super model. Basic model can inherit the attributes, interfaces and behavior from other basic model.

$A=\{a_i \mid i\in\mathbb{N}\}$, is the attributes set.

$A'\subseteq A$, is the user-setting attributes set which allow the user configure the initial conditions of basic model instance.

s_0 is the initial state of basic model. The state of basic model is a vector composed of all attributes value $s=(a_0,a_1\ldots,a_n)$, The states of all the time sequence form the state set $S=\{s_i \mid i\in\mathbb{N}\}$, $s_0\in S$.

$Q=\{(s, e) \mid s\in S, 0\le e\le\tau(s)\}$, is the total state set. The variable e implies the elapse time of basic model from the last transition, $0\le e\le\tau(s)$, $\tau(s)$ is the time segment between tow transitions.

$I=\{I_i \mid i\in\mathbb{N}\}$, is the input interface set. An interface can be specified as $I_i=<n, ec, tm>$, n is the unique name of the input interface; ec is the event type received from the input interface; tm is the mode of event transmission, $tm\in$ {ware，wareless}. Simulation model can interact with other models by receiving events from the input interfaces.

$X=\{x_i \mid i\in\mathbb{N}\}$, is the input events set. (x, i) is the input vector, with $x\in X$, $i\in I$, imply that input event is received from the input interface i.

$O=\{O_i \mid i\in\mathbb{N}\}$, is the output interface set. Simulation model can interact with other models by sending events to the output interfaces.

$Y=\{y_i \mid i\in\mathbb{N}\}$, is the output events set. (y, o) is the output vector, $y\in Y$, $o\in O$.

δ: $Q\times X\times I\to S$, is the external transition function, which is used to define the operations when receiving input events. The input parameter are the total state q and input vector (x, i), the output result is the updated state after external transition. Its operational semantic is:

$$s\xrightarrow{\delta(q,x,i)}s'$$

ε: $S\to S$, is the internal transition function, which is used to define the operations when the advancing time is reached. The input parameter is the state s, the output result is the updated state after internal transition. Its operational semantic is:

$$s\xrightarrow{\varepsilon(s)}s'$$

τ: $S\to\mathbb{R}^{+}_{0,\infty}$, is the time-advancing function, $\mathbb{R}^{+}_{0,\infty}$ is the positive Integer set. It is used to define the next time of internal transition. The input parameter is the state s, output is the lasting time Δt at current state, $\Delta t=\tau(s)$. When the state of the model is changed, the function τ will be invoked. If the model don't receive external input event during the time segment $[t, t+\Delta t]$, then the internal event will be generated at the time when $e==\Delta t$, and trigger the internal transition ε.

λ: $S\to Y\times O$, is the output function, which is used to output the states when states are changed. The output result

is the vector (y, o).

The execution process of basic model is illustrated in Figure 1. When model receives input event from input interface, the external transition function is triggered, and the states are changed. Time advance function control the time of internal transition function, when the elapsed time $e == ta(s)$, then change to state $\delta_{int}(s)$, then the next internal transition time is advanced to time now + $ta(s)$. When the states are changed, output function $\lambda(s)$ will be invoked and send events to the output interfaces.

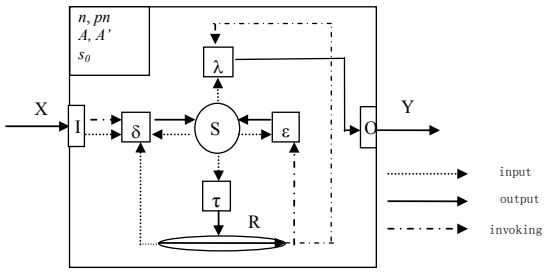

Figure 1: The execution mechanism of basic model

2.2 Composite model formalism

Composite model is a kind of abstract system with the same structure and composed by other basic model or composite model. The behavior of composite model is performed by the sub-models. Composite model can be composed with other models as a basic model to construct higher level composite model. Composite model can be specified by a 10 tuple:

$CM=< n, A', I, O, M, EIC, EOC, IC, HC, Select >$

Type constraint:

$n \in \mathbb{NP}$;

State constraints:

$A'=\{a'_i \mid i \in \mathbb{N}\}$, $A' \subseteq \cup m_i.A'$;

