A FRAMEWORK FOR DISCRETE EVENTS SYSTEMS ENACTMENT

Hamzat Olanrewaju Aliyu[1,3]
[1]School of Info. and Comm. Tech.
Federal University of Technology
Minna, Nigeria
hamzat.aliyu@futminna.edu.ng

Oumar Maïga[2,3]
[2]Université des Sciences,
des Techniques et des Technologies
Bamako, Mali
maigabababa78@yahoo.fr

Mamadou Kaba Traoré[3]
[3]LIMOS CNRS UMR 6158
Université Blaise Pascal
Clermont-Ferrand, France
traore@isima.fr

KEYWORDS
Enactment, Discrete Events System, System Analysis, Prototyping, Design Patterns

ABSTRACT

This paper proposes a framework to guide the synthesis of program codes from Discrete Event Systems (DES) models for the enactment of the systems. Enactment in this context is the execution of a system's specification for real time verification of the specified properties and/or building a software solution for the system. Though it has been used extensively within the last decades to automate workflows and business processes, enactment is less pronounced mainstream computational system analysis domain. We believe that enactment of DES models can complement the more exploited methodologies like simulation and formal methods in model-based systems analysis. We propose a framework that provides a template for code synthesis from DEVS(Discrete Events System Specification)-based models and an execution protocol based on Object-Oriented Observer design pattern for the real time interpretation of system's properties. We provide a simple case study to illustrate the use of the framework.

INTRODUCTION

Simulation and testing have been used extensively within the last decades in the study, development and improvement of complex systems. Simulation and Formal Methods of various categories are most pronounced in this domain for studying dynamic and static system properties respectively. While simulation techniques are mostly scenario-based investigations of properties using time approximations through the advancement of execution time-at discrete steps- to the times of occurrences of events of interest, most formal methods deal with the static proof of the satisfaction or otherwise of certain quality and reliability properties (e.g., completeness, safety, deadlock freedom) throughout the entire life cycle of the system represented by the specification. It is a general belief that no single analysis methodology is sufficient to study all aspects of a system, hence multiple techniques are used to get complementary insights of the system.

Another system analysis (cum implementation) methodology which is more pronounced in business process management (BPM) (Van Der Aalst et al. 2003, Jeston and Nelis 2014) is enactment. In the field of BPM, enactment may be simply described as the execution of process definitions created by a workflow (Kouvas et al. 2010) where a workflow is described as the complete or partial automation of business processes during which a set of procedure rules is used to pass information and work lists from one participant to another for necessary actions (Ottensooser and Fekete 2007). A more general software engineering description of enactment provided in (Dowson and Fernström 1994) is the execution or interpretation of software process definitions. According to the authors, an enactment mechanism may also interact with the environment (e.g., human-in-the-loop, software and hardware devices) to provide supports that are consistent with the process definitions; this property, interaction with external actors, is in fact another feature that differentiates enactment from mainstream simulation mechanisms in addition to the execution of system's functionalities in real time. Finally, in service engineering and Human-Computer Interaction, it can be inferred from (Holmlid and Evenson 2007) that enactment is used to describe the playing out of the functionalities represented by a prototype where a prototype is described as an object that represents the functionality but not the appearance of a finished artifact which can be used as a proof that a certain theory or concept or technology works or otherwise (Holmquist 2005).

To be able to verify a system's behavior in real time, there is need for an operational model of the system; an operational model in this context is one that can be executed in a suitable software environment (Bruno and Agarwal 1995). Analysis of traces generated from such executions can give further insights into the system's behavior as well as point out certain inconsistencies, missing requirements, verification of timing correctness in real-time systems etc. Using appropriate model-driven software engineering techniques, such executable programs to enact systems' properties can be synthesized from models created in some modeling environments. But before then, we must address questions such as "what should be the structure of the so-called operational program? What is the operational semantics of the chosen structure? ...". We try to address some of the possible questions with the framework proposed in this paper.

Wepropose an Object-Oriented framework that facilitates the synthesis (and specification) of operational (executable) representation of DES models for the enactment of such systems. In order to be general enough to accommodate a large category of DESs, our description of DES is guided by DEVS (Zeigler et al. 2000), a mathematical formalism that provide a sound basis for hierarchical description of DESs based on system-theoretic. Our choice of DEVS for generality is informed by the fact that it is considered to provide a common platform for describing most kinds of DESs and even approximated models for some kinds of non-discrete event systems (Vangheluwe 2000).

