Model and Simulation Engines for Distributed Simulation of Discrete Event Systems

José Ángel Bañares and José Manuel Colom

Computer Science and Systems Engineering Department
Aragón Institute of Engineering Research (I3A)
University of Zaragoza, Zaragoza, Spain
{banares,jm}@unizar.es

Abstract. The construction of efficient distributed simulation engines for discrete event systems (DES) remains a challenge. The vast majority of simulations that are developed today are based on federation of modular sequential simulations. This paper proposes the steps to fill the gap from specifications based on Petri Nets to an efficient simulation of the net throughout a distributed application devoted to this purpose and exploiting the versatility of cloud infrastructures. The outcomes of the proposed DES distributed simulation are: (1) an adapted execution model of PN that is based in the generation and management of events related to the enabling and occurrence of transitions; (2) simple simulation engines for these adapted PN, each hosting a subset of transitions; (3) an scheme for deployment of a set of connected simulation engines; and (4) a simple mechanism for dynamic load balancing by merging/splitting the subsets of transitions hosted in simulation engines.

Keywords: Distributed simulation · Discrete event systems · Dynamic load balancing · Petri nets.

1 Introduction

In many fields, ranging from healthcare monitoring to industrial manufacturing applications, the systems are becoming very large and complex. Moreover, they must be designed as part of an interconnected world. Smart systems (Cities, Buildings, Factories, Logistics) are examples of such systems. They share a set of characteristics such as: involvement of physical and computational interactions, integration of human behaviour into the processes, consideration of sustainability and economical requirements, and achievement of unprecedented levels of scale and complexity. The construction of models for these systems, that retain the essential elements and parameters for its design, is an accepted strategy to cope with these systems. Nevertheless, the modelling of these systems often gives rise to models that cannot be used in practice in the design, analysis or implementation processes. These problems arise because of the high-level semantics of the models obtained or the size of the model, which makes the model unmanageable for the available tools inside the engineering process oriented to design, the evaluation of architectural solutions, or the assessment of system performance.

For this kind of complex and scalable systems, simulation becomes the only alternative available in practice for the different tasks in its life cycle. In this case, it is an essential tool for system operation, to dynamically enable the continuous design, configuration, monitoring and maintenance of operational capability, quality, and efficiency. The capacity to trigger simulations in a short period of time to anticipate the effect of control actions is an essential tool to transform the high-volume of continuously streaming data into knowledge for decision support.

This paper is focused on discrete event Systems (DES), where the evolution from one system's state to another is produced as a consequence of the appearance of a *discrete relevant fact* for the system that is called *event*. The system's actions happen or are executed while the system is in a state and have an associated temporary duration, an economic cost, etc. The completion of a system action causes the state change of the system in an atomic manner. The simulation of a DES consists in the execution of a model that represents the system. This model must represent the state of the system and the state transitions that are the discrete state changes that may occur at discrete points in simulation time, and when an event occurs [33].

The introduction of acceleration techniques in simulation applications has been a permanent objective that has been strongly related with the growth of the size of the systems to be simulated. Parallelism and distribution are techniques oriented to this goal. However, to obtain simulation execution times better than in a centralized simulation it is necessary to consider a careful selection of the execution model to be used, the partition and distribution of the model to be simulated, and the analysis of message traffic between the simulation engines, which in general are closely coupled tasks.

Parallel and Distributed Discrete Event Simulations tools offer the ability to perform detailed simulations of large-scale computer networks [11], traffic [32], and military applications [29], among other applications. Despite the relevance of large-scale DES simulations, this type of problem is far from be solved and poses important challenges [14, 12]. The difficulty to move these applications to the cloud can be exemplified by the modelling and simulation of the cloud itself [5]. A review of thirty-three cloud simulators is presented in [4], but just one of the tools reviewed *cloud2Sim* considers distributed simulations.

The main challenges of Distributed DES simulations pointed in [12] are:

- The definition of **modeling languages** allowing the generation of *efficient parallel and distributed simulation code*.
- The statement of a **clear execution semantic of the model**, and the **execution policy** of its interpreter [18, 26]. They must be oriented to a distributed implementation.
- The availability of ***load balancing mechanisms*** to cope with the unpredictability of the underlying execution environment [8, 7].
- The incorporation of the ***economic cost*** and ***energy consumption***, in addition to the traditional speedup metric in distributed simulations.