$\Gamma: A' \rightarrow \cup m_i.A'$;

Interaction constraints:

$I=\{I_i \mid i \in \mathbb{N}\}$, $X=\{x_i \mid i \in \mathbb{N}\}$, $\eta_{in}: I \rightarrow X$;

$(x,i), x \in \eta_{in}(i), i \in I$;

$O=\{O_i \mid i \in \mathbb{N}\}$; $Y=\{y_i \mid i \in \mathbb{N}\}$, $\eta_{out}: O \rightarrow Y$;

$(y,o), y \in \eta_{out}(i), o \in O$;

Structure constraints:

$M=\{m_i \mid i \in \mathbb{N}\}$, $m_i=<id_i, n, pn, A_i, A'_i, s_{0i}, I_i, O_i, \delta_i, \varepsilon_i, \tau_i, \lambda_i>$;

$EIC \subseteq CM \times I \times M \times \{m_i.I\}$;

$EOC \subseteq M \times \{m_i.O\} \times CM \times O$;

$IC \subseteq M \times \{m_i.O\} \times M \times \{m_i.I\}$;

$HC \subseteq M \times M$;

Conflict constraint:

Select: 2^M-Ø→M

n is the unique name of composite model.

$A'=\{a'_i \mid i \in \mathbb{N}\}$, is the user-setting attributes set which allow the user configure the initial conditions of composite model instance. $\Gamma: A' \rightarrow \cup mi.A'$, is a mapping function which can map the initial attributes of composite model to it's sub-models.

I, O are the input and output interface set. The definition is same as basic model.

$M=\{m_i \mid i \in \mathbb{N}\}$, is the sub model instances set. Each model instance has the same structure as basic model, it can be specified as $m_i=<id_i, n, pn, A_i, A'_i, s_{0i}, I_i, O_i, \delta_i, \varepsilon_i, \tau_i, \lambda_i>$. id_i is the unique identifier of each model instance.

$EIC \subseteq CM \times I \times M \times \{m_i.I\}$, is the set of links which connect from the input interfaces of the composite model to the input interfaces of the sub-model instances.

$EOC \subseteq M \times \{m_i.O\} \times CM \times O$, is the set of links which connect from the output interfaces of the sub-model instances to the output interfaces of the composite models.

$IC \subseteq M \times \{m_i.O\} \times M \times \{m_i.I\}$, is the set of links which connect from the output interfaces to the input interfaces of sub-model instances.

EIC, EOC, IC define the interaction relation explicitly by connections, which can be looked as the message transmission channel. Each eic, eoc, ic have the same formal structure $<id, m_{src}, if_{src}, m_{sink}, if_{sink}>$, id is the unique identifier of the link, m_{src} is the link source model instance; if_{src} is the link source interface; m_{sink} is the link sink model instance, if_{sink} is the link sink interface.

$HC \subseteq M \times M$, is the set of links which represent the owner relationship. Each hc has the formal structure $<id, m_{src}, m_{sink}>$, m_{src} is the model instance own the instance m_{sink}.

$Select$: 2^M-Ø→M, is the tie-breaking function, which is used to select one sub-model instance to process the simultaneous internal events.

The following figure is the black-box view and white-box view of composite model.

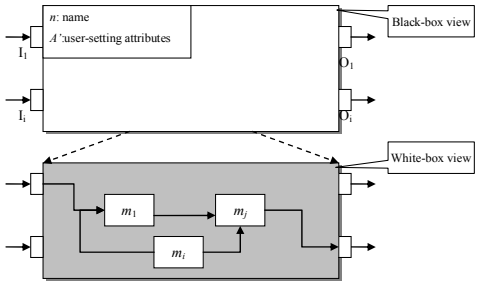

Figure 2: Composite model view

2.3 Scenario model formalism

Scenario model is one kind of abstract system with the information of models and simulation. Scenario model define the models anticipating simulation and the simulation setting, which can be regarded as one composite model instance with the simulation configuration.

$SM=<T_b, T_e, C_e, Batch, cm>$

Simulation constraints:

$T_b \in \mathbb{R}$;

$T_e \in \mathbb{R}$;

$C_e \in$ {true, false}

$Batch \in \mathbb{N}$

Model constrant:

$cm =< id, n, A', I, O, M, EIC, EOC, IC, HC, Select >$;

In which, T_b is beginning time of simulation; T_e is end time of simulation; C_e is end condition of simulation; *Batch* is the simulation times of a scenario.