The challenge here, however, is that the operational semantics of DEVS is a simulation protocol while what we require is a semantic to drive the execution of the specified

behavioral processes. In this paper, we explore the mapping of DEVS concepts onto the Object-Oriented "Observer design pattern" (Gamma et al. 1994) to provide an execution semantics. We have chosen the observer design pattern to take benefit of its natural dialect for enacting the reactive systems and its ease of implementation in most general purpose programming languages. Of course it has some limitations that put its absolute suitability in question. We show in a later section, the measures we have taken to palliate some of these deficiencies (at least those that could have significant effects on the objective of the work).

We present overviews of DEVS formalism and relevant software engineering design patterns in the next section to set the scene for the reader to follow subsequent sections; then we compare and contrast the framework's intent and methodology with those of some related work and finally, Afterwards, we present the essential elements of the framework followed by a case study to show its usability. we conclude the paper with discussions and perspectives.

BACKGROUND

Discrete Events System Specification (DEVS)

DEVS (Zeigler et al. 2000) is a system-theoretic mathematical formalism for specifying DESs as abstract mathematical objects for simulation.

Basically, DEVS defines two abstraction levels for DESs - atomic and coupled DEVS. An atomic DEVS has a time base; state, input and output sets; and functions that define successive states and outputs events. A coupled DEVS on the other hand is an hierarchical composition of two or more atomic and/or coupled DEVS as components while specifying couplings between their input/output ports to enable their interactions.

Traditionally, DEVS exists in two major forms: classic DEVS (CDEVS) (Zeigler, 1976) and parallel DEVS (PDEVS) (Chow and Zeigler 1994, Chow, 1996), the main difference being that the latter supports concurrent state transition events within components of a coupled DEVS while the former does not. In this paper, we present PDEVS.

Atomic DEVS (AM), which is defined as:

$$AM = < X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta > \quad (1)$$

$$X = \{(p, v), p \in IPort \wedge v \in dom(p)\} \quad (2)$$

$$Y = \{(q, v), q \in OPort \wedge v \in dom(q)\} \quad (3)$$

X and *Y* are the sets of input and output events respectively. *IPort* and *OPort* are the sets of input ports and output ports respectively. *S* is the set of states and at any given moment, a DEVS model is in a state $s \in S$.

$$ta: S \rightarrow \mathbb{R}^{+}_{0,\infty}\,;\ \lambda: S \rightarrow Y^{b};\ \delta_{int}: S \rightarrow S;$$

$$\delta_{ext}: Q \times X^{b} \rightarrow S \text{ and } \ \delta_{conf}: S \times X^{b} \rightarrow S.$$

$$Q = \{(s, e) | s \in S, e \in [0, ta(s)]\}$$

The time advance function, *ta,* maps each state to a duration after which an internal state transition, δ_{int} is automatically fired. The external transition function, δ_{ext} where *Q* is called the set of total states and *e* is the elapsed time since the last state transition, defines the system's response to an input event when the time advance of the current state has not expired. If the input event coincides with the expiration of the time advanced, the confluent transition function, δ_{conf} is invoked instead. The function λ defines the outputs that may accompany internal and/or confluent state transitions.

Coupled DEVS, CM which is defined as:

$$CM = < X, Y, D, \{M_d\}_{d \in D}, EIC, EOC, IC >$$

$$EIC = \left\{ \begin{matrix} ((M, ip_M), (d, ip_d)) | \\ ip_M \in IPorts_{M,} ip_d \in IPorts_d \end{matrix} \right\}$$

$$EOC = \left\{ \begin{matrix} ((d, op_d), (M, op_M)) | \\ op_M \in OPorts_{M,}\ op_d \in OPorts_d \end{matrix} \right\}$$

$$IC = \left\{ \begin{matrix} ((a, op_a), (b, ip_b)) | \\ \in OPorts_a\ , ip_b\, op_a \in IPorts_b \end{matrix} \right\}$$

X and Y are as defined in (2) and (3)respectively. D is the set of component names with the specification of component $d \in D$ represented by M_d. *EIC* is the external input coupling relation, *EOC* is the external output coupling relation and *IC* is the internal coupling relation. The reader may consult (Zeigler et al. 2000) for further details about DEVS formalism and its operational semantics.

