Conflict Management in PDEVS: An Experience in Modelling and Simulation of Time Petri Nets

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

Keywords
PDEVS, modelling and simulation, conflict management, time Petri nets, Java.

Abstract

PDEVS (Parallel DEVS) is a well-known formalism for the specification of complex concurrent systems organized as an interconnection of atomic and coupled interacting components. The abstract simulator of a PDEVS model is normally founded on the assumption of maximal parallelism: multiple components are allowed to undertake at the same time an independent state transition. This paper argues that the hypothesis of maximal parallelism does not allow PDEVS to adequately model and simulate systems where simultaneous state transitions are conflicting to one another. As an example, an original PDEVS model of Merlin and Farber Time Petri Nets is proposed. The realization owes to ActorDEVS, a lean and efficient PDEVS M&S framework in Java, which enables the simulation control structure to be customized. The accomplished experience suggests that some points in the formal definition of PDEVS should possibly be adapted in order to widen the applicability of the language.

1. INTRODUCTION

Parallel DEVS (Zeigler *et al.*, 2000) –PDEVS– is a well-known and accepted formalism for the specification of complex discrete or continuous systems abstracted as a network of concurrent, timed and interacting atomic or coupled models. The formalism borrows from systems theory and enables a seamless transformation of a specification into an executable system, e.g. using an object-oriented modelling and simulation framework directly based on software architecture concepts (Shaw & Garlan, 1996)(Zeigler & Sarjoughian, 2003).

PDEVS defines a simulation architecture which associates a simulator with each atomic component. Moreover, in a coupled model a coordinator is used which orchestrates the behaviour of the various internal simulators in order to guarantee coherent time advancement. The simulation architecture makes it possible for multiple components to undergo simultaneous state transitions. In other words, PDEVS naturally relies on the hypothesis of maximal parallelism, thus exploiting the intrinsic degree of concurrency which comes with many models.

PDEVS expressive power has been used as a common denominator for supporting other formalisms (Zeigler *et al.*, 2000)(Vangheluwe, 2000). This work argues, however, that the definition of PDEVS does not allow for proper management of conflicts which may arise in systems modelling. As an example, this paper describes an original mapping of the Time Petri Nets (TPNs) (Merlin & Farber, 1976)(Berthomieu & Diaz, 1991) formalism often used for communication protocols or embedded real-time systems modelling, onto PDEVS. The experience was concretely carried out using ActorDEVS (Cicirelli *et al.*, 2006), a minimal flexible and efficient agent-based M&S framework developed in Java, which makes it possible to specialize the event-driven simulation engine. The experience indicates that some points of PDEVS should possibly be adapted in order to widen the applicability of the formalism.

The paper is structured as follows. Section 2 reviews basic concepts of PDEVS. Section 3 discusses aspects of PDEVS which relate to conflict management. Section 4 summarizes concepts of Time Petri Nets together with a model example. Section 5 introduces ActorDEVS. Section 6 describes the proposed mapping of TPNs onto ActorDEVS. Finally, conclusions are presented along with an indication of on-going and future work.

2. PDEVS CONCEPTS

2.1 Atomic components

A PDEVS 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*
  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.*

The sets X, S and Y are typically products of other sets. S, in particular, is normally the product of a set of *control states* (said also *phases*) and other sets built over the values of a certain number of variables used to describe the system at hand.

At any time the component is in some state $s\in S$. The component can remain 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 component 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 component moves to its next state $s'=\delta_{int}(s)$ determined by the internal transition function δ_{int}. In addition, just *before* making the internal transition, the component produces the output computed by the output function $\lambda(s)$. During its stay in s, the component 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 enter time in *s* (or, equivalently, the time of last transition). The component then exits state *s* moving to 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}, after having applied the output function. The default behaviour of δ_{con} first applies the internal transition function and then the external transition function. This behaviour, though, can be redefined.

After entering state *s'*, the new time advance value *ta(s')* is computed and the same story continues.

It is worth noting that "*there is no way to generate an output directly from an external transition*" (see page 12 in (Zeigler & Sarjoughian, 2003)). 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 output generation.

PDEVS emphasizes that a *bag* of simultaneous inputs can be received by the atomic component which in general can have a reaction to the combination of inputs which is different from the effect of sequential reactions to the separately received inputs.

The above description does not mention the component interface ports. In practice, inputs are effectively received from corresponding typed *input ports* and similarly, outputs are generated through typed *output ports*. Actually *X* is a set of pairs *<inp,v>* where *inp* is an input port and *v* the type of values which can flow through *inp*. Y is a set of pairs *<outp,v>* where *outp* is an output port. Ports are architectural elements which favour modular system (re)configuration. A component refers only to its interface ports. It has no knowledge about the identity of cooperating partners which can potentially be changed at runtime.

2.2 Coupled components

A coupled component (subnet) is an interconnection of existing atomic or coupled components (see Fig. 1). Formally, it is a structure *N* defined as $N=(X,Y,D,\{M_d|d \in D\},EIC,EOC,IC\}$, where

- *X* and *Y* are input and output sets of the coupled component
- *D* is a set of (sub) component identifiers (or names)
- *M* is a set of (sub) PDEVS components whose interconnection gives rise to the coupled model
- *EIC* is the external to internal coupling function
- *EOC* is the internal to external coupling function
- *IC* is the internal to internal coupling function.