$cm =< n, A', I, O, M, EIC, EOC, IC, HC, Select >$, is a composite model instance which contains the models configuration of current simulation.

3 ESimL

According to the basic formalism of simulation model, combined with XML language, an Executable Simulation Language (ESimL) has been raised. Some other simulation languages are also based on XML such as SRML [5] [6] and SML [7] [8]. ESimL is a simulation language based on XML and script language, which supports model representation and execution. It uses a group of XML tags to represent simulation model's static structure and infuses script language to describe simulation model behavior. The corresponding relationship between ESimL elements and simulation model formalism is summarized in Table 1 below:

Table 1. Relationship between ESimL elements and simulation model formalism

Simulation model formal specification	ESimL elements
BM	*ItemClass* element
I	*EventSink* elements set
O	*EventDispatcher* elements set
A	*Property* elements set
δ	*Script* element with *function eventSinkName*(*Event* event)
ε	*Script* element with *function eventSinkName*(*Event* event)
λ	*Script* element with simulator API *SendEvent*(*String* eventDispatcherName, Event event) *PostEvent*(*String* eventDispatcherName, *Event* event) *BroadcastEvent*(*Event* event, *Boolean* direction)
τ	*Script* element with simulator API *ScheduleEvent*(*String* eventSinkName, Event event) *PostEvent*() and *BroadcastEvent*()
CM	*ItemClass* element
I	*EventSink* elements set
O	*EventDispatcher* elements set
M, *EIC*, *IC*, *EOC*, *HC*	*Item* elements set with *link* element
Select	Simultaneous events will be processed according as the priority of model instances
SM	*Simulation* element

Basic model *BM* and composite model *CM* are declared by *ItemClass* element. Scenario model *SM* is declared by *Simulation* element. Input and output interfaces of basic model and composite model are declared by *EventSink* and *EventDispatcher* elements. Basic model's property *A* is declared by *Property* element. Composite model's sub-model sets and various connecting relationships *M*, *EIC*, *IC*, *EOC*, *HC* are declared collectively by *Item* element and *Item*'s sub element *Link*. The above are all model's static attributes description, which has the same meaning in Part 2's formal specification. Basic model's dynamic behavior attributes specification δ, ε, λ, τ is conducted by script language in *Script* element.

δ, ε transition functions, implementation formalism in javascript language is displayed in the following code. *Function* is the javascript function declaration. *EventSinkName* is the transition function name which must be identical with the name of one of *EventSink* elements defined in *ItemCla*ss. *Event* is the parameter of the transition function, which is the received event. Transition function can update model's states by the reference of external event or internal event.

```
function OnEventSinkName(event){
    var x=event.x;
    var y=event.time;
    …
}
```

λ, τ functions are implemented with the API offered by ESimL virtual machine. *SendEvent*() is used to send synchronous event to the output interface, *PostEvent*() is used to send asynchronous event to the output interface, *BroadcastEvent*() is used to send synchronous or asynchronous event among the composite models at different levels. *ScheduleEvent*() is used to schedule the next internal event by the model itself , *PostEvent*() and *BroadcastEvent*() can also perform the time advancing behavior when they send asynchronous events.

The architecture of ESimL schema is illustrated in Figure 3. ESimL schema is composed of *ESimL*,

EventClass, *ItemClass*, *Simulation*, *Script*, *Property*, *EventDispatcher*, *EventSink*, *Item*, *ItemPrototype*, *PropertyValue*, *Link* and *Event*, 13 elements all together.

ESimL is the root element of ESimL, which can comprise zero or more *ItemClass*, *EventClass* and *Simulation* as its sub-elements. ESimL virtual machine can load ESimL file and parse the definition of *ItemClass*, *EventClass* and *Simulation*, then generate model instances and events, drive the simulation. *EventClass* element is used to define the event class which is the data format used for interaction among model instances. All events used in the ESimL must be one type of defined event class. *ItemPrototype* element is used to define the prototype of one *ItemClass*. A prototype is one kind of *ItemClass* with the same property value. *Item* can also be instantiated from an *ItemPrototype*. *PropertyValue* element is used to initialize the property value. *Link* element is used to define the interaction relationship between model instances. *Event* can define the event instance in a simulation.