Object-Oriented Design Patterns

Design patterns in Object-Oriented modeling are documented solutions to some general problems that can be reused to build new models. In this subsection, we present the overviews of two design patterns from (Gamma et al. 1994) that are re-used in later sections to define the metamodel of our enactment framework.

Observer Design Pattern

It is a behavioral pattern for establishing relationships between objects at runtime such that changes in the state of an object (referred to as subject) trigger some actions in another (the observer). It is defined by the Gang of Four (Gamma et al. 1994) as a pattern that "*define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.*"

Figure 1 shows an overview of the observer pattern. The basic idea is that the *Subject* maintains a list of references to some independent objects called the *Observers*. Whenever there is a change of state in the subject, all its observers must be notified by the invocation of the *update* method of each of them. Each observer (i.e., ConcreteObserver) must implement its update method to implement the corresponding actions to be taken whenever this happens. This pattern is widely used in Graphical User Interface (GUI) programming and it provides the underlying principle for the Model-View-Controller (MVC) architecture (Krasner and Pope 1988) so that all views are automatically updated whenever there is a change of state in the model.

Command Design Pattern

The *command* design pattern is shown in Figure 2. A command in this context means a method call. The pattern provides a methodology to encapsulate a command in an object and issue it (the command) in such a way that the requested operation and the requesting object do not have to know each other.

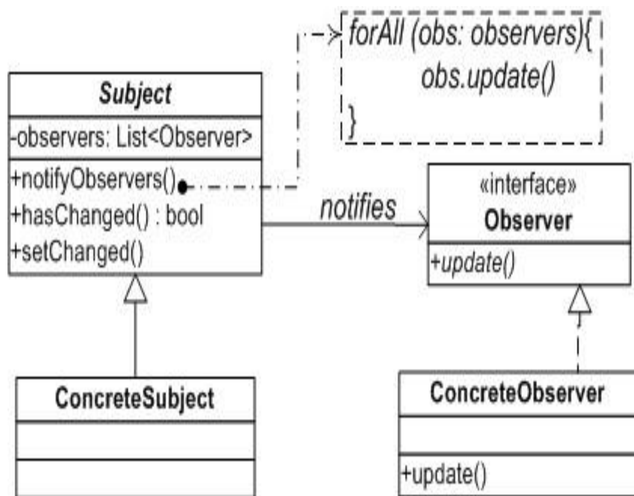

Figure 1: Observer Design Pattern

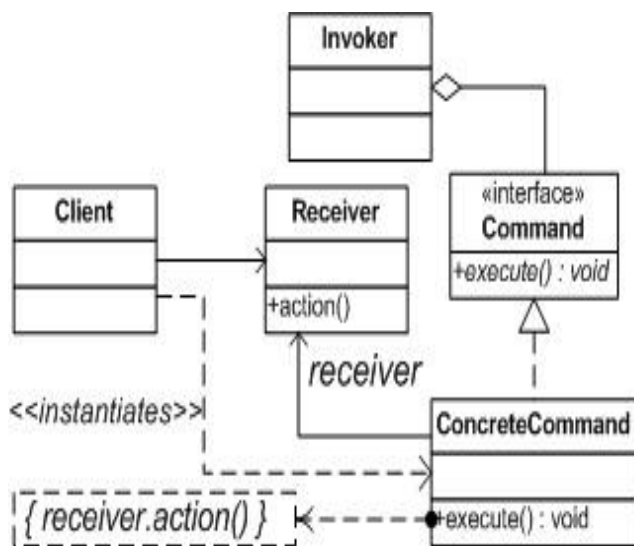

Figure 2: Command Design Pattern

From Figure 2, *Client* is the requesting object while the method *action*() of *Receiver* is the requested operation. *Client* creates the request command and delegates its execution to the *Invoker* which manages a queue of command threads. The invoker identifies the receiver of the request carried by each command in its queue and then executes the command. When its *execute()* method is invoked, the command delivers its request by invoking the appropriate *action()* method. This pattern provides a methodology for asynchronous (i.e., non-blocking) method call, sharing of a method call among multiple objects, saving method calls in a queue so that they are executed when the necessary conditions have been satisfied, etc. It has also been used to decouple clients from server methods in Asynchronous Remote Method Invocation (ARMI) (e.g., Raje et al. 1997)