Petri Nets (PNs) have been pointed out as a good formalism for modelling realistic features and perspectives of reactive and distributed systems, such as control flow, data, resources, and for analyzing and verifying many properties. The automatic analysis of properties is supported by software tools, and when formal analysis become impracticable, the model must be simulated. In this paper we focus on these challenges using PNs. Section 2 presents succintly the work related to the distributed simulation of DES. Section 3 provides a global overview of the methodology. It covers the automatic translation of PN specifications in efficient parallel and distributed code, the impact of distribution on the execution policies, the definition of simple interpreters to support a distributed simulation, and the support for efficient load balancing between distributed simulators. In this paper we specialize the methodology presented in [30] for conceptual modeling of DES in distributed simulation. The methodology is supported by a high level PN (HLPN) based specification supporting modularity and hierarchy for the modeling of complex systems that was presented in [20]. In our previous work [19], we show how to translate a HLPN into a flat model by means of an elaboration process. In this paper we focus into the process to automatize the translation of flat PN models to efficient parallel simulation code. Finally, section 4 presents an actor architecture for an efficient distributed simulation exploiting the PN execution representation.

2 Related work

There has been a significant amount of work in the field of parallel and distributed DES simulation. A historical review can be found in [10], and many of the current challenges of the discipline has been recently collected in [12].

The discipline began defining *logical processes (LP)* and the *synchronization problem* with what is known as the Chandy/Misra/Bryant algorithm. Synchronization protocols and variants of conservative and optimistic approaches continues to be a focus of research to address synchronization and performance issues associated with executing parallel discrete event simulations in cloud computing [17].

The other focus of research is concerned with an architectural point of view, the development of middleware, frameworks and standards. The High Level Architecture (HLA) is an standard developed by the United State's Department of Defense to perform distributed simulations for military purposes that became an Open IEEE Standard, and has been adopted as the facto standard for federating simulations [31]. Dynamic balancing for HLA-based simulations remains a challenge [8].

Distributed computing programs do not have the same requirements as those of parallel DES programs and thus infrastructure must be specifically designed to support this simulation environment. In [13], Fujimoto et al. propose a master/worker architecture called *Aurora*. Cloud computing is focusing the research with the expectation that the development of Simulations as a Service will hide

the difficulty of developing efficient parallel simulation and will made distributed simulation broadly accesible to all users [27].

Related with the use of PNs for simulating DESs, the translation of a system model expressed by a PN to an actual hardware or software system with the same behavior as the model is a PN implementation. Given a PN model of a DES, the *simulation* of the system can be done by *playing the token game*, i.e. by moving tokens when transitions are enabled. If a deterministic or stochastic time interpretation is associated to transitions – Timed PNs (TPNs) or Stochastic PNs (SPNs) –, the interpretation of the TPN or SPN yields, actually, a Discrete Event Simulation system.

The implementation of a PN can be classified as *compiled* or *interpreted*. The *compiled* implementation generates code whose behavior corresponds to PN evolutions, while an interpreted PN codifies the structure and marking as data structures used by one or more interpreters to make the PN evolve. Compiled implementations has been the option for the development of discrete event control systems [21, 25]. *Interpreted* implementations has the advantage of separating the model specification from the simulator, which provides a number of benefits summarized by Z. Zeigler in [33]: (1) The model is not wired with the simulator, which enables the portability of the model to other simulator and interoperability at a high level of abstraction; (2) Algorithms for distributed simulation can be presented independently of the model; and (3) Model complexity is related with the number of resources required to correctly simulate a model. All these benefits are related with requirements for a distributed simulation. The principle of separation of model and simulator remains the base for scaling resources according to the size of the model, workload balancing by moving parts of the model between distributed simulator engines, and reusing good well defined PDES algorithms.

There has been substantial work in the 1990's on distributed simulation of TPNs [1, 28, 6, 23, 9], which show the TPN formalism can contribute to the efficient implementation of distributed discrete event simulations thanks to the PN structure. These works focused on good partitioning algorithms based on the PN structure and synchronization algorithms. In these works, the TPN is decomposed into a set of LPs assuming a FIFO communication. The interface of the LP is defined by a subset of places, and arcs connecting with these places are replaced by communication channels. LPs interact exchanging time-stamped messages that represent token transfers, and each LP executes a simulation engine that implements the same simulation strategy to interpret the PN partition and to preserve causality with events simulated by other LPs. However, these approaches do not consider the automated translation of the PN structure to efficient code for simulation engines.