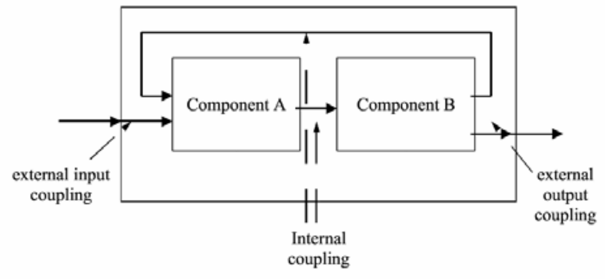

Fig. 1. Schema of a coupled model

Coupling is actually based on port interconnections. An external input coupling, for instance, is a binding from an input port of the coupled model to a matching input port of an internal component, and so forth. In the case an internal component is itself a coupled model, composition naturally generates a hierarchical model.

2.3 Simulation structures

In its basic formulation (Zeigler *et al*., 2000), a PDVES atomic component comes with an associated distinct *simulator* which is responsible of its time management. The simulators of the various internal components of a coupled model are orchestrated by a *coordinator* which ensures consistent time management through proper message interchange. The coordinator, in particular, guarantees that multiple internal simulators can make simultaneous internal/confluent transitions (maximal parallelism). Toward this, first *all* the outputs generated by parallel components are collected, then they are processed by relevant destination components.

As pointed out e.g. in (Himmelspach & Uhrmacher, 2006), the default simulation architecture for PDEVS has a negative impact on the execution performance which prevents simulation of large models. The overhead is mainly due to individual simulators which map on distinct execution threads which obviously introduce space and time problems. In addition, the number of exchanged messages among simulators and the coordinator also contributes to execution overhead. In the literature, either a hierarchical model is preliminarily flattened into one in which coupled models no longer exist, e.g. (Kim *et al*., 2000), or some form of a sequential non-threaded simulation algorithm is employed (Glinsky & Wainer, 2005)(Himmelspach & Uhrmacher, 2006) which e.g. retains the hierarchical structure but improves specifically the simulation operation. In this work, an agent-based M&S framework is used for the experiments where both flattening and a single simulation engine are adopted for executing a PDVES system.

3. CONFLICT AND TIME MANAGEMENT ISSUES

PDEVS formal definition does not account for the existence of conflicts among parallel components. When a component exhausts its dwell-time in its current state, its behaviour is committed: the output function gets executed, then either the internal (no external event arrived) or confluent (an external event arrived) function is run to establish the next state. In other terms, the arrival of an external event in the last time of staying in the current state, does not prevent the output function to be executed. This behaviour seems to contradict the assumption that "there is no way to generate an output from an external transition". Indeed, in many modelling examples the confluent function is redefined to behave exactly as the external function. In this case, an output is generated *just* before an external transition! Independently from any function signature consideration, the question is not of purely syntactical nature because it affects semantics and directly leads to the problem of conflict management in PDVES.

Consider a component which models a sender in a protocol, which after each message transmission is willing to await a certain maximum time (timeout) for an ack to be received, before re-sending the message (as part of the output function). In the case the ack event arrives strictly before timeout expiration (the external transition function is invoked) there is no doubt that the timeout consequence (message resend) must be cancelled and the timeout reset. The same behaviour should be allowed to occur even when the ack event arrives in the last time of the timeout. In the light of standard PDEVS, the arrival of ack in the last moment of sender timeout, causes in any case (i.e., with or without a redefinition of the confluent function) the message to be resent. Situations like this demand for proper management of conflicts. This work argues that it is prudent to not anticipate execution of the output function when the confluent function is to be invoked. Rather this matter should be left to the semantics of the confluent function itself.

The confluent function of PDEVS replaced the select() function of Classic DEVS (Zeigler *et al*., 2000) which was specifically introduced for *tie-breaking* when multiple components of a coupled model have state transitions occurring at the same simulation time. select() typically realizes a deterministic tie-breaking. It is the viewpoint of this work that also the use of select() is unsatisfactory for conflict management because it tends to resolve conflicts in an inflexible way.

Another critical issue of PDEVS semantics concerns time management in states. When an external transition is chosen which does not cause current state to be abandoned (i.e., self-loop transition), there is no need to stop time advancement in current state. As a matter of fact, in DEVSJAVA (Zeigler & Sarjoughian, 2003) a continue() function is used to explicitly state that time should be allowed to continue advancing when a self-loop transition occurs. This work favours the more natural choice of letting time implicitly to continue during self-loop transitions. In the case the modeller wants a different behaviour (pre-emption and time restart) a transitory state can be introduced so as to be entered and then immediately exited by returning back to the original state.

The time advancement issue also interacts with the conflict problem. Suppose that an external event arrives in the last time of current state of a component, but the confluent function is redefined to coincide with the external function which commands a self-loop transition (i.e., no effective conflict occurs which pre-empts the component). In this case, restarting time would be wrong because the component *persists* in its current behaviour despite the arrival of the external event. The same argument can obviously be repeated when the external event arrives before the last time and the event effectively does not conflict with the behaviour of the receiving component. In this case too time should *not* be restarted.