RELATED WORK

PROTOB (Baldassari et al. 1989; Baldassari and Bruno 1991) is a system development environment that integrates tools, for modeling, prototyping and implementation of distributed systems using an operational software life cycle paradigm. In PROTOB, systems are described with PROT nets, an Object-Oriented formalism that combines high-level features of timed Petri nets, and workflows to model event-driven distributed systems. PROT nets describes a system as consisting of interacting autonomous objects called "actors" where each actor is an instance of a class. The behaviour of a class is described in a Petri nets dialect as consisting of *places* (describing the states) and *transitions* through which places are connected with *arcs*. An active place has a queue of message-carrying *tokens* that are moved from places to places through transitions. Some places are designated for Input/output operations to allow actors to interact with one another. Message passing between actors is achieved by moving tokens between their I/O interfaces. According to the authors, operational program codes can be generated from PROT nets specifications for general purpose languages though it is not clear what the structures of such codes look like. The similarity between PROTOB approach and the enactment framework presented in this paper is that the system description in both cases are based on some well defined formalisms - Petri nets in PROTOB and DEVS in our framework. Interestingly, the approach proposed in this paper can arguably accommodate a broader category of DESs based on the fact that the underlying formalism, DEVS, has been proven to provide a common denominator for most DES formalisms including Petri nets (Vangheluwe 2000).

Another interesting work that is related to that presented here is the one discussed in (Hu and Zeigler 2004). It proposed an approach of Model Continuity to Support Software Development for Distributed Robotic Systems based on Modeling-Simulation-Execution methodology (Hu and Zeigler 2002). As defined by the authors, Model continuity refers to the ability to use the same model of a system throughout its design phases, provides an effective way to manage this development complexity and maintain consistency of the software. Model continuity is ensured by using the same model in modeling, simulation and execution phases. Real-Time DEVS and Dynamic Structure concepts are used in modeling phase in order to support the modeling of the robots sensors and actuators as activities and dynamic reconfiguration of robots. In the simulation phase, different DEVS simulator implementations (supporting different communications schemes from point to-point socket communication to advanced middleware such as CORBA) are used for the incremental verification of the model. The real-time execution is achieved by mapping the robot specifications into a real hardware execution environment controlled by DEVS real-time execution engine. It is, however, not clear what is the methodology used in building the said execution engine. The main similarity with this work and that presented in this paper is that the system description is based on DEVS-based systems in both cases. It is however different from ours in that the enactment engine proposed in the work resides in hardware for enactment of robot systems while ours is a software enactment on any suitable system.

ENACTMENT FRAMEWORK FOR DES

The methodology we propose is to express DEVS-based concepts using the dialect of the observer design pattern for the purpose of enactment. We do this by registering system ports as observers of other ports that may influence them. However, we acknowledge the fact that the notification process in the observer pattern poses some undesired effects

during the exchange of messages between ports; the processes of the system sending the message will be blocked until the receiving system finishes treating the message received. The effect is even more complicated when there is a cascade of notifications.

This is due to the synchronous calls to the update methods of the observers. We have tried to address this problem by using the command pattern to decouple the subject from its observer during notifications.

Figure 3 shows our attempt to introduce an asynchronous message passing between the subject and its observers to make it more suitable for enacting systems' behaviors in real time. Compared to the command pattern presented in Figure 2, *Subject*, *Observer, Notifier, Notification* and *ConcreteNotification* are equivalent to *Client, Receiver, Invoker, Command* and *ConcreteCommand* respectively.

Therefore, the subject will delegate the notifications of observers to *Notifier* and continue its activities. Since the subject does not expect any return value from these method calls, it is easy to just use the "fire and forget" approach. Notifier has a pool of threads to which the requests are assigned on arrival, hence it does not have to always create threads thereby minimizing the overhead that may be incurred due to thread creation.