3 Event driven simulation based on Petri nets

3.1 The overall methodological approach

In this section we present an approach to automatize the translation of TPN specifications into efficient parallel and distributed simulation code. The pro-

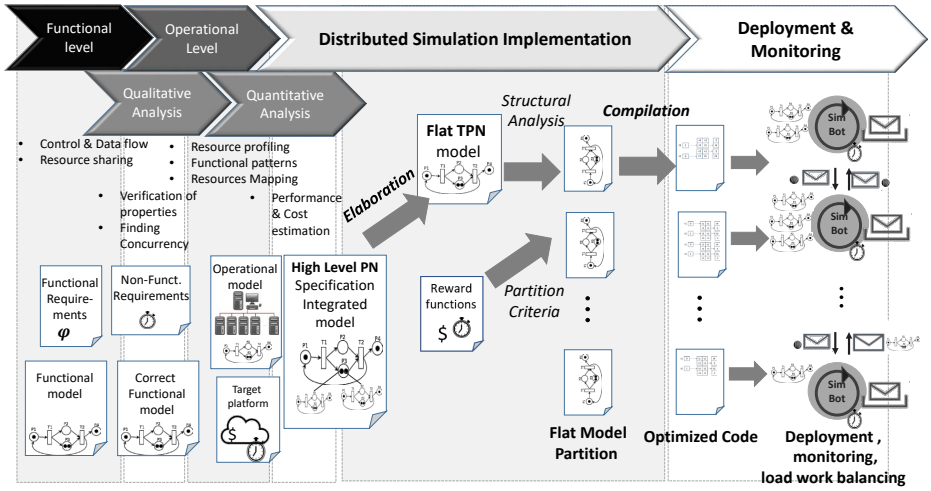
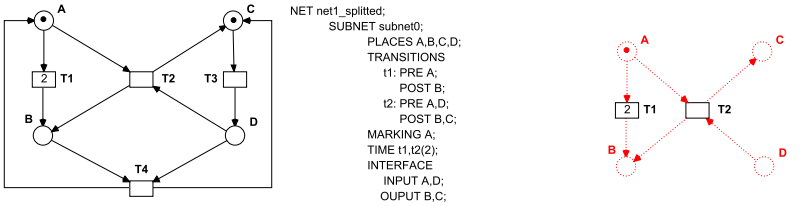


Fig. 1: PN-based process from specification to distributed model execution.

posed steps are part of a methodological approach to manage the complexity of developing the logic of a complex system taking into account functional and not functional requirements, and gradually incorporating restrictions imposed by the underlying hardware infrastructure that was presented in [30]. Figure 1 depicts some steps of the methodology: *Functional models of systems* are built focusing on a set of concurrent communicating processes competing for shared resources. *Qualitative analysis* checks the model and help to find maximum concurrency. The *operational model* enriches the model with characteristics of the execution platform to develop a *quantitative analysis*. This analysis provides information useful for the partition of the model providing metrics required for the distributed execution. Figure 1 focuses on the last steps:

Elaboration The objective of the first steps is the modelling of complex and large scale DES. It requires a formal description of different facets supported by a hierarchical and component decomposition. Modular and hierarchical PN specification, such as introduced in [20], provides more compact and manageable descriptions. However, the interpretation of a high level model can introduce important sources of inefficiency due to higher levels of abstraction such as efficient matching to evaluate enabled transitions [22], or introduce complex synchronization protocols in distributed simulations. Instead of the direct emulation of high level models, we propose transform the original model to be simulated into a flat Place/Transition net model. This transformation process called *elaboration* was illustrated in [19] with the elaboration of a high level PN to a flat model of sequential state machines.