The above observations are at the basis of the experience described in this paper, related to modelling the Time Petri Nets –TPNs– (Merlin & Farber, 1976) formalism on top of PDEVS. TPN models are representative of a large class of time–dependent systems (e.g., communication protocols, embedded real–time systems and so forth) where conflicts are to be managed at runtime (race policy of transition firing).

4. TIME PETRI NETS CONCEPTS

The following provides the definition of Time Petri Nets (Merlin & Farber, 1976) assumed in this paper. Basically, Merlin & Farber formalism is augmented with inhibitor arcs. A TPN is a tuple $TPN=(P,T,A,I_{nh},W,M_0,I^s)$ where

- P and T are non empty and disjoint sets respectively of places and transitions of the underlying Petri net (Murata, 1989)
- A is a set of arcs: $A \subseteq P \times T \cup T \times P$
- I_{nh} is a set of inhibitor arcs: $I_{nh} \subseteq P \times T$
- W associates weights to arcs: $W: A \cup I_{nh} \rightarrow N$, with N the set of natural numbers. Weights are assumed strictly positive for arcs in A, 0 for inhibitor arcs
- M_0 is the initial marking: $M_0: P \rightarrow N$ in the usual sense of Petri nets
- I^s is the static firing interval function: $I^s: T \rightarrow R \times (R \cup \{\infty\})$.

Place $p \in P$ is an *input place* for transition t if there is an arc (p,t) in A. Place p is an *inhibitor place* for t if $(p,t) \in I_{nh}$, i.e. there exists an *inhibitor arc* connecting p to t. An inhibitor arc is graphically represented by a dot terminated line. A place p is an *output place* for t if there exists an arc (t,p) in A. The set of input and inhibitor places of t is said its *preset*. The set of output places constitutes the transition *postset*.

I^s associates with each transition t a *dense* firing interval whose bounds are assumed to be specified by non negative reals: $I^s(t)=[a,b]$ with $0 \le a \le b$, b can be ∞. Bound a is said the (static) *earliest firing time* (EFT^s) of t, b the (static) *latest firing time* of t (LFT^s).

Let M be a marking. Transition t is said enabled in M, denoted by $M[t>$, iff

$$M[t> \Leftrightarrow \forall p \in preset(t) \begin{cases} M(p) \ge W(p,t) \text{ if } (p,t) \notin I_{nh} \\ M(p) == 0 \text{ if } (p,t) \in I_{nh} \end{cases}$$

As soon as a transition t is enabled, it *starts* firing (server perspective). The firing end event is constrained to occur in the time interval associated with the transition. Let τ be an instant in time when transition t is enabled. Provided t is continuously enabled, t cannot fire before $\tau+a$ but *must* fire before or at $\tau+b$, unless it is disabled by the firing of another transition (*race firing policy*). At the time transition firing ends, tokens are removed from the input places and new tokens are generated to output places as in classic Petri nets. Let M_{before} be the net marking just before t completes its firing. Firing end of t transforms M_{before} in M_{after}, denoted by $M_{before}[t>M_{after}$, by an instantaneous and atomic process in two phases:

```
(phase 1-token withdrawal)
∀p∈P     if p∈preset(t) then M'(p)=M_before(p)-W(p,t)
         else M'(p)=M_before(p) endif

(phase 2-token deposit)
∀p∈P     if p∈postset(t) then M_after(p)=M'(p)+W(t,p)
         else M_after(p)=M'(p) endif
```

where M' represents the intermediate marking generated after token withdrawal.

A transition t_p is said to be *persistent* to the firing of transition t if $M_{before}[t_p>$ **and** $M'[t_p>$ **and** $M_{after}[t_p>$. After the token deposit phase one or more transitions t_{new} can become enabled, i.e. $M_{after}[t_{new}>$.They are said *newly enabled transitions*. The token deposit phase, though, because of the existence of inhibitor arcs, can also reveal non persistent transitions t_{np} which lose their enabling in M_{after}, i.e. $M_{before}[t_{np}>$ **and** $M'[t_{np}>$ **and not** $M_{after}[t_{np}>$. A subtle case concerns non persistent transitions characterized by $M_{before}[t_{np}>$ **and not** $M'[t_{np}>$ **and** $M_{after}[t_{np}>$. Firing of these transitions is pre-empted. After that they behave as newly enabled transitions.

4.1 An Example

Fig. 2 shows a TPN model based on the Alternating Bit Protocol (ABP) (Berthomieu & Diaz, 1991). ABP allows transmission of messages between a sender and a receiver over an unreliable transmission medium. Messages or acknowledgments can be lost in transit. Recovery from losses is supported by a timeout mechanism followed by re-transmission. The sender records the time at which it sends a message and in the case the acknowledgment of this delivery is not received in time, the message is re-sent. The mechanism is intended for recovering from losses and for preventing the acceptance of duplicate messages. Upon receiving a message, the receiver must be able to detect if it is a new message or a duplicate of a previous message. Toward this, the sender attaches as header of messages, prior to transmission, a modulo-2 sequence number. In addition, for each received message, an acknowledgment is sent back to the sender that carries the same sequence number of the received packet. Since the sender alternates the transmission of messages with header 0 and header 1, two subnets in Fig. 2 model the sender in the two cases. Two subnets for the channel and the receiver are correspondingly defined.