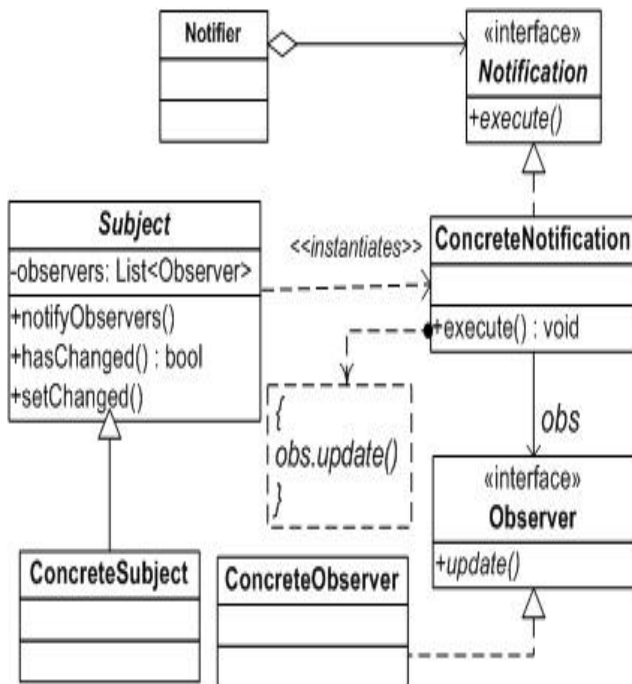

Figure 3: Observer Pattern with Asynchronous Notification

Meta Model of the Framework

We present the metamodel of the enactment framework in the segment of Figure 4 that is enclosed within a dashed box. Using the observer pattern with asynchronous notification, a DES is described as the AbstractSystem which implements the *Observer* interface while its generic input and output ports can observe and be observed by other objects. A system has a clock that monitors the time advance of the current state; the clock inherits the Subject class so that it can notify the system at appropriate instants.

All methods in the *AtomicSystem* and *CoupledSystem* classes are abstract; therefore the concrete atomic and coupled system classes using the framework must implement them to provide the specific elements of the system being modeled. The *update* method of the *AbstractSystem* class has the implementation of the enactment protocol (to be provided in the next sub-section) which calls the user-defined functions when they are needed. The *doInternalTransition, doExternalTransition* and *doConfluentTransition* allow the user to describe the internal, external and confluent transition functions respectively. *setCurrentStatus* method is used to define the system's states based on the instantaneous values of the state variables to be defined by the user in the concrete class. Similarly, *mapTimeAdvance* and *mapOutputEvents* methods must be implemented to provide the time advance and output functions respectively. Method *mapActivities* can be used to define the activities to be enacted for each state during execution. An activity is a set of operations that do not lead to change in state variables, reception of inputs and output events.

Coupling between any two ports in the in the *CoupledSystem* is realized by adding the target port to the list of observers of the source port.

Enactment Protocol

A transition in the state of an *AtomicSystem* can be triggered by a timed event (an automatic notification from the clock when the time advance of the current state has elapsed), an input event (a notification from an input port upon receipt of a new value) or both. By default, an *AtomicSystem* is a registered observer of its *clock* and all its input ports, so this allows for automatic notifications from both sides. In any case, an event is an object that encapsulates a message(value) and information about the nature of its source, whether a port or clock. When the system receives notifications, all events received are stored in the event bag (*eventBag*) of the system. Then the system's reaction will depend on the content of the bag.

If the event bag contains a time event, then it sends outputs (if any) to the appropriate output port(s) and then check if there are also input events in the bag. If a port event is found, then the *doConfluentTransition* method is invoked, otherwise *doInternalTransition* method is invoked. If the event bag contains only input events, then *doExternalTransition* method is invoked.

Implementation

We have implemented the framework's metamodel and enactment protocol in Java. To use the framework, we can simply create classes inheriting from the AtomicSystem and CoupledSystem classes of the framework to get the skeletons the appropriate system unit. The user only needs to specify the properties that are peculiar to the system under study while the enactment mechanism is driven by the framework. Based on this implementation, we present a simple case study in the next section to illustrate its use.

Figure 4: Metamodel of Enactment Framework

CASE STUDY

We present a small example of a traffic light control system to illustrate the extension of the enactment framework for real time execution of DES. The system consists of two components, control and display. The four states of the control, their durations and the corresponding light color to be on the display unit are summarized in Table 1. The control unit has only one output port which is connected to the only input port of the display unit as described in Figure 5. Whenever, there is a change in the state (internal transition) of the control unit, it sends an output which is received by the display unit to show the appropriate light color to the road user.