Compilation. The compilation stage transforms a flat Place/Transition net into executable code/data. A classical PN simulation engine follows a repeti-




```
NET net1;
    SUBNET alone_sunet;
        PLACES A,B,C,D;
        TRANSITIONS
            t1: PRE A;
                POST B;
            t2: PRE A,D;
                POST B,C;
            t3 PRE C;
                POST D;
            t4 PRE B,D;
                POST A,C;
        MARKING
            A,C;
        TIME
            t1,t2(2),t3,t4;
    END_SUBNET;
END_NET;
```

```
NET net1_splitted;
    SUBNET subnet0;
        PLACES A,B,C,D;
        TRANSITIONS
            t1: PRE A;
                POST B;
            t2: PRE A,D;
                POST B,C;
        MARKING A;
        TIME t1,t2(2);
        INTERFACE
            INPUT A,D;
            OUPUT B,C;
    END_SUBNET;
    SUBNET subnet1;
        PLACES A_sub1,B_sub1,C_sub1,D_sub1;
        TRANSITIONS
            t3 PRE C_sub1;
                POST D_sub1;
            t4 PRE B_sub1,D_sub1;
                POST A_sub1,C_sub1;
        MARKING C;
        TIME t3,t4;
        INTERFACE
            INPUT B_sub1,C_sub1;
            OUTPUT A_sub1,D_sub1;
    END_SUBNET;
    SYNCHRONIZATION
        (subnet0,A,B,C,D) <= (A_global,B_global,C_global,D_global)
        (subnet1,A_sub1,B_sub1,C_sub1,D_sub1) <= (A_global,B_global,C_global,D_global)
END_NET
```

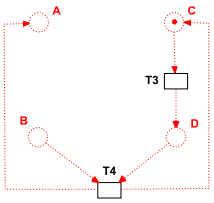



Fig. 2: Graphical, textual and splitted textual specifications of a TPN.

tive cycle that involves: a) to scan enabled transitions; b) to fire some of the detected enabled transitions (executing, maybe, some associated activity), and c) to update the marking (the state) of the PN. Although the elaboration process simplifies the complexity of the enabling tests of transitions by removing the need of unification algorithms, the enabling test in a Place/Transition net remains to consume most of the interpreter loop.

Distribution. Partitioning requires to proceed, a priori, identifying the *good* subnets in which the original one is divided. The initial model partition can be supported by applying structural and behavioral analysis [15], in this sense strategies based into the identification of sequential state machines can be used (computing for example p-semiflows in an incremental way). Alternative partition approaches can be found in [1, 21]. The hardware architecture and syncronization algorithms can be taken into account [7, 2, 19].

Load Balancing. Thanks to a simulation based on identical simulation engines working on data structures and variables representing PNs, it is possible to make a dynamic reconfiguration of the initial partition: (1) by fusion of the data structures of two simulation engines in only one; or (2) by splitting the data structure contained in a simulation engine into two separate data structures over two distinct simulation engines. This dynamic reconfiguration is not possible in simulation contexts where the system to be simulated is not a data structure (e.g. the system is a program that must be compiled).

3.2 Simulation of PNs oriented to distributed implementation

We can compare the interpretation of PN simulation engines with the interpretation of rule based systems (RBS). Both describe how a model evolves in time.

RBS take advantage of *temporal redundancy* based on the idea that most of data in memory does not change when a rule is fired in each interpretation cycle, and most of rules remains enabled or not enabled. Based on this idea, a compilation process builds a (RETE) network that connects state changes with rules affected by state changes, and store partial matching operations. An adaptation of the RETE network for the centralized interpretation of HLPNs was proposed in [22]. It is possible to go beyond improving the efficiency of the PN interpreter by 1) removing complex matching operations in the elaboration process; 2) replacing them by simple linear functions; and 3) incorporating postconditions to the compilation process. In RBS postconditions are left out of the compilation process due to postconditions produces data modifications that can not be related with state changes. However PNs explicitly specify preconditions and state changes giving the possibility of compiling them in a network. This approach is followed in [3] defining the so called ***Linear Enabling Function*** (LEF) of a transition in Place/Transition specifications that allow to characterize when a transition is enabled (can occur).