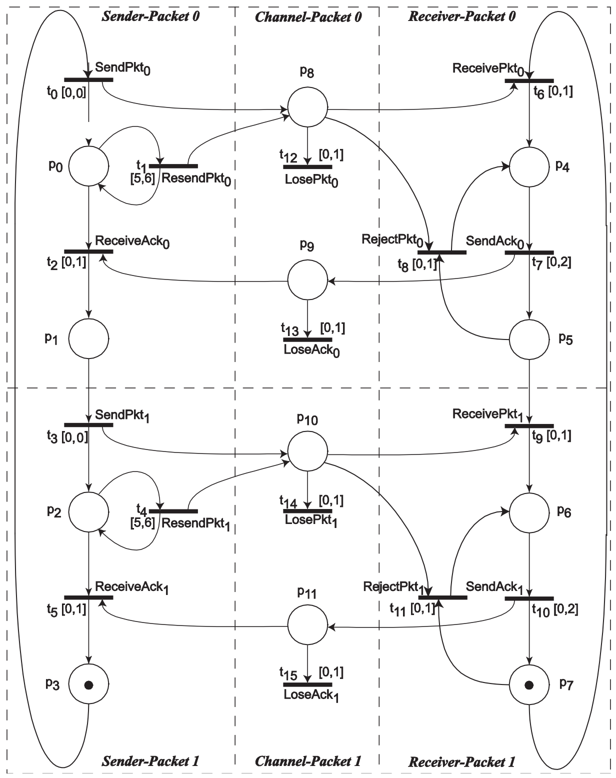

Fig. 2. A TPN model for the Alternating Bit Protocol

Transition t0 represents the sender which transmits a packet with header 0. t1 models the sender which sets a timeout to occur after 5 or 6 time units since message transmission. t2 denotes the sender which receives the acknowledgment of sent message and moves to the lower section in which it transmits a message with header 1. Transitions t3, t4 and t5 have the same meaning respectively of transitions t0, t1 and t2. Transitions t12 and t13 (and similarly t14 and t15) model respectively the channel which can lose a transmitted message or its ack. Receiver transition t6 accepts a message with header 0. Transition t7 sends the ack. However, would a message be received again with header 0 (duplicate), the receiver discards it and then re-transmits the ack. In a similar way operates the subnet t9, t10 and t11 of receiver which handles a message with header 1.

In Fig. 2 some transitions are conflicting to one another. For example, t1 and t2 can never complete their firing together. In the case both transitions should fire at the same time, one of the two is chosen non deterministically and prevents the other to complete its firing. These situations are not directly supported by PDEVS.

5. ActorDEVS

This section summarizes ActorDEVS (Cicirelli *et al*., 2006), an actor (agent) based minimal framework supporting modelling and simulation of PDEVS models in Java. The hosting architecture (Cicirelli *et al*., 2007a-b) rests on a variant of the Actors Model (Agha, 1986). Adopted actors are light-weight thread-less reactive objects. They have a *message interface*, an *encapsulated* internal *data state* and a *behaviour* patterned as a *finite state machine*. The communication model centres on asynchronous message passing. The arrival of a message to an actor triggers a response (handler() method) which includes the following basic actions: (a) modification of internal data (b) control state change (become() method) (c) creation of new actors (d) transmission (send() method) of messages to known actors (*acquaintances*), including itself for proactive behaviour. Elementary actions are exported by the Actor base class. Messages are heir of Message base class. A subsystem of co-located actors (*theatre*) runs on a computing node. Message scheduling/dispatching services are provided in a theatre by a *control machine* which can be specialized to discrete-event simulation or real-time operation. Multiple theatres can synchronize to one another for distributed simulation (Cicirelli *et al*., 2007a).

