AGENT METHODOLOGICAL LAYERS IN REPAST SIMPHONY

Franco Cicirelli, Angelo Furfaro, Libero Nigro, Francesco Pupo
Laboratorio di Ingegneria del Software
Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria
87036 Rende (CS) – Italy
Email: {f.cicirelli,a.furfaro}@deis.unical.it, {l.nigro,f.pupo}@unical.it

KEYWORDS
Agent-based modeling and simulation, Repast Simphony, Java, actors, P–DEVS, Petri nets.

ABSTRACT

Repast Simphony (RS) is a popular toolbox for agent-based modeling and simulation (ABMS) of complex systems. It can be used from within the Eclipse IDE with Java being the main implementation language. Moreover, visual modeling is supported by agent flowcharts. Powerful features of RS include contexts and projections which allow the modeler to build e.g. situated multi-agent systems (MAS) which can easily be configured and visualized in the RS runtime system. RS lacks of a reference agent methodology. Rather the modeler is free to define and follow her/his own methodology with RS: procedurally, declaratively or visual-based. This openness was exploited in this work for supporting different notions of agents, thus addressing the modeling needs of various application domains. In particular this paper proposes an embed in RS of an actor model which provides a lightweight notion of agents. The actor model is then used as a kernel for supporting more abstract but rigorous modeling languages like Parallel DEVS (P–DEVS) and time-extended Petri nets. A P–DEVS modeling example is reported to demonstrate the usefulness of supporting multiple agent methodological layers in RS.

INTRODUCTION

Agent-based modeling and simulation (ABMS) is currently recognized (North & Macal, 2007)(Macal & North, 2011) as a fundamental tool for predicting the behavior of complex systems in such diverse fields as financial, social, biological and engineering domains. The strength of ABMS stems from its ability to mimics a reality by means of domain specific components/agents interacting and coordinating to one another and to organize agents on the basis of their behaviors. Nowadays ABMS can be practiced by using several modeling languages and tools. Repast Simphony (RS) is a leading free and open-source ABMS toolkit which is integrated with Eclipse IDE. It is the latest member of the REcursive Porous Agent Simulation Toolkit family which is mainly based on Java. RS (North *et al*., 2005a-b), though, adds visual point-and-click tools for designing agents, specifying agent behaviors, and for executing, collecting and inspecting simulation results using a variety of external analysis tools. Contexts and projections are a key for organizing agents over spatial structures (e.g. toroidal grids and networks) which have a connection to display and visualization means of the RS runtime system. A basic feature of RS is the absence of a reference agent methodology. For instance, the modeler can introduce agents as plain-old-java-objects (POJO) whose methods can be annotated with declarative scheduling (scheduled methods or watch annotations). In the general case, method scheduling has to be explicitly achieved by submitting actions e.g. to the default scheduler, at an arbitrary dense time.

In the work described in this paper, RS openness is exploited to experiment with different agent based modeling methodologies, tailored to the needs of specific problem domains. In particular, as a basic agent kernel, a lightweight actor model (Cicirelli *et al*, 2009) was adapted to work with RS. Actors encapsulate a data status and interact to one another by asynchronous message-passing. Actor behaviors are realized as finite state machines, possibly patterned as distilled statecharts (Cicirelli *et al*., 2011a). The actor model has proved to be very flexible and suitable for scalable complex agent-based simulations (Cicirelli *et al*., 2009). It was also used as the execution platform for distributing RepastJ models over HLA for high-performance conservative parallel simulations (Cicirelli *et al*., 2011b).

The original contribution of authors current work is to use RS enriched by actors so as to provide a clean agent methodology integrated with agent spaces and visualization means of RS. A benefit of the actor framework is its ability to support higher agent layers e.g. enabling more abstract but rigorous agent modeling. In particular a Parallel DEVS (Zeigler *et al*., 2000)(Zeigler & Sarjoughian, 2005) layer was achieved and on top of it a Time Petri Net layer (Merlin & Farber, 1976) and a Time Stream Petri Net layer (Diaz & Senac, 1994)(Cicirelli *et al*., 2013) enabling real-time, multimedia and workflow modeling and analysis. Other layers can be added.