Table 1: Specification of Traffic Light System

Control states	Duration of control state (units)	Display color
ready	3	yellow
moving	10	Green
braking	3	Yellow
stopping	5	Red

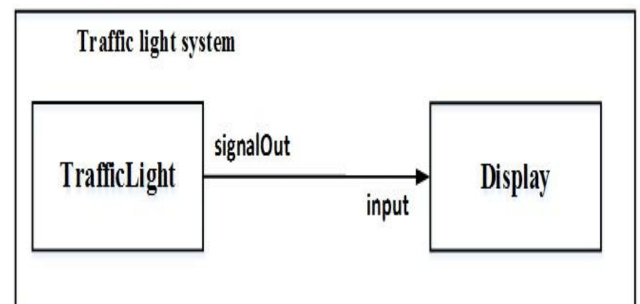

Figure 5: Traffic Light System

The specification of the system is presented in Figures 6-9. The control unit is shown in Figure 6. It is an atomic unit, so it has to *extend* the framework's *AtomicSystem* class which provides the required system-specific methods to be completed as indicated by the methods with *@override* annotation in Figure 6. The single output port is created using the *addOutputPort* method provided by the framework.

The display unit is presented in Figure 7. It is also an atomic unit and maintains only one state with approximately infinite time advance as indicates by the *Long.MAX_VALUE* in the *computeLifeSpanFunction*. So, it will never receive a *time event* since the time advance will never expire. Therefore,

```
package example;
import enactment.AtomicSystem;
import enactment.Port;
import enactment.Utilities;
import enactment.designExceptions.*;
import java.util.ArrayList;

public class TrafficLight extends AtomicSystem {
    private enum Status{MOVING, BRAKING, STOPING, READY};
    private Status state;
public TrafficLight(String name) {
    super(name);
    registerPorts();
    state=Status.READY;}

private void registerPorts(){
    try {//addInputPort(name, type); addOutputPort(name, type);
        addOutputPort("statusOut", new String());
    } catch (DuplicateIdException e){e.printStackTrace();}
}
@Override
protected long computeLifeSpanFunction() {
    long ltime;//ltime is in milliseconds
    if (state==Status.BRAKING) ltime = 3000;
    else if(state==Status.MOVING) ltime = 10000;
    else if(state==Status.STOPING) ltime = 5000;
    else if (state==Status.READY) ltime = 3000;
    else ltime=2000; return ltime;}
@Override
protected void doInternalTranssition() {//internal transition function
    mapOutputEvents(state); //send output on port"statusOut"
    System.out.println(Utilities.getCurrentTime()+":"+" "+
    getName().toUpperCase()+": Internal transition from "+state);
    if (state==Status.MOVING) state = Status.BRAKING; //set target state
    else if(state==Status.BRAKING) state =Status.STOPING;
    else if(state==Status.STOPING) state = Status.READY;
    else if (state==Status.READY) state =Status.MOVING;
    else state =Status.MOVING;
    System.out.println(Utilities.getCurrentTime()+":"
    +" "+this.getName().toUpperCase()+": New state: "+state);}
@Override
protected void doExternalTransition(ArrayList<Port> eventBag){
    // No external transition specified}
@Override
protected void doConfluentTransition(ArrayList<Port> eventBag){
    // No confluent transition specified}
@Override
protected void mapOutputEvents(Object event) {
    Status st = (Status)event;
    if (st==Status.READY) sendMessage("statusOut","GREEN");
    else if(st==Status.MOVING) sendMessage("statusOut","YELLOW");
    else if(st==Status.BRAKING) sendMessage("statusOut","RED");
    else sendMessage("statusOut","YELLOW");
}
@Override
protected void mapActivities() {
    // no activities specified}
}
```

Figure 6: The Control Unit of Traffic Light System

only external transition is possible. Whenever, it receives an input event (which is an instruction from the control unit), it changes the color of light displayed to the new color received.

```
package example;
import java.util.ArrayList;
import enactment.AtomicSystem;
import enactment.Port;
import enactment.Utilities;
import enactment.designExceptions.*;
public class Display extends AtomicSystem {
    private String color;
    public Display(String name) {
        super(name); registerPorts();
        color = "RED";  }
    private void registerPorts() {
        try { addInputPort("input", new String());
        }catch(DuplicateIdException e){e.printStackTrace();}}
@Override
protected void doInternalTranssition(){}
@Override
protected void doExternalTransition(ArrayList<Port> eventBag) {
    color = (String) eventBag.remove(0).getValue();
    System.out.println(Utilities.getCurrentTime()+": "
    +this.getName().toUpperCase()+": received: "+ color );
    mapActivities();}
@Override
protected void doConfluentTransition(ArrayList<Port> eventBag){}
@Override
protected long computeLifeSpanFunction() {
    long lspan = Long.MAX_VALUE; return lspan;}
@Override
protected void mapOutputEvents(Object event) { }
@Override
protected void mapActivities() {
    System.out.println(Utilities.getCurrentTime()+": "+
    getName().toUpperCase()+":Displaying color:"+color+"\n");}
}
```