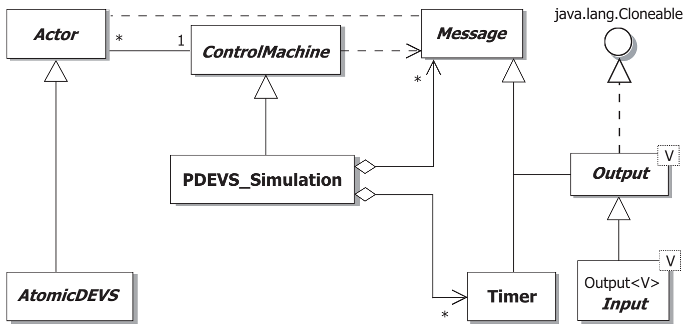

Fig. 3. UML class diagram of ActorDEVS

The UML class diagram in Fig. 3 depicts the ActorDEVS framework, which rests on a few design patterns (Gamma *et al*., 1995). For example, the strategy pattern is employed to transparently weave the control structure, i.e. the simulation engine, to an application; the template method is adopted for structuring the AtomicDEVS abstract actor class which is the base for achieving, through inheritance, the concrete atomic components required by the application. Typed input/output ports of components are directly mapped on to messages. Toward this, 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>. Class Output<V> exploits the command and prototype design patterns. As a command, it has a route() operation to transmit its content to its destination actor, whose identity is established at configuration time. As a prototype, an output message is cloned in order to create an initialized copy of itself which is actually sent to its destination actor. Services get()/set() permit respectively to retrieve/modify the data component of an Output<V> message. Method linkTo(receiver) allows an output message to be bound to a given receiver actor. The Output<V> class is provided of a *recycler* so as to avoid cloning when a consumed (i.e., yet processed) output message is available for recycling. A Timer class is provided which inherits from Message and supports the notion of a timed message. A timer can be set()/reset() and the elapsed time since its setting or the remaining time before its expiring can be checked. When a timer is set, the identity of a timeout message, its receiver actor, and the relative expiration time must be given. Before expiring, a timer can be reset. In this paper a discrete time model is assumed. However, a dense time model could in alternative be used. Current time is available to any component through the method now() of Actor which in turn depends on the time notion of a control machine.

Different control machines can be built. PDEVS_Simulation used in this paper manages the simulation clock and the following message data structures: a priority queue (heap-based) of set timers, and a collection of bags of instantaneous messages, separately maintained on a per component basis, which are to be dispatched at current simulation time. Instantaneous messages take precedence with respect to timers. The control structure repeats a basic loop. At each iteration, a bag of instantaneous messages, if there are any, directed to a receiver atomic actor is selected and passed to an invocation of the handler() method of AtomicDEVS. If there are no pending instantaneous messages, the most imminent timer is allowed to fire thus advancing the simulation time. The corresponding timeout message is then passed to receiver handler() method.

PDEVS_Simulation is an efficient control engine. Sources of efficiency are: (a) the adopted message data structures, (b) the flattening of coupled models (more is said later in this section) which implies a reduction in the number of exchanged messages, (c) the automatic recycling of processed messages which in most applications reduces the risks of garbage collection interventions.

The programming style descends from the abstract class AtomicDEVS which exports basic PDEVS functions as abstract methods which a user-defined component class must override. The actual signature of functions is clarified in Fig. 4.

```
public abstract int delta_int( int phase );
public abstract int delta_ext( int phase, long e, Iterable<Message> x );
public int delta_con( int phase, long e, Iterable<Message> x ){//default
    lambda( phase );
    return delta_ext( delta_int( phase ), 0, x );
}//delta_con
public abstract long ta( int phase );
public abstract void lambda( int phase );
```

Fig. 4. Signatures of PDEVS functions

As one can see, all transition functions return an int which codifies the next phase of the atomic component (from the perspective of the actor model, a PDEVS component is a finite state machine built over the control states or phases of the component). For simplicity, functions receive only the phase portion of the component state. The remaining part of the state (i.e. state variables) is assumed to be directly accessed from within the function bodies in the Java class which realizes the component. The delta_con() method is concrete in AtomicDEVS and implements the default behaviour of the confluent function. Of course, a concrete component can redefine delta_con() to achieve a different behaviour.

A bag of inputs is an object created by PDEVS_Simulation, of an Iterable<Message> class. This way the modeller can navigate over the received messages (i.e. external events) by iterating the x object.

The void handler(Iterable<Message> x) method of AtomicDEVS implements PDEVS semantics, i.e. it is in charge of making internal/external transitions and of checking simultaneity of an external event and an internal event (internal transition) in which case the confluent function is invoked. AtomicDEVS uses a timer and a timeout built-in message to enforce the temporal behaviour of the component.

ActorDEVS naturally borrows from the underlying actor model its component-based character and the compositional mechanism for making coupled models (Cicirelli *et al*., 2007b). Mechanisms prove effective for building highly dynamic systems whose structure can change in the runtime. To avoid message overhead during communications with coupled models, i.e. to ensure hierarchical models are flattened from the point of view of the control machine, it is sufficient to override the send() method in coupled model/actors. The overridden send() can (synchronously) redirect to internal components the external incoming messages directed to the coupled model. Similarly, internally generated messages destined to external components can be relayed to their final destinations by having the coupled component which directly (and transparently) configures the output ports of internal components so as to refer to their relevant external receivers.

6. MODELLING TIME PETRI NETS USING ActorDEVS

The following describes a mapping of Time Petri Nets (TPNs) on to ActorDEVS. The approach can easily be adapted to work with other time-extended Petri net formalisms. The modelling process associates each transition with a distinct PDEVS atomic component. Every transition gets equipped of its preset and postset. Places are passive topological objects, holding configuration information only. Each place knows transitions of which it is an input place. At each marking change, a place automatically notifies its relevant transitions which thus have a chance to check their enabled/disabled status. TPN modelling is based on a few classes as shown in Fig. 5.

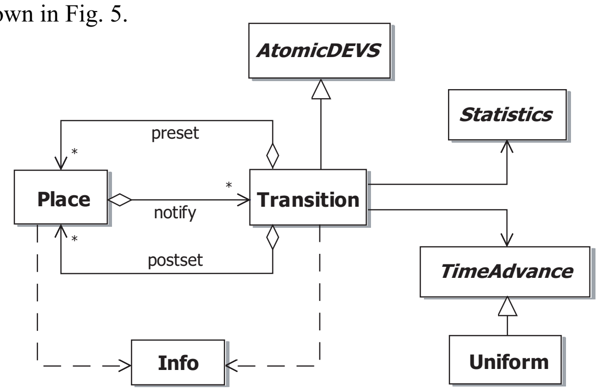

Fig. 5. Class diagram for TPN modelling in ActorDEVS

TimeAdvance defines the long timeAdvance() method which returns the firing time of a transition. Uniform is a specialization of TimeAdvance. It receives the static firing interval of a transition.