This paper is structured as follows. First the basic actor model is proposed and embedded in Repast Simphony, which is suited for general ABMS of large and scalable systems. Then a P–DEVS modeling layer is described which was achieved on top of actors. After that a modeling example is presented and simulated using

Proceedings 27th European Conference on Modelling and Simulation ©ECMS Webjørn Rekdalsbakken, Robin T. Bye, Houxiang Zhang (Editors)
ISBN: 978-0-9564944-6-7 / ISBN: 978-0-9564944-7-4 (CD)

P–DEVS, exploiting features provided by Repast Simphony. The possibility of customizing P–DEVS modeling to cope with conflict management e.g. in time-extended Petri nets, is then discussed. Finally, conclusions are drawn with an indication of on-going work.

Actors - A Basic Agent Layer for Repast Simphony

Actors are lightweight agents which *encapsulate* a *data status* and have a *behavior* for responding to messages belonging to a given *message interface*. The communication model is based on *asynchronous message-passing*. Actor behavior is modeled by a finite state machine. An actor is a *reactive object* which responds to an incoming message on the basis of its current state and message contents. Message reception is implicit. An actor is at rest until a message arrives. Message processing triggers a state transition and the execution of a not pre-emptable atomic action. Messages can be unexpected in the current state. Unexpected messages can be discarded or their processing postponed by remembering them in local data or in states of the life cycle. Basic operations on actors are:

- *new*, for creating a new actor
- *become*, for changing the actor state
- (non blocking) *send* for transmitting messages to *acquaintance* actors (including itself for proactive behavior). The send operation carries a message and an absolute timestamp at which the message has to be heard. When absent, the timestamp evaluates to current time (now)
- *now*, which returns the time notion to actors. The now operation is actually provided by a control machine (see below).

The evolution of a subsystem of actors running on a single processor is regulated by a *control machine* component, which is in charge of (transparently) scheduling sent messages, applying to them an application dependent control structure (e.g. based on discrete-event simulation) and dispatching them to actors e.g. according to time. Actors are thread-less. Only one thread exists in the control machine which supports the interleaved execution of actors. The control machine repeats a basic loop where at each iteration one message is selected in the set of pending messages, and delivered to its target actor by invoking on it the *handler* method with the message as an argument. Messages are the unit of scheduling.

The above picture of the actor model corresponds to a single execution node of the more general Theatre distributed architecture (Cicirelli *et al.*, 2009) which is beyond the scope of this paper.

The actor framework is implemented in Java through the services of a few base abstract classes (see also Fig. 2): Actor, Message and ControlMachine. Programmer defined actor classes must derive, directly or indirectly, from Actor. Application message classes must inherit from Message. The Message class includes such fields (and related get/set methods) as *receiver*, *timestamp*, and *removed* flag. The time model is assumed to be dense. The removed flag is used to invalidate a scheduled message so that it will no longer be dispatched. It is worth noting that the flag avoids physically dropping the message from the message queue used by the control machine.

Mapping actors onto Repast Simphony (RS) was achieved by developing a RepastS_Simulation class which is a specialization of ControlMachine (see Fig. 2) and by adopting the Java based modeling style enabled by RS. RepastS_Simulation rests on the RS default scheduler and customizes basic schedule/unscheduled methods of the ControlMachine by interfacing with the RS scheduler. An actor message is wrapped into an RS action. Action scheduling specifies (i) the message receiver as the target agent, (ii) the message timestamp as the action occurrence tick, and (iii) the handler method on the receiver actor which has to be invoked, at action dispatch time, with the message as an argument.

To complete preparation of a Java based ABMS model based on actors and RS, a context builder class (see pseudo code in Fig. 1) has to be prepared equipped of *build*, *initialize* and *finalize* methods. The build method is responsible of instantiating the RepastS_Simulation control machine, getting any simulation parameters (including the simulation time limit) entered by the user through the RS runtime system, and to create actors and adding them to the root main context of RS. The actual initialization of actors is realized in the initialize method, whose execution at time 0 is scheduled in the build method. The initialize method also schedules the execution of the finalize method at the end time of the simulation.

```
public class ModelCxtBuilder extends
    ContexBuilder<Object>{
    common model data declaration
    public Context<Object>
        build( Context<Object> context ){
        //to properly enable the use of projections
        set the id of (root) context
        achieve default scheduler of Repast Simphony
        get simulation parameters, if there are any, from the
            runtime system
        create a RepastS_Simulation instance with the
            simulation time limit as an argument
        create actor instances and add them to context
        add actors to context
        schedule execution of the initialize method at time 0
        return context
    }//build
    void initialize(){
        initialize actors by sending first messages
        schedule execution of the finalize method at the time
            limit of simulation
    }//initialize
    void finalize(){
        execute model finalization actions
    }//finalize
}//ModelCxtBuilder
```