We propose in this section the translation of the structure and marking of a Place/Transition net to a set of LEFs as optimized code for distributed simulation engines. The LEF of a transition t, $f_t : \mathbf{R}(\mathcal{N}, \mathbf{m_0}) \longrightarrow \mathbb{Z}$, maps each marking $\mathbf{m}$ belonging to the set of reachable markings, $\mathbf{R}(\mathcal{N}, \mathbf{m_0})$, to an integer, in such a way that t can occur for $\mathbf{m}$, iff $f_t(\mathbf{m}) \leq 0$. For example, for transition $T2$ in the net of Figure 2, its LEF is: $f_{T2}(\mathbf{m}) = 2-(\mathbf{m}[A]+\mathbf{m}[D]), \forall \mathbf{m} \in \mathbf{R}(\mathcal{N}, \mathbf{m_0})$, where $\mathbf{m_0}$ is the initial marking depicted in Figure 2 (places A, C marked with a token and the rest of places unmarked). Observe that at $\mathbf{m_0}$, the value of the LEF is $f_{T2}(\mathbf{m_0}) = 1 > 0$, i.e. the transition $T2$ is not enabled at $\mathbf{m_0}$. Nevertheless, at the reachable marking, $\mathbf{m}$, that contains one token in place A and one token in D, $f_{T2}(\mathbf{m}) = 0$ indicating that the transition $T2$ is enabled and can occur.

The use of LEFs, as presented before for the characterization of the enabling of a transition, requires a explicit representation of the marking of the net and the LEF itself as a function. For a distributed simulation this gives rise to two problems that make this execution model of a PN not well-adapted for this purpose: (1) The explicit representation of the marking in a distributed environment is a set of shared variables between a set of distributed simulation engines that requires mechanisms for the maintenance of coherence and consistency of the marking variables (this is is in fact a bottleneck for distributed simulation); (2) The funcional representation of the LEF requires its continuous evaluation for the marking in order to determine the enabling of a transition.

To address these two problems, the LEF mechanism for a transition is implemented according to the following principles: (1) Only the current value of the LEF (initially this value corresponds to the value of the LEF at $\mathbf{m_0}$ and computed in compilation time) is stored; (2) Each time a transition occurs in the net, a constant is sent to each transition which enabling has been affected by the occurrence of the transition. This constant is used for the updating of the LEF of the affected transition.

With this strategy, the explicit representation of the marking and the re-evaluation of the LEF are not needed. The changes of a LEF are based in the constants sent by the transitions that modify its enabling conditions. This is the reason why this execution model for PN's simulation becomes a Discrete Event Simulation because the events are the constants sent by the occurrence of a transition to all transitions whose enabling conditions have been changed by its occurrence, and the simulation state becomes the current values of LEFs.

That is, if $\mathbf{m} \xrightarrow{t'} \mathbf{m'}$, $f_t(\mathbf{m'})$ can be computed from the value of $f_t(\mathbf{m})$ and a static parameter known at compilation time that represents the change of f_t after the occurrence of t'. This parameter corresponds to changes in the contents of tokens of the input places of t as a consequence of the occurrence of t'. Thus, the updating equation for any LEF when t' occurs takes the form $f_t(\mathbf{m'}) = f_t(\mathbf{m}) + UF(t' \longrightarrow t)$, where $UF(t' \longrightarrow t)$ is known as the *Updating Factor* of t' over t obtained from the structure of the net and its initial marking.

According to the previous comments, the compilation of a Place/transition net produces a representation of the net where there is an entry for each transition *(information of the LEF mechanism)*, t, grouping: (1) The variable maintaining the current value of the LEF and initialized to $f_t(\mathbf{m_0})$; and (2) The list of updating factors (simulation's events), $UF(t \longrightarrow t')$, that will be sent to each $t' \in ({}^\bullet t)^\bullet \cup (t^\bullet)^\bullet$ whose enabling conditions have been affected by the occurrence of t. Observe that the partition of a model for a distributed simulation only requires to define the set of transitions to be grouped in each one of distributed simulation engines and load the previous data associated to the transitions in the corresponding engine. The information associated to each transition corresponding to the described LEF mechanism is independent to the information of any other transition. So, in order to perform the dynamic load balancing of the simulation's workload, it is enough to move from one engine to another the information of the LEF mechanism of the transitions to be moved. See Figure 3.

The kernel of each distributed simulation engine to implement the Discrete Event Simulation of the Petri Net, according to the execution model based on the LEF mechanism described before, essentially: (1) Updates the LEFs of transitions with the updating factor interchanged; (2) Scans the list of the variables containing the current values of the LEFs in order to detect the enabled ones (values less than or equal to 0); and (3) Proceeds to make all the operations corresponding to the occurrence of the enabled transitions, executing the associated actions, and sending the list of Updating Factors stored together the value of the LEF. Figure 4 presents an algorithm that implements this kernel of the basic simulation engine [3], using the following information associated to each transition, t', belonging to the part of the PN model to be simulated (See Figure 3): (1) **Identifier** of t'. A *global name* recognised in all sites of the simulation process; (2) $\tau(t')$. **Deterministic firing time** associated to transition t'. It stands for the duration time of the action associated to the occurrence of t'; (3) **Counter.** Variable containing the current value of the LEF $f_{t'}(\mathbf{m})$,initialized with $f_{t'}(\mathbf{m_0})$, and updated whenever the transition –or a transition affecting it– occurs, according to the received Updating Factor; (4) **Immediate Updating**