Figure 7: Display Unit of the Traffic Light System

Figure 8 shows the coupled system that has the control and display units as components. Being a coupled system, it *extends* the *CompositeSystem* class of the framework which provides the required methods to be completed. It basically creates and registers its components and establish any coupling(s) between them. In this case, there is only one coupling between the components as shown.

Figure 9 shows an excerpt from the result of the enactment of the specification. The first column of the result shows the wall clock time, the second column shows the identity of the component in context and the third column shows the event being reported. Note that each component has its *clock* that monitors its activities based on the ticks of the wall clock.

```java
package example;
import java.util.ArrayList;
import enactment.CompositeSystem;
import enactment.Port;
import enactment.designExceptions.*;

public class TrafficTest extends CompositeSystem {
    private TrafficLight lightControl;
    private Display displayUnit;

    public TrafficTest(String name) {
        super(name);
        lightControl = new TrafficLight("control");
        displayUnit = new Display("Display");
        registerComponents();
        doCouplings();
}
private void registerComponents() {
    try {
        addComponent(lightControl);
        addComponent(displayUnit);
    } catch (DuplicateIdException e){e.printStackTrace();}
}
@Override
protected void doCouplings() {
    try {//connectIC(source sys, source port, target Sys, target port)
        connectIC(lightControl, "statusOut", displayUnit, "input");
    } catch (InvalidCouplingException e) {e.printStackTrace();
    } catch (NoSuchPortExistsException e) {e.printStackTrace();}
}
```

Figure 8: Coupled Traffic Light System

With a starting state of "*READY*", the "*CONTROL*" received a time event at "15:31:37", sent an output as specified in the model and did an internal transition to assume the "*MOVING*" state. The output sent by "*CONTROL*" was received by the "*DISPLAY*" at the same time which concurrently changed its display color (activity) accordingly. Recall that the lifespan of the "*MOVING*" state is 10 milliseconds as specified in Figure 6, therefore, the next input event arrived at "15:31:47" and subsequent lines of the result segment can be read similarly.

CONCLUSIONS

We have presented an Object-Oriented framework for the enactment of DESs. The framework provides a template to guide the synthesis/writing of program codes from DEVS-based models and the protocol for real time enactment of system's behavior. The main idea is to be able to generate or specify an operational model in form of software systems to verify and validate the real time behavior of system models

```
Console
<terminated> TrafficEnact [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (13 Aug 2015 15:31:34)
15:31:37: CONTROL: TIME EVENT RECEIVED
15:31:37: CONTROL: Internal transition from READY
15:31:37: CONTROL: New state: MOVING

15:31:37: DISPLAY: INPUT EVENT RECEIVED
15:31:37: DISPLAY: received: GREEN
15:31:37: DISPLAY:Displaying color:GREEN

15:31:47: CONTROL: TIME EVENT RECEIVED
15:31:47: CONTROL: Internal transition from MOVING
15:31:47: CONTROL: New state: BRAKING

15:31:47: DISPLAY: INPUT EVENT RECEIVED
15:31:47: DISPLAY: received: YELLOW
15:31:47: DISPLAY:Displaying color:YELLOW

15:31:50: CONTROL: TIME EVENT RECEIVED
15:31:50: CONTROL: Internal transition from BRAKING
15:31:50: CONTROL: New state: STOPING

15:31:50: DISPLAY: INPUT EVENT RECEIVED
15:31:50: DISPLAY: received: RED
15:31:50: DISPLAY:Displaying color:RED

15:31:55: CONTROL: TIME EVENT RECEIVED
15:31:55: CONTROL: Internal transition from STOPING
15:31:55: CONTROL: New state: READY

15:31:55: DISPLAY: INPUT EVENT RECEIVED
15:31:55: DISPLAY: received: YELLOW
15:31:55: DISPLAY:Displaying color:YELLOW

15:31:58: CONTROL: TIME EVENT RECEIVED
15:31:58: CONTROL: Internal transition from READY
15:31:58: CONTROL: New state: MOVING

15:31:58: DISPLAY: INPUT EVENT RECEIVED
15:31:58: DISPLAY: received: GREEN
15:31:58: DISPLAY:Displaying color:GREEN
```

Figure 9: Enactment Traces of the Traffic Light System

using wall clock time as reference. i.e., the scheduling and execution of events are done based on the physical time.