Figure 1 – Schema of a model context builder based on actors

The resultant actor framework is effective and provides to the RS modeler the ability to follow a simple yet powerful and general agent methodology for building complex and scalable ABMS models which can exploit contexts and projections (spaces) and the services exposed by the RS runtime system, i.e. parameter setting, simulation batches with parameter sweep, collection of simulation results and their graphical rendering for analysis and so forth. Besides modeling at the actor/agent layer, more abstract methodological layers can be built and used for ABMS.

A PARALLEL DEVS METHODOLOGICAL LAYER

A Parallel DEVS (P–DEVS) (Zeigler *et al.*, 2000)(Zeigler & Sarjoughian, 2005)(Cicirelli *et al.*, 2007)(Cicirelli *et al.*, 2013) implementation based on actors was achieved, which enables Repast Simphony to be used also according to formal P–DEVS modeling.

An overview to Parallel DEVS

A model is a coupled component which consists of an interconnection of atomic (or behavioral) and coupled (or structural) components, and so forth recursively. As a consequence, a P–DEVS model has a natural hierarchically structure. An atomic component is a structure M defined as $M=<X,S,Y,\delta_{int},\delta_{ext},\delta_{con},\lambda,ta>$ where

- X is the set of input values
- S is a set of states
- Y is the set of output values
- $\delta_{int}:S\rightarrow S$ is the *internal transition* function
- $\delta_{ext}:Q\times X^b\rightarrow S$ is the *external transition* function, where $Q=\{(s,e)|s\in S,\ 0\leq e\leq ta(s)\}$ is the set of *total states* and e is the *elapsed time* since last transition
- X^b denotes the collection of *bags* over X (in a bag some elements may occur more than once)
- $\delta_{con}:Q\times X^b\rightarrow S$ is the *confluent transition* function
- $\lambda:S\rightarrow Y^b$ is the *output function*
- $ta:S\rightarrow R^+_{0,\infty}$ is the time *advance function.*

Sets X, S and Y are typically products of other sets. S, in particular, is normally the product of a set of *control states* (or *phases*) and other sets built over the values of a certain number of variables used to describe the system at hand.
Meaning of the elements of M can be stated as follows. At any time the system is in some state $s\in S$. The system can stay in s for the time duration (dwell-time) $ta(s)$. ta(s) can be 0, in which case s is said a *transitory state*, or it can be ∞, in which case it is said a *passive state* because the system can remain forever in s if no external event interrupts. Provided no external event arrives, at the end of (supposed finite) time value $ta(s)$, the system moves to its next state $s'=\delta_{int}(s)$ established by the internal transition function δ_{int}. Moreover, just *before* making the internal transition, the system produces the output computed by the output function $\lambda(s)$. During its remaining in s, the system can receive an external event x which can cause s to be exited earlier than $ta(s)$. Let $e\leq ta(s)$ be the elapsed time since the entering time in s (or, equivalently, the time of last transition). The system then exits the state s moving to the next state $s'=\delta_{ext}(s,e,x)$ determined by the external transition function δ_{ext}. As a particular case, the external event x can arrive when $e=ta(s)$. In this case two events occur simultaneously: the internal transition event and the external transition event. The next state s', in this *collision* situation, is determined by the confluent transition function δ_{con}. Default behavior of δ_{con} consists of executing the output function and then the external transition function on the phase determined by application of the internal transition (see Fig. 3). This behavior can be redefined to comply with the application needs. After entering state s', the new time advance value $ta(s')$ is computed and the same story continues. Following a self-loop external transition, i.e. where $s'=s$, there is no need to stop time advancement in current state.
It is worth noting that there is no way to generate an output directly from an external transition. An output can only occur just before an internal transition. To have an external transition cause an output without a delay, a transitory state can be entered from which the exiting internal transition is preceded by the generation of the output value.
Parallel DEVS emphasizes that a *bag* of simultaneous inputs can be received by an atomic component which in general can have a specific reaction to the combination of inputs which is different from the effect of sequential reactions to the separately received inputs.
In practice, an atomic component receives its inputs from *typed input ports* and generates outputs through *typed output ports*. Ports are architectural elements which favor a modular system design. A component refers only to its interface ports. It has no knowledge about the identity of cooperating partners. A coupled component (subnet) is a port interconnection of existing atomic or coupled (hierarchical) components.
Each component can be equipped with its own simulator. In this work, though, a single simulation structure is used which serves an entire P–DEVS flattened model where it is minimized the number of exchanged messages.