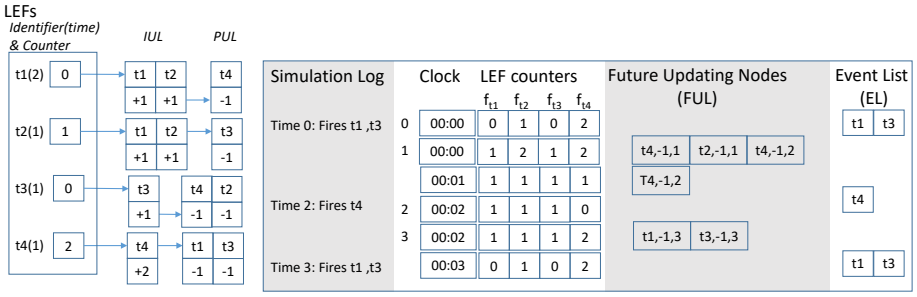


Fig. 3: Compilation result for the PN in Fig. 2 using the LEF mechanism.

List ($IUL(t')$) Set of transitions $({}^\bullet t')^\bullet$ whose LEFs must be updated after the occurrence of t' containing the corresponding Updating Factor to be sent (Note that $({}^\bullet t')^\bullet$ includes t'); (5) **Projected Updating List** ($PUL(t')$) Set of transitions $(t'^\bullet)^\bullet$ whose LEFs must be updated after the occurrence of t' containing the corresponding Updating Factor to be sent.

The algorithm in Figure 4 receives the *LEFs*, a list of transition nodes representing the PN to simulate for the simulation engine, and the limit of the virtual time to be simulated. *EL* contains enabled transitions. *FUL* contains *Future Updating Nodes* (FUNs), and the function *insert-FUL()* maintains them ordered by time. *FUL* plays the role of the Future Event List in an event-driven simulation algorithm for DES. A FUN holds: a pointer (*pt*) to the transition to be updated, the updating factor $UF(t' \to t)$ delivered by each fired transition ($t \in (t'^\bullet)^\bullet$), and the time (*time*) at which the updating must take effect. *head-FUL* is a pointer to *FUL*, *pop(FUL)* pops and returns the head of *FUL*, and we access the fields of *FUN*s using the dot notation. The variable *clock* holds the current simulation time. Figure 3 shows transition nodes in the previously presented PN.

Observe that the interpreter immediately applies *IUF* updating factors, which represents removing tokens from previous places, once a transition occurs, but insert events in *PUL*, which represent that tokens will be appear in posterior places at future clock time. It is important to note also that the interpreter takes all enabled transitions in the *EL* in order, solving in this way conflicts. This execution policy avoids the state with tokens simultaneously in place `A` and `D`, which is a possible state. A random number of enabled transitions can be taken in each interpretation cycle, or updating factors in *IUL* and *PUL* can be atomically applied, representing and atomic occurrence of transitions. This alternative implementations suppose alternative execution policies that can result in different executions. To avoid it, beside the execution policy specification, it is required to identify transitions in conflict and the policy to solve them.

Finally, it is important to point out that the execution model based on the LEF mechanism makes unnecessary the representation and updating of the marking of the PN model. Nevertheless, the construction of the marking of the PN after the occurrence of a sequence of transitions can be easily done collecting

```
1: procedure SIMULATE(Lefs, EL, simulationTime)
2:     VT ← 0; FUL ← {};
3:     for all (t' ∈ EL) do                                   ▷ Fires enabled transitions
4:         if (f_t'(M) ≤ 0) then                              ▷ Checks transition is enabled yet
5:             for all (t ∈ IUL(t')) do
6:                 f_t(M) ← f_t(M) + UF(t' → t);
7:                 if (t = t' and f_t(M) ≤ 0) then            ▷ Avoids race conditions
8:                     insert-FUL (t, 0, τ(t) + clock);
9:                 end if
10:            end for
11:            for all (t ∈ PUL(t')) do
12:                insert-FUL (t, UF(t' → t), τ(t') + clock);
13:            end for
14:        end if
15:    end for
16:    if (head-FUL.time > clock) then VT ← head-FUL.time     ▷ Update Virtual Time
17:    end if
18:    while (head-FUL.time = VT) do                          ▷ Update Event List
19:        t ← head-FUL.pt; f_t(M) := f_t(M) + head-FUL.UF;
20:        if (f_t(M) ≤ 0) then insert(EL, t);
21:        end if
22:        head-FUL ← pop(FUL);
23:    end while
24: end procedure
```