Time advancement is uniformly distributed in the firing interval. Info objects are used by a place when notifying transitions. An info object contains the id of a place, the marking of the place and a boolean value telling if this marking was generated by a withdraw or a deposit operation. Statistics is the ancestor of *transducer* classes. Transitions of a given TPN model can be fed by a specific statistics object useful for collecting information about model dynamics. A statistics object can be initialized with the entire model marking vector. Method fire() of Statistics receives the id of a fired transition and the associated occurrence time.

A transition admits two phases: PASSIVE (not enabled) and FIRING (enabled and under firing). Input port interface (message types) and constructor of Transition are illustrated in Fig. 6.

```
public class Transition extends AtomicDEVS{
    public static class Notify extends Input<Info>{}//Notify
    private static class Commit extends Message{}//Commit
    public Transition( int id, TimeAdvance tA, Statistics stat ){
        this.id=id; this.tA=tA; this.stat=stat;
        initialPhase( PASSIVE );
    }
    …
    private final byte PASSIVE=0, FIRING=1;
    private Commit commit=new Commit();
    …
}//Transition
```

Fig. 6. An excerpt of the Transition class

A transition receives Notify messages from its input places at each marking change. Commit is a local message class. One single commit message exists which the transition sends to itself during a firing process. This particular ability of ActorDEVS atomic components derives from their actor character.

Internal transition function delta_int() (Fig. 7) simply returns PASSIVE as the next status.

```
public int delta_int( int phase ){
    return PASSIVE;
}//delta_int
```

Fig. 7. Internal transition function

The time advance function ta() is shown in Fig. 8. As soon as the transition moves to the FIRING phase, its dwell-time is established by asking the TimeAdvance object tA.

```
public long ta( int phase ){
    if( phase==PASSIVE ) return INFINITY;
    return tA.timeAdvance();
}//ta
```

Fig. 8. Time advance function

The output function lambda(), which is invoked just before a transition completes its firing, is depicted in Fig. 9.

Fig. 10 shows (partially in pseudo-code) the helper method enabled() which checks the enabling status of the transition. enabled() receives explicitly the marking to be checked. This solution was adopted to properly implement the two phases of the atomic firing process described in section 4. In particular, the Transition component maintains a "withdraw marking" wm and a "deposit marking" dm. wm captures the intermediate marking generated just after the withdrawal sub-phase of a firing process. dm corresponds to the marking at the end of the deposit sub-phase of the firing process.

```
public void lambda( int phase ){
    if( phase==FIRING ){
        //token withdrawl
        for( Place p: preset ){
            p.withdraw( preset.weight( p ) );
        }
        //token deposit
        for( Place p: postset ){
            p.deposit( postset.weight( p ) );
        }
        if( stat!=null ) stat.fire( id, now() ); // signal this firing to statistics
    }
}//lambda
```

Fig. 9. Output function

```
private boolean enabled( Marking m ){
    for( Place p: preset ){
        int weight=preset.weight( p );
        if( weight==0 && m( p ).tokens()!=0 ) return false;
        if( weight>0 && m( p ).tokens()<weight ) return false;
    }
    return true;
}//enabled
```

Fig. 10. Helper function checking the enabled status

```
public int delta_ext( int phase, long e, Iterable<Message> x ){
    if( x.iterator().next()==commit ){
        if( phase==PASSIVE && enabled( dm ) ) phase=FIRING;
        else if( phase==FIRING && !enabled( dm ) )  phase=PASSIVE;
    }
    else{//bag of notify messages
        //first scan
        for( Message m: x ){
            if( m is a Notify message whose Info is tagged "withdraw" ){
                update wm with marking information in m;
                update dm with marking information in m;
            }
        }
        //second scan
        for( Message m: x ){
            if( m is a Notify message whose Info is tagged "deposit" ){
                update dm with marking information in m;
            }
        }
        if( phase==PASSIVE && enabled( wm ) ) phase=FIRING;
        else if( phase==FIRING && !enabled( wm ) ) phase=PASSIVE;
        send( commit ); //prepare to commit
    }
    return phase;
}//delta_ext
```

Fig. 11. External transition function

Fig. 11 portrays in pseudo-code the body of the external transition function delta_ext(). delta_ext() gets invoked as soon as this transition, which is under firing, receives a bag of Notify messages from its input places. In response to this bag the transition verifies its persistent or non persistent status with respect to just fired transition. All of this is important for properly disabling transitions in effective conflict with the fired one. The delta_ext() function navigates two times the bag of Notify messages. The first scan serves for building the withdraw marking wm and (partially) the deposit marking dm (some places are both input and output places). The second scan completes definition of the deposit marking dm.

In order to separate the effect of withdraw from that of deposit, this transition first takes consequence of marking wm, then it sends to itself the commit message triggering again the execution of delta_ext() which definitely establishes the next status of the transition. It is guaranteed that this transition will receive the commit message *before* another transition can begin its firing process. All of this is a consequence of the fact that PDEVS_Simulation engine sequentially fires internal transitions (time events) of components.