Mapping P–DEVS onto RS actors

In the following, an original implementation of P–DEVS which builds on previous authors work, e.g. (Cicirelli *et al.*, 2007), and complies specifically with Repast Simphony, is presented. The organization is portrayed in the UML class diagram of Fig. 2.
AtomicDEVS abstract class, derived from Actor, is the base for achieving, through inheritance, the concrete atomic components required by an application.

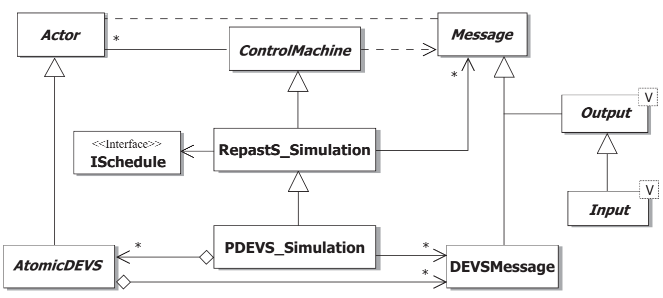

Figure 2 – UML class diagram supporting P–DEVS on top of actors in Repast Simphony

```
public abstract int delta_int( int phase );
public abstract int delta_ext( int phase,
                               double e, Iterable<Message> x );
public int delta_con( int phase, double e,
                               Iterable<Message> x ){
    //default behavior
    lambda( phase );
    return delta_ext( delta_int( phase ), 0, x );
}//delta_con
public abstract double ta( int phase );
public abstract void lambda( int phase );
public void handler( Message m );
public void handler( Iterable<DEVSMessage> bag );
public void initialPhase( int phase ); //sets the initial phase of
                                       //the component
```
Figure 3 - AtomicDEVS method interface

Component internal phases are coded as integers. Method signatures in AtomicDEVS are shown in Fig. 3. Typed input/output ports of components are directly mapped on to messages. In particular the Output<V> and Input<V> derived classes of Message are introduced which are generic in the type V of the carried data. In particular, Input<V> is derived from Output<V>. Services get()/set() permit respectively to achieve/modify the data component of an Output<V> message. The method linkTo(receiver) of DEVSMessage allows an output message to be bound to a given receiver actor.

PDEVS_Simulation specializes RepastS_Simulation by taking into account the specific semantics of P–DEVS models and constraints introduced by RS. An evolution of a P–DEVS model can be viewed as a sequence of *epochs* where all the actions (simultaneous events) to be accomplished at a same time are carried out in due order. Toward this timed and untimed messages are handled.

Timed messages, scheduled by RS, correspond to internal events of atomic components which have exhausted their dwell-time in current phase. Untimed messages are managed by PDEVS_Simulation and are grouped into bags to be received as external messages in the current epoch. As required by P–DEVS, all instantaneous messages created at current time must be collected into bags so as for them to be processed by relevant recipients before starting the next epoch.

Therefore, it is the responsibility of PDEVS_Simulation to ensure first all contemporary timed messages (and corresponding internal transitions) are processed and, finally, all the collected bags of messages are sent to target components. When all bags existing in current time have been processed, the next epoch is entered.

Since contemporary timed messages and bags have the same occurrence time (i.e. they are simultaneous events), it is mandatory to order messages in such a way that timed message processing precedes bags processing. More precisely, PDESV_Simulation maintains the collection of the atomic devs components which have bags at current time. Dispatch of bags to these components is accomplished by a dispatchBags method of PDEVS_Simulation. A single action for executing dispatchBags is scheduled at current time to occur at the *end* of present epoch. To achieve the required ordering, the priority field of RS actions is exploited. The scheduled action for dispatchBags gets a lower priority than that used for scheduling timed messages.

Moreover, the AtomicDEVS introduces two versions of the handler method (see Fig. 3). The first one overrides the handler method of the Actor base class. It receives a Message as an argument and executes the output and internal transition functions of the component. The second version of handler has a bag as parameter, modeled as an Iterable<DEVSMessage> object. This version of the handler method executes actions of the external transition of the component. The basic version of handler is also capable of discovering a collision situation which occurs when the timed message finds a non empty bag destined to the component. In these cases the confluent function of the component is invoked instead of output and internal transition functions.