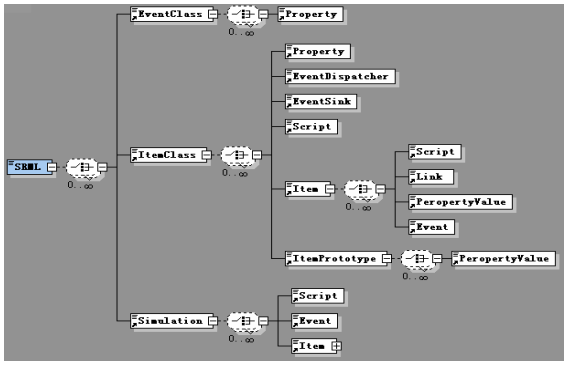

Figure 3: The architecture of ESimL schema

4 ESimL virtual machine

ESiml virtual machine (EVM) is used to interpret and execute the simulation model represented by ESimL. EVM comprises root simulator and basic simulator. Root simulator takes charge of controlling the whole simulation according to scenario model. Basic simulator takes charge of executing the basic model and composite model.

4.1 Root simulator algorithm

```
1while（simNo ≠Batch）do
2    when receive（START, T_b）
3        t←T_b
4        send（START, t） to cm
5        while（t ≠T_e ∧ C_e ≠ true）do
6            t←t_Ncm
7            "internal transition"
8            send（TRANSITION, t）to cm
9        end while
10   end
11   simNo++
12end while
```

While simulation times are less than *Batch*, continue next one until simulation run times are identical with *Batch*. When receive initialization signal(START， T_b), start a simulation, update current time $t \leftarrow T_b$, T_b is simulation begin time; Send initialization signal to the simulator of composite model *cm*, *cm* will initialize its sub-models. While simulation end condition is not fulfilled, advance to next time $t \leftarrow t_{Ncm}$, send internal transition signal to *cm*.

4.2 Basic simulator algorithm

The tasks of basic simulator are to process the basic and composite model's actions of initialization, external transition, internal transition and output. Correspondently basic simulator will receive 4 kinds of signals, which are (START, *t*), (TRANSITION, *t*), (*x*, *i*, *t*) and (*y*, *o*, *t*).

(1) Initialization

```
1when receive (START, t)
2    t_L ← t
3    if M == Ø
4        s ←s_0
5        t_N ← t +τ(s)
6    else
7        ∀ m ∈ M : send (START, t) to child m
8        t_N ←min { t_N ∪ t_Nm | m ∈M}
9    end if
10end
```

When receiving initialization signal (START, *t*), initialize the last transition time $t_L \leftarrow t$. If current model is basic model ($M == \emptyset$), then initialize the state $s \leftarrow s_0$, and the next transition time $t_N \leftarrow t + \tau(s)$. Otherwise, current model is the composite model, then simulator will send initialization signal to all sub-models and calculate the next transition time $t_N \leftarrow \min \{ t_N \cup t_{Nm} \mid m \in M \}$.

(2) External transition

```
1when receive (x, i, t)
2    if t ∉ [ t_L, t_N] ERROR end if
3    if M == Ø
4        e ← t - t_L
5        s ←δ(s, e, x)
6        o←∏_O(λ(s))
7        y←∏_Y(λ(s))
8        send (y, o, t) to parent
9        t_L← t
10       t_N← t +τ(s)
11   else
12       ∀ m∈∏_M (Self ⋈ EIC ⋈ M)
13           ∀ i∈∏_m.I (i ⋈ EIC ⋈ m.I)
14               send (x, i, t) to m
15       t_L←max { t_L | m ∈M }
16       t_N←min { t_N | m ∈M }
17   end if
18end
```