Fig. 12 shows the confluent function delta_con() which is redefined so as to coincide with the delta_ext().

```
public int delta_con( int phase, long e, Iterable<Message> x ){
    return delta_ext( phase, e, x );
}//delta_con
```

Fig. 12. Confluent transition function

The resultant behaviour ensures that when a transition *t* completes its firing, it immediately notifies all its *influencees*, i.e. transitions having in their preset places affected by *t* firing. As a consequence, all conflicting transitions with *t*, even at last time in their firing interval, can get disabled thus interrupting their current firing. On the other hand, an under firing transition which persists to *t* firing, keeps unaltered its time advancing status.
As a final remark, class Transition exports addPreset(place[,weight])/ addPostset(place[,weight]) methods (when omitted, the weight defaults to 1) which facilitate configuration of a TPN coupled model.

6.1 Coupled Model for the Alternating Bit Protocol

Fig. 13 shows a summary of the configuration operations which create a coupled model corresponding to the ABP TPN model of section 4.1. For demonstration purposes, the statistics object is only bound to channel transitions t12, t13, t14 and t15 which lose a message or an ack, to transition t0 and t3 which transmit a message, and to transitions t1 and t4 which model timeout and message re-transmission. The simulation lasts 10^6 time units.

Setting the initial marking of places forces transitions to start querying their enabling status.

As a simple example of property checking, "correct" setting of timeouts in Fig. 2 was monitored. To this end the minimal ABPStat class shown in Fig. 14 was prepared which is able to compute (i) the maximum delay time of a loss event since its causing send/resend operation, and (ii) the minimum/maximum elapsed time between a loss event and its subsequent firing of timeout which resends the message.

```
public class ABP{
    public static void main( String []args ){
        //create control machine
        ControlMachine cm=new PDEVS_Simulation( 1000000 /*tEnd*/ );
        //create places
        Place p0=new Place(0); //0 is the place ID
        …
        Place p11=new Place(11);
        //create statistics object
        Statistics stat=new ABPStat();
        //create transitions
        Transition t0=new Transition( 0, new Uniform(0,0), stat);
        Transition t1=new Transition( 1, new Uniform(5,6), stat );
        Transition t2=new Transition( 2, new Uniform(0,1), null );
        Transition t3=new Transition( 3, new Uniform(0,0), stat);
        Transition t4=new Transition( 4, new Uniform(5,6), stat );
        …
        Transition t12=new Transition( 12, new Uniform(0,1), stat );
        Transition t13=new Transition( 13, new Uniform(0,1), stat );
        Transition t14=new Transition( 14, new Uniform(0,1), stat );
        Transition t15=new Transition( 15, new Uniform(0,1), stat );
        //configure preset/postset of transitions
        t0.addPreset( p3 ); t0.addPostset( p0 ); t0.addPostset( p8 );
        …
        //configure place output ports
        Output p0_out1=new Transition.Notify();
        Output p0_out2=new Transition.Notify();
        p0_out1.linkTo( t1 ); p0_out2.linkTo( t2 );
        p0.setOut( p0_out1, p0_out2 );
        …
        //set model initial marking
        p0.initMarking( 0 ); p1.initMarking( 0 );
        p2.initMarking( 0 ); p3.initMarking( 1 );
        …
        cm.controller(); //start simulation
        System.out.println( stat ); //display statistics
    }
}//ABP
```

Fig. 13. An ActorDEVS coupled model for the Alternating Bit Protocol

```
public class ABPStat extends Statistics{
    private     long tLoss, tSend, tResend, maxDelayToLoss=0,
                minDelayFromLossToResend=Long.MAX_VALUE,
                maxDelayFromLossToResend=0;
    public void fire( int id, long now ){
        if( id==0 || id==1 || id==3 || id==4 ){
            tSend=now;
            if( id==1 || id==4 ){
                tResend=now;
                if( tResend-tLoss>maxDelayFromLossToResend )
                    maxDelayFromLossToResend=tResend-tLoss;
                else if( tResend-tLoss<minDelayFromLossToResend )
                    minDelayFromLossToResend=tResend-tLoss;
            }
        }
        else if( id>=12 || id<=15 ){
            tLoss=now;
            if( tLoss-tSend>maxDelayToLoss )
                maxDelayToLoss=tLoss-tSend;
        }
    }//fire
    public String toString(){
        return "Max delay from (re)send to loss="+maxDelayToLoss+"\n"+
        "Min delay from loss to resend="+minDelayFromLossToResend+"\n"+
        "Max delay from loss to resend="+maxDelayFromLossToResend+"\n";
    }//toString
}//ABPStat
```

Fig. 14. The ABPStat class

At simulation end the ABPStat furnished the following data:

```
Max delay from (re)send to loss=4          (1)
Min delay from loss to resend=1            (2)
Max delay from loss to resend=6            (3)
```

As one can see from point (1) a message can be lost after a maximum of 4 tu are elapsed from the send/resend message operation. Moreover, following a loss event, the consequent timeout event is fired after 1 to 6 tu (points (2) and (3)). The minimal delay of 1 tu occurs when the loss event delays of its maximum and the timeout (t1 or t4) fires at its *earliest* time. The maximum delay of 6 tu happens when the loss event occurs immediately (t12 or t14 fires after 0 tu) and the consequent timeout event fires at its *latest* time. In the light of above discussion, the simulation experiments confirmed that the time interval [5,6] of timeout transitions t1 and t4 is correctly established.