From the above discussion it follows that the modeler has to concentrate only to the specification of the basic functions of atomic components, namely the time advance, lambda (output), the delta_int (internal transition), delta_ext (external transition) and delta_con (confluence transition) functions. All of this delivers a declarative modeling style, with all the scheduling burden being insulated in the PDEVS_Simulation and the AtomicDEVS classes.

A Modelling Example based on Parallel DEVS

Fig. 4 shows a coupled model inspired by (Zeigler & Sarjoughian, 2002). The example concerns a simple neural network where components (neurons) are interconnected by input/output ports. The network models a direct graph e.g. of cities, with assigned distances between adjacent cities. The problem is to find the shortest path between a Start and a Dest pair of cities.

Nodes of the graph are instances of the FireOnceNeuron (FON) and PulseGenerator atomic components. The PulseGenerator is in charge of generating pulses which are then simultaneously broadcast to A, B, I and C components.

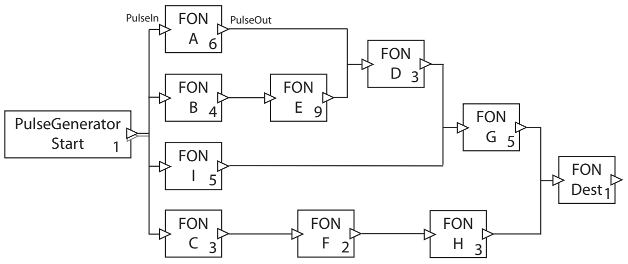

Figure 4 – A P–DEVS neural network for computing the shortest path in a direct graph

In Fig. 4 the firing delay of neurons (shown in the component box) expresses the weight (e.g. travelling time) of the arcs. Each atomic component is supposed to be equipped by a single input and a single output port carrying pulses. The final component Dest has its output port unconnected. The PulseGenerator atomic component has an array of output ports.

The FireOnceNeuron model (see Fig. 5) remains in the RECEPTIVE phase until one or a bag of pulse external events arrive. The external transition function moves it to the FIRE state where the component remains for a duration given by firingDelay>0, after which it emits a pulse and enters the REFRACT state. The time advance of REFRACT is infinity. Therefore, the neuron after firing moves and remains in REFRACT forever, never being able to fire again. Further pulses received when staying in FIRE or REFRACT are simply ignored. In this example a bag of input pulses has the effect corresponding to any single pulse of the bag.

In Fig. 5, thick arrows denote external transitions, thin arrows internal transitions, dashed arrows output generation. The FireOnceNeuron Java class directly corresponds to the P–DEVS formal definition of the component. The only extra code added is related to placing the component over the adopted grid space. The model *linkto* relationships are also established explicitly as network links.

Table 1 depicts the formal P–DEVS definition of the FireOnceNeuron. The output function has been explicitly defined only for states with a finite time advance value. For RECEPTIVE and REFRACT states the output function implicitly associates the null output. From the property of FireOnceNeuron to propagate only the first received pulse, which necessarily is the faster one, derives the ability of the overall network to naturally computing the shortest path. It is worth noting that the model of Fig. 5 adopts the default confluent transition function.

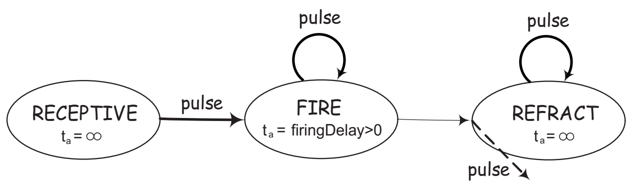

Figure 5 - Control states and transitions of the FireOnceNeuron atomic component

Table 1. P–DEVS specification of FireOnceNeuron

X={pulse}
S={RECEPTIVE, FIRE, REFRACT}
Y={pulse}
δ_{int}(FIRE)=REFRACT
δ_{ext}(RECEPTIVE,e, pulse^b)=FIRE
δ_{ext}(FIRE,e, pulse^b)=FIRE
δ_{ext}(REFRACT,e, pulse^b)=REFRACT
λ(FIRE)={pulse}
ta(RECEPTIVE)=∞
ta(FIRE)=firingDelay
ta(REFRACT)=∞