When receiving external signal(x, i, t), simulator will execute external transition action. If $t \notin [\, t_L, t_N]$, that implies some exceptions occur, simulation will terminate. If current model is the basic model, calculate the elapse time from last transition, $e \leftarrow t - t_L$; invoke external transition and update the state, $s \leftarrow \delta(s, e, x)$; calculate output vector (y, o), $o \leftarrow \prod_O(\lambda(s))$, $y \leftarrow \prod_Y(\lambda(s))$, and send ($y$, o) to parent simulator, which will translate output signal to other model's input signal; update t_L and t_N, $t_L \leftarrow t$, $t_N \leftarrow t + \tau(s)$. If current model is the composite model, then find the destination sub-models $\prod_M (i \bowtie EIC \bowtie M)$ and their input interfaces $\prod_{m.I} (i \bowtie EIC \bowtie m.I)$, $m \in \prod_M (i \bowtie EIC \bowtie M)$, update external signals (x, i, t), $i \in \prod_{m.I} (i \bowtie EIC \bowtie m.I)$, and send the signal to all sub-model m; update the time $t_L \leftarrow \max \{ t_L \mid m \in M \}$, $t_N \leftarrow \min \{ t_N \mid m \in M \}$.

(3) Internal transition

```
1when receive (TRANSITION, t)
2    if t ≠ t_N ERROR end if
3    if M == Ø
4        e ← t - t_L
5        s ←ε (s)
6        o←∏_O(λ(s))
7        y←∏_Y(λ(s))
8        send (y, o, t) to parent
9        t_L← t
10       t_N← t + τ(s)
11   else
12       m ← select (M)
13       send(TRANSITION, t) to child m
14       t_L←max { t_L | m ∈M }
15       t_N←min { t_N ∪ t_Nm | m ∈M }
16   end if
17end
```

When receiving internal transition signal (TRANSITION, t), simulator will execute internal transition action. If $t \neq t_N$, that implies some exceptions occur, simulation will terminate. If current model is the basic model, calculate the elapse time from last transition, $e \leftarrow t - t_L$; invoke internal transition and update the state, $s \leftarrow \varepsilon(s)$; calculate output vector (y, o), $o \leftarrow \prod_O(\lambda(s))$, $y \leftarrow \prod_Y(\lambda(s))$, and send ($y$, o) to parent simulator, which will translate output signal to other model's input signal; update t_L and t_N, $t_L \leftarrow t$, $t_N \leftarrow t + \tau(s)$. If current model is the composite model, select one sub-model by tie-breaking function to execute internal transition, $m \leftarrow$ select (M), and send internal transition signal (TRANSITION, t) to m; update t_L and t_N, $t_L \leftarrow \max \{ t_L \mid m \in M \}$, $t_N \leftarrow \min \{ t_N \mid m \in M \}$.

(4) Output

```
1when receive (y, o, t) from child m
2    ∀ o' ∈∏_O (o ⋈ EOC ⋈ Self.O)
3        send (y, o', t) to Self
4    ∀ m' ∈∏_M (m ⋈ IC ⋈ M)
5        ∀ i∈∏_I (o ⋈ IC ⋈ m'.I)
6            x←y
7            send (x, i, t) to m'
8end
```

Output algorithm is only for composite model, which is responsible for the output signal transmission of sub-models. When receiving sub-models' output signal (y, o, t), simulator will execute output action. Find the output interfaces $\prod_O (o \bowtie EOC \bowtie Self.O)$ of the current composite model by the links set EOC, then send the output signal (y, o', t), $o' \in \prod_O (o \bowtie EOC \bowtie Self.O)$; Find the destination sub-models $\prod_M (m \bowtie IC \bowtie M)$ and their input interfaces $\prod_I (o \bowtie IC \bowtie m'.I)$, $m' \in \prod_M (m \bowtie IC \bowtie M)$ by the links set IC, update the input signal (x, i, t), $i \in \prod_I (o \bowtie IC \bowtie m'.I)$, then send the signal to m'.

5 Simulation Case Study

Use ESimL to build Lorenz system model [9]. Values of the parameters are σ=10，λ=24，b=2，initial value $x_i(0)$=1.0.

$$\dot{x}_1 = \sigma(x_2 - x_1)$$

$$\dot{x}_2 = (1 + \lambda - x_3)x_1 - x_2$$

$$\dot{x}_3 = x_1 x_2 - b x_3$$

To build this differential equation model, we need to build relevant addition model, subtract model, constant model and integrator model. We also have to define the interaction data formats and then construct these model instances according to the equation and define the simulation scenario to begin simulation finally. The followings are part of the main ESimL code to approve its description ability.