Fig. 4: Centralized simulator engine.

a log containing the occurrence of transitions each one labelled with the simulation time. From this log of labelled transitions, the occurrence sequence can be reconstructed and then using the net state equation (an algebraic computation), for example, compute the reached marking from the initial one.

4 A framework for distributed Simulations of DESs

Distributed simulation of TPNs will be based on many identical simulation engines distributed over the execution platform, and each one devoted to the simulation of a subnet of the original one. Each subnet is represented in the corresponding simulation engine as a data structure. In [24], K.S. Perumalla points out the need of micro-kernels specialized in simulation for building distributed simulations. The idea is to have a micro-kernel that collects the core invariant portion of distributed DES simulation techniques, and avoids to develop entirely the systems from scratch. The core must permit traditional implementations (conservative or optimistic), and the incorporation of newer techniques. We will call *SimBots* to our micro-kernels implementing LP.

The simulation engine proposed by Chiola and Ferscha in [6] manage subnet regions as a subset of places, transitions, and arcs of the original net. This implies that the interface is defined by a subset of places. Moreover, in [6] conservative and optimistic approaches assume a communication channel for each arc connecting the corresponding TPN regions. Therefore, a dynamic workload balancing would require a continuous interface redefinition configured by different channels, which is a very big problem. The availability of a scalable architecture for large-scale simulations requires and event driven execution model. The

actor model based on asynchronous message passing has been selected for the design of large scale distributed simulations as unit of concurrency [16]. It is an event driven model that scales to a large number of actors and removes the complexity of locking mechanisms. A single immutable interface that consists of a mailbox that buffers incoming messages, and a pattern based selection of messages to process them provides the flexibility for configuring different partitions. A distributed simulation based on the LEF mechanism requires a set of simple simulation engines called *SimBot* that each one can be considered as an actor. A *SimBot* will have a mailbox interface, an execution kernel based on the LEF mechanisms, and where is possible to implement different services: synchronization strategies, PN interpretations, load balancing redistributing transition nodes, and self-configuring partition strategies according to the state of computational and network resources.

A *SimBots*' system can be seen in Figure 5. Transition nodes configure a network that processes events triggering updating factors of adjacent transitions when the guard representing the counter equals or less than zero. To distribute the network, it is only required to route messages to the SimBot that contains the corresponding transition nodes. It can be easily done with a *transition service name*, or using routing services supported by actor models such as Akka. Figure 5 shows in the wall clock time axis how evolves the system. Initially the *Simulator System* receives a textual specification of a flat PN, and it is sent to the PNcompiler to obtain the *LEF data structure*. Figure 2 shows how an initial PN specification can be split in subnets. An initial criteria can be as simple as avoid the distribution of transitions in conflict, which can be obtained by structural analysis and to balance the number of transitions in subnets. The *monitoring & load balancing* actor deploys compiled code and monitors the simulation state.

Once the code is deployed, *SimBots* can interchange asynchronous messages with time-stamped updating factors. Each *Simbot* execute the same strategy, incorporating events of adjacent SimBots to the interpretation loop. The *Monitoring & load balancing* actor can recover logs (list of time-stamped triggered transitions). By joining and ordering events it can be obtained a global consistent state and it is possible to monitor the simulation. The bottom part of the figure shows how the system can perform workload balancing as the result of a self-configuration of adjacent actors. The compilation process also incorporates structural information to know adjacent actors, that is, *SimBots* that send or receive updating factors. In the load balancing process, the *SimBot* must be sychronized with adjacent *SimBots* until the LEF data structure corresponding to transitions can be moved. The set of transitions hold by a *SimBot* can be split to distribute it between adjacent *SimBots*, or can be joined in a *SimBot* resulting in a inactive one if there is not transitions to deal.