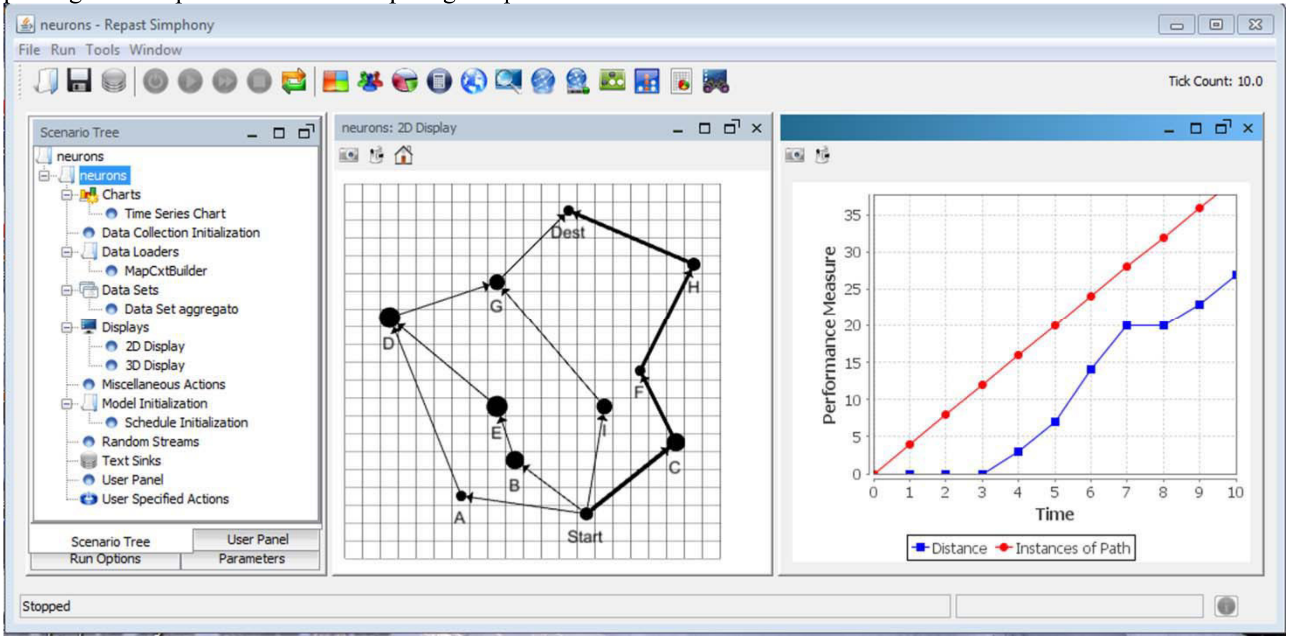

Figure 6 – A screenshot of the RS runtime environment relevant to the execution of the neurons model

The neural network model was configured in a MapCxtBuilder class which creates the component instances and links them according to the topology portrayed in Fig. 4. Components are situated on a grid projection and their inter-relationships expressed by a network projection.

Four instances of a class Path are injected, periodically, by the PulseGenerator as values on to its output ports and flow through the network. A Path object collects the id of crossed component nodes along the current travelled distance. As a consequence the path object which arrives to the Dest component will contain the shortest path.

Fig. 6 shows model display in the RS runtime environment. As one can see the shortest path gets highlighted in the 2D display panel. For demonstration purposes, a time series chart was also generated which reports the number of path objects injected, from time to time, in the network by the pulse generated and the cumulative distance, i.e. the sum of distances travelled by path objects at each time. The latter can be viewed as a sort of cost tied to the algorithm. It worth noting that generation of the chart time series is an orthogonal aspect of the model, i.e. no information has been put in the modeled system about the chart. The simulation stops as soon as the faster path object reaches the Dest node. As a consequence, the last time point in the x axes of the chart also mirrors the length of the found shortest path.

Time Extended Petri Nets Layers

The P−DEVS framework described in the previous section was successfully used as a starting point for supporting Time Petri Nets (TPN) (Merlin & Farber, 1976) and Time Stream Petri Nets (TSPN)(Diaz & Senac, 1994) which are useful for modeling and analysis of time-dependent systems, e.g. embedded real-time systems with time-constraints, multimedia/ hypermedia systems, workflow systems (Cicirelli *et al.*, 2013). TSPNs associate a dense time interval to input arcs only, whereas TPNs specify a dense time interval to transitions only, which constrains transition firing. In a TSPN model one out of ten firing or synchronization rules can be attached to a transition. Depending on the firing rule, the transition is then able to determine, at each time it becomes enabled, a dynamic time interval (possibly empty) which affects its firing. In common with TPN transitions, a *strong firing model* is ensured: a TSPN transition can only fire at any time of its time interval, provided it remains continually enabled.