The code for interaction data formats definition:

```
<EventClass Name = "Number">
    <Property Name = "Number" DataType = "Double"/>
```

```
</EventClass>
```

The code for integrator model definition:

```
<ItemClass Name="IntegratorBlock">
    <Property Name="InitValue" DataType ="Double"/>
    <Property Name="DeltaValue" DataType ="Double"/>
    <Property Name="Step" DataType ="Double"/>
    <Property Name="Result" DataType ="Double"/>
    <EventSink Name = "dValue" EventClass="Number"  LinkFixed = "True"/>
    <EventSink Name = "Tick" EventClass="Tick"  LinkFixed = "True"/>
    <EventDispatcher Name = "CurrentValue" EventClass="Number"/>
    <Script Type="text/javascript"><![CDATA[
        This.Step = 0.001;
        var event = new Event("Tick");
        event.Time = This.Step;
        Simulation.ScheduleEvent("Tick",event);
        function OndValue(e){
            This.DeltaValue = e.Number;
        }
        function OnTick(e){
            This.Result = This.Result + This.DeltaValue * This.Step;
            var event = new Event("Number");
            event.Number = This.Result;
            event.Time = Simulation.CurrentTime;
            Simulation.SendEvent("CurrentValue",event);

            event = new Event("Tick");
            event.Time = Simulation.CurrentTime + This.Step;
            Simulation.ScheduleEvent("Tick",event);
        }
    ]]></Script>
</ItemClass>
```

The code for simulation scenario definition:

```
<Simulation Name = "Lorenz" StartTime = "0" EndTime = "50" Batches = "1">
    <Item ItemClass = " Lorenz Model" ItemID = "10">
    </Item>
</Simulation>
```

Visual simulation model and simulation results are displayed in Figure 4 below:

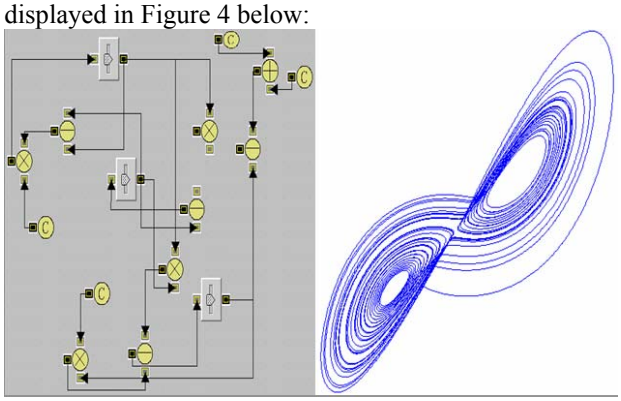

Figure 4: Lorenz system model and simulation result

References

[1] Andreas Tolk. Avoiding another Green Elephant – A Proposal for the Next Generation HLA based on the Model Driven Architecture. 2002 Fall Simulation Interoperability Workshop, Orlando, Florida, September 2002.

[2] Andreas Tolk. Composable Mission Spaces and M&S Repositories - Applicability of Open Standards. 2004 Spring Simulation Interoperability Workshop Washington, D.C., April 2004

[3] Zeigler, B.P., Theory of System Modeling and Simulation . New York: Academic Press, 2000.

[4] Ki Jung Hong, Tag Gon Kim, and In Sup Kwon. DEVSIF: RELATIONAL ALGEBRAIC DEVS INTERMEDIATE FORMAT.2002.

[5] Steven W. Reichenthal. SRML - Simulation Reference Markup Language. W3C Note. 2003. <http://www.w3.org/TR/SRML/>

[6] Steven W. Reichenthal. SRML: A Foundation for Representing BOMs and Supporting Reuse. SIW Fall Conference, 2002.

[7] SML, Simulation Modeling Language. Available online via http://www.threadtec.com/sml [accessed August 1, 2002].

[8] E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds.. NEXT GENERATION SIMULATION ENVIRONMENTS FOUNDED ON OPEN SOURCE SOFTWARE AND XML-BASED STANDARD INTERFACES. Proceedings of the 2002 Winter Simulation Conference, 2002.

[9] Fishwick, P. A., Simulation Model Design and Execution, Prentice-Hall Inc, 1995.