7. CONCLUSIONS

This paper describes an experience about managing conflicts in PDEVS. As an example, a mapping of Time Petri Nets (Merlin & Farber, 1976) on to PDEVS is proposed, which permits modelling and simulation of communication protocols or embedded real-time systems. Experiments were conducted using ActorDEVS, a minimal flexible and efficient agent-based framework developed in Java.

Owing to its underlying actor-based framework, ActorDEVS can work with different control engines. For example, both standard behaviour of (flattened) PDEVS and a particular control structure suited to conflict management were implemented.

The following points emerged from the experience which affect PDEVS operation:

- it is wise to not anticipate execution of the output function when the confluent function is up to be invoked. Rather the matter should be left to the semantics of the confluent function itself
- it is convenient to not stop time advancement in states when an external event triggers a self-loop transition
- it is useful for an atomic component to possibly send messages to itself.

The described work is being extended in the following directions:

- automating the construction of ActorDEVS coupled models corresponding to TPN models expressed in XML (Billington *et al*., 2003)
- mapping in ActorDEVS other time-extended Petri net formalisms, possibly with modularity constructs (e.g. supporting an abstract transition notion which can be refined with a subnet) and high-level concepts (tokens as colours)
- using ActorDEVS for modelling and simulation variable structure systems (Hu *et al*., 2005)(Cicirelli *et al*., 2007b)
- experimenting with ActorDEVS in distributed simulations. The framework is being ported to distributed contexts where the transport layer can be either Java Socket, Java RMI or HLA/RTI (Cicirelli *et al*., 2007a).

REFERENCES

Agha, G. (1986). *Actors: A model for concurrent computation in distributed systems*, Cambridge, MIT Press.

Berthomieu B. and M. Diaz (1991). Modelling and verification of time dependent systems using time Petri nets. *IEEE Trans. Software Eng.*, **17**(3):259-273.

Billington J., S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber (2003). The Petri net markup language: concepts, technology, and tools. In *Proc. of the 24th Int. Conf. on Application and Theory of Petri Net*s, LNCS 2679, pages 483–505. Springer.

Cicirelli F., A. Furfaro and L. Nigro (2006). A DEVS M&S framework based on Java and actors. In *Proc. of 2nd European Modelling and Simulation Symposium (EMSS 2006)*, pp. 337-342.

Cicirelli F., A. Furfaro, A. Giordano and L. Nigro (2007a). An agent infrastructure for distributed simulations over HLA and a case study using unmanned aerial vehicles. In *Proc. of 40th Annual Simulation Symposium*, IEEE Computer Society Press, pp. 231-238, March, Norfolk (VA).

Cicirelli F., A. Furfaro, L. Nigro and F. Pupo (2007b). A component-based architecture for modelling and simulation of adaptive complex systems. *21st European Conference on Modelling and Simulation (ECMS'07),* 4-6 June, Prague.

Hu X., B.P. Zeigler and S. Mittal (2005). Variable structure in DEVS component-based modelling and simulation. *Simulation*, **81**(2):91-102.

Gamma E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns*. Addison-Wesley.

Glinsky E. and G. Wainer (2005). Devstone: a benchmarking technique for studying performance of devs modelling and simulation environments. In *9th IEEE Int. Symposium on Parallel and Distributed Simulation and Real-Time Applications*, pp. 265-272.

Himmelspach J. and A.M. Uhrmacher (2006). Sequential processing of PDVES models. In *Proc. of 2nd European Modelling and Simulation Symposium (EMSS 2006)*, pp. 239-244.

Kim K., W. Kang, B. Sagong and H. Seo (2000). Efficient distributed simulation of hierarchical DEVS models: transforming model structure into a non-hierarchical one. In *Proc. of the 33rd Annual Simulation Symposium* (*ANSS'00*), IEEE Computer Society, pp. 227-233.

Merlin P. and D. Farber (1976). Recoverability of communication protocols – implications of a theoretical study. *IEEE Transactions on Communications*, **24**(9):1036–1043.

Murata T. (1989). Petri nets: properties, analysis and applications. *Proc. of the IEEE*, **77**(4), pp. 541-580.

Shaw M. and D. Garlan (1996). *Software architecture: perspective on an emerging discipline*. Prentice-Hall.

Vangheluwe H.L. (2000). *DEVS as a common denominator for multi-formalism hybrid systems modelling*. In Andras Varga, ed., *IEEE Int. Symposium on Computer-Aided Control System Design*, pp. 129-134, IEEE Computer Society Press, September, Anchorage, Alaska.

Zeigler B.P. and H.S. Sarjoughian (2003). Introduction to DEVS modelling and simulation with Java: developing component-based simulation models. *http://www.acims.arizona.edu*.

Zeigler B.P., H. Praehofer, and T.G. Kim (2000). *Theory of modeling and simulation*. 2nd Edition, New York, NY, Academic Press.