Every TPN/TSPN transition is modeled as a distinct P−DEVS atomic model.

Proper TPN/TSPN modelling requires the handling of a problem not covered by standard P−DEVS: that of *conflicts* arising, in a component, at the last time of stay in its current phase (Cicirelli *et al.*, 2007). Whereas P−DEVS normally relies on the *maximal parallelism* hypothesis which implies that all timed messages occurring at a same epoch are always committed and there is no way for an internal event to forbid the processing of a different timed message of the same epoch, in TPN/TSPN and similar models, the firing of a transition, although occurring at the last time point of its dynamic time interval, can be disabled by the firing of a conflicting transition.

Conflict management can be achieved, in these models, through a redefinition of the confluent function so as to coincide with the external transition function, thus permitting the removal of some pending timed messages of current epoch, and the discard of the corresponding lambda/output function which would send messages to partner components. The mentioned redefinition allows a transition to be informed at any time of the firing of a conflicting transition so as to adjust its state accordingly.

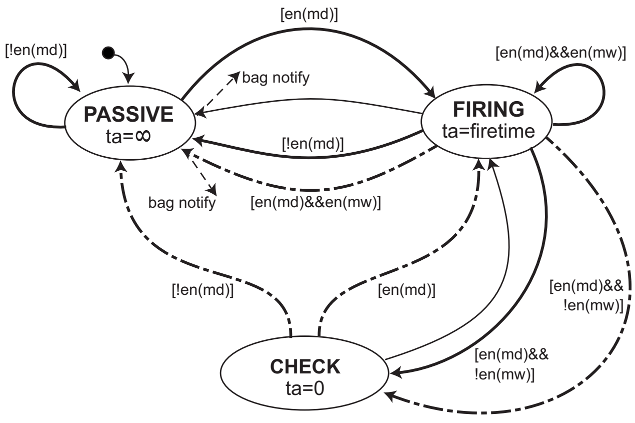

Figure 7 – P−DEVS model of a TSPN transition

Fig. 7 portrays the dynamic behaviour of a TSPN transition (a similar model exists for a TPN transition). Thick arrows represent external transitions triggered by a bag of Notify() messages. Arrows which are both tick and dashed denote confluent transitions. Before accomplishing an internal transition (thin arrow), the lambda() function (thin dashed arrows in Fig. 7) is executed which generates output events to partner components. More in particular, the lambda() function of a fired transition creates a bag of Notify() messages as a consequence of changing the marking of places during its firing process. When coming back from CHECK to FIRING due to the elapsed time internal event, the lambda() function is simply void.

A bag of messages is received by the external/confluent function of a Transition component, as an Iterable object. The en() predicate returns true if the transition is logically enabled, i.e., a sufficient number of tokens is available in the preset. As part of the en() service, the dynamic firing interval of the transition is calculated, on the basis of the enabledness and temporal interval of the input arcs and the synchronization rule selected for the transition.

The atomic firing process of a transition *t* occurs in two steps: first tokens are withdrawn from the preset of *t*, then tokens are deposited in the postset of *t*. A critical point is concerned with the detection of under firing transitions which lose their enabling during the first withdrawl step of the firing process of *t*, but are anyway enabled at the end of the firing process. Such transitions must correctly be handled as newly enabled transitions.

In order to realize the firing process, different notify messages are employed to separately carry out a withdraw marking and a deposit marking relevant to a given transition.

In Fig. 7, the arrival of a bag of Notify() messages in the FIRING phase, causes a self-loop external transition in the case the transition is persistent to current firing, i.e. it is enabled both during and at the end of the firing. As soos as a transition detects it loses its enabling in a withdrawal step, but it is enabled in the deposit phase, it switches from FIRING to the CHECK transitory phase. From CHECK the transition can re-enter the FIRING phase if the transition finds itself ultimately enabled in the deposit step, otherwise it reaches the PASSIVE phase. It should be noted that due to interleaving of concurrent actions occurring at a same time, it is perfectly possible that an under CHECK transition can lose its enabling for the firing process of another conflicting transition.