We used the Object-Oriented Observer design pattern to express DEVS-based system constructs by mapping the system's structural and behavioral properties to the structure and semantics respectively of the observer pattern. The subject-observer relations are used to establish couplings between ports of the components of a system while the notification mechanisms are used to trigger state transitions. We provided a Java implementation of the framework and a

case study to illustrate its use to specify and enact discrete events systems.

In future research, we intend to provide a standard format for the traces of the execution so that it can be amenable to further rigorous analysis. We also intend to integrate the framework with compatible model-based development environments to extend the kinds of analysis they provide. With appropriate applications of model-driven technologies, we can synthesize enactment codes based on the template provided from any suitable DEVS-based model.

REFERENCES

Baldassari, M. and Bruno, G. 1991. PROTOB: An object oriented methodology for developing discrete event dynamic systems. In *High-Level Petri Nets* (pp. 624-648). Springer Berlin Heidelberg.

Baldassari, M., Bruno, G., Russi, V. and Zompi, R. 1989. PROTOB a hierarchical object-oriented CASE tool for distributed systems. In *ESEC'89* (pp. 424-445). Springer Berlin Heidelberg.

Bruno, G. and Agarwal, R. 1995. Validating software requirements using operational models. In *Objective Software Quality* (pp. 78-93). Springer Berlin Heidelberg.

Chow, A. C. H., and Zeigler, B. P. 1994. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th conference on Winter simulation* (pp. 716-722). Society for Computer Simulation International.

Dowson, M. and Fernström, C. 1994. Towards requirements for enactment mechanisms. In *Software Process Technology* (pp. 90-106). Springer Berlin Heidelberg.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1994. *Design patterns: elements of reusable object-oriented software*. Pearson Education.

Holmlid, S. and Evenson, S. 2007. Prototyping and enacting services: Lessons learned from human-centered methods. In *Proceedings from the 10th Quality in Services conference, QUIS* (Vol. 10).

Holmquist, L. E. 2005. Prototyping: Generating ideas or cargo cult designs?.*Interactions*, *12*(2), 48-54.

Hu, X. and Zeigler, B. P. 2002. *An integrated modeling and simulation methodology for intelligent systems design and testing*. ARIZONA UNIV TUCSON.

Hu, X. and Zeigler, B. P. 2004. Model continuity to support software development for distributed robotic systems: A team formation example. *Journal of Intelligent and Robotic Systems*, *39*(1), 71-87.

Jeston, J. and Nelis, J. 2014. *Business process management*. Routledge.

Kouvas, G., Grefen, P., and Juan, A. 2010. Business Process Enactment. In *Dynamic Business Process Formation for Instant Virtual Enterprises* (pp. 113-132). Springer London.

Krasner, G. E. and Pope, S. T. 1988. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, *1*(3), 26-49.

Ottensooser, A. and Fekete, A. 2007. An Enactment-Engine Based on Use-Cases. *Business Process Management*, 230-245.

Raje, R. R., Williams, J. I. and Boyles, M. 1997. Asynchronous remote method invocation (ARMI) mechanism for Java. *Concurrency - Practice and Experience*, *9*(11), 1207-1211.

Van Der Aalst, W. M., Ter Hofstede, A. H. and Weske, M. 2003. Business process management: A survey. In *Business process management* (pp. 1-12). Springer Berlin Heidelberg.

Vangheluwe, H. L. 2000. DEVS as a common denominator for multi-formalism hybrid systems modelling. In *Computer-Aided Control System Design, 2000.* (pp. 129-134). IEEE.

Zeigler, B. P. 1976. Theory of modelling and simulation. Wiley.

Zeigler, B. P., Praehofer, H. and Kim, T. G. 2000. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. Academic press.