As a final remark, it is worthy of note that a time violated TSPN transition, i.e. one which has no valid dynamic firing interval, remains in the FIRING phase until it gets disabled, without the possibility of completing its firing.

A TSPN model is configured by first creating the (passive) place objects and the transition components. For each transition the synchronization rule is furnished. Places are then added to the preset/postset of transitions. After that input arc connections among places and transitions and vice versa are established. Model bootstrap is finally ensured by defining the initial marking of each place which in turn generates the initial bags for transitions.

The TSPN modeling layer was exploited, in a significant case, in the modeling and simulation of complex workflow systems (Cicirelli *et al.*, 2013).

CONCLUSIONS

The Repast Simphony agent methodological layers described in this paper are a key for promoting agent modeling eclecticism, that is the possibility of choosing the agent modeling layer most apt to a given problem domain. The realizations are founded on a lightweight actor model which provides asynchronous message passing and control specific strategies of message scheduling/dispatching.

On-going work is geared at

- improving and optimizing the realizations by applying them to ABMS of large systems
- exploring visual modeling with some agent layer, e.g. P–DEVS which is almost declarative in character, in the sense that an atomic component rests only on local variables and the specification of basic internal transition, external transition, confluent transition, time advance and lambda/output functions
- experimenting with time-extended Petri nets layers in the modeling and analysis of real-time tasking sets
- extending the approach by addressing other agent layers as well.

REFERENCES

Cicirelli, F., A. Furfaro, L. Nigro (2007). Conflict management in PDEVS: An experience in modeling and simulation of time Petri nets. In *Proc. of Summer Computer Simulation Conference (SCSC'07).*

Cicirelli, F., A. Furfaro, L. Nigro (2009). An Agent Infrastructure over HLA for distributed simulation of reconfigurable systems and its application to UAV coordination. *Simulation Trans. of The Society for Modelling and Simulation International*, **85**(1):17-32.

Cicirelli, F., A. Furfaro, L. Nigro (2011a). Modelling and simulation of complex manufacturing systems using statechart-based actors. *Simulation Modelling Practice and Theory*, **19**(2):685-703.

Cicirelli, F., A. Furfaro, A. Giordano, L. Nigro (2011b). HLA_ACTOR_REPAST: An approach to distributing Repast models for high-performance simulations. *Simulation Modelling Practice and Theory*, **19**(1):283-300.

Cicirelli, F., A. Furfaro, L. Nigro (2013). Using time stream Petri nets for workflow modeling analysis and enactment. *Simulation Trans. of The Society for Modelling and Simulation International*, **89**(1): 68-86.

Diaz, M., P. Senac (1994). Time stream Petri nets: A model for timed multimedia information. In *Proc. of the 15th Int. Conf. on Application and Theory of Petri Nets*, London, UK, pp. 219–238.

Macal, C.M., M.J. North (2011). Introductory tutorial: agent-based modeling and simulation. In *Proc. of Winter Simulation Conference*, pp. 1456-1469.

Merlin, P., D. Farber (1976). Recoverability of communication protocols–implications of a theoretical study. *IEEE Trans. on Comm.* **24**: 1036–1043.

North, M.J., T.R. Howe, N.T. Collier, J.R. Vos (2005a). Repast Simphony development environment. In *Proc. of the Agent 2005 Conf. on Generative Social Processes, Models and Mechanisms*, ANL/DIS-06-01, Oct.

North, M.J., T.R. Howe, N.T. Collier, J.R. Vos (2005b). Repast Simphony runtime system. In *Proc. of the Agent 2005 Conf. on Generative Social Processes, Models and Mechanisms*, ANL/DIS-06-01, Oct.

North, M.J. & C.M. Macal (2007). *Managing business complexity – Discovering solutions with Agent-Based Modeling and Simulation*, Oxford University Press.

Repast, on-line, http://repast.sourceforge.net

Zeigler, B.P., H. Praehofer, T.G. Kim (2000). *Theory of modeling and simulation*. Second Edition, Academic Press.

Zeigler, B.P. and H.S. Sarjoughian (2002). DEVS component-based M&S framework: an introduction. *http://www.acims.arizona/EDUCATION/ECE575Fall03/ECE575Fall03.html.*

Zeigler, B.P. and H.S. Sarjoughian (2005). Introduction to DEVS modeling and simulation with Java: developing component-based simulation models. *http://www.acims.arizona.edu.*