5 Conclusions and Future Work

A process to fill the gap between high level specification of complex DESs and the generation of code for scalable and dynamic distributed simulations has been

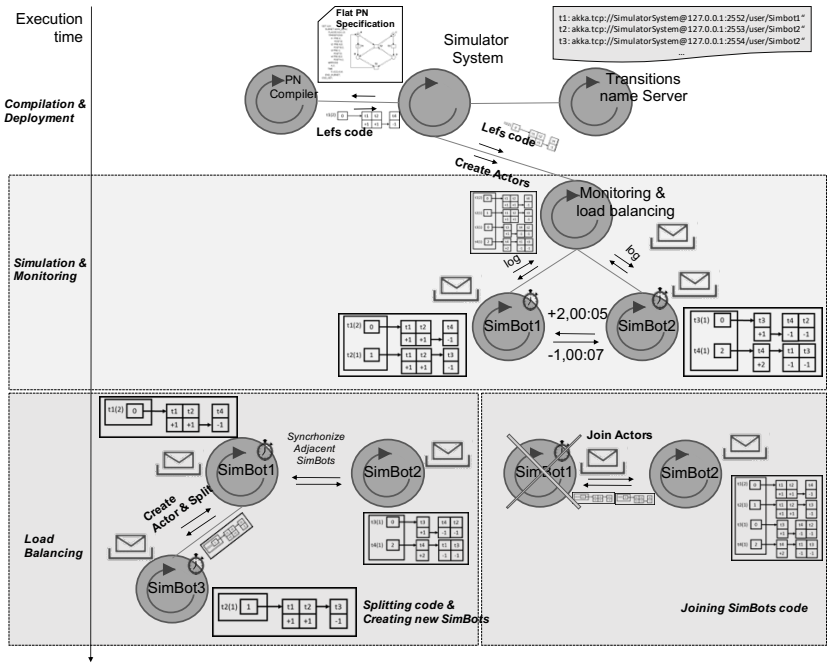


Fig. 5: An actor-based architecture for distributed PN simulations.

presented. The process is based on the well known formalism of PNs, and it is presented an efficient representation for its interpretation. The codification lacks of state representation and makes easy load balancing between interpreters. From the performance point of view, the simulation technique of PNs presented here is the method *Enabled Transitions* (ET) introduced in [25], where places are all 1-bounded. In this paper, it is shown that in a centralized environment ET is better than other simulation techniques when the number of processes grows above a threshold. If we are concerned with distributed simulations, the cost of interchanged messages must be also considered. In general, the transmission of constants after transition firing must be considered in all methods, but in our method we don't need messages related to the maintenance of a global and consistent state of the full model required by other methods.

An actor architecture for distributed simulation of PNs has been also presented. Currently a prototype has been developed in Akka, with a compiler for simple binary PNs, and a basic *SimBot* actor able to interpret and transfer LEF data structures to adjacent *SimBots*. The execution model presented here and the mechanisms for its distributed implementation are the core for the development of any simulation strategy, e.g. conservative, optimistic, etc. It will be the basis for the exploration of new synchronization algorithms, self-configuring policies, and the definition of complex partition criteria considering economical aspect that will allow the development of Simulation as a Service.

Acknowledgments Work financed by the Aragonese Government and the European Regional Development Fund "Construyendo Europa desde Aragón".

References

1. Ammar, H.H., Deng, S.: Parallel simulation of stochastic petri nets using spatial decomposition. In: 1991., IEEE Int. Symposium on Circuits and Systems. pp. 826–829 vol.2 (Jun 1991)
2. Boukerche, A.: An adaptive partitioning algorithm for distributed discrete event simulation systems. J. Parallel Distrib. Com. 62(9), 1454–1475 (2002)
3. Briz, J.L., Colom, J.M.: Implementation of weighted place/transition nets based on linear enabling functions. In: International Conference on Application and Theory of Petri Nets. pp. 99–118. Springer (1994)
4. Byrne, J., Svorobej, S., Giannoutakis, K.M., Tzovaras, D., Byrne, P.J., Östberg, P.O., Gourinovitch, A., Lynn, T.: A review of cloud computing simulation platforms and related environments. In: CLOSER (2017)
5. Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A.F., Buyya, R.: Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw. Pract. Exper. 41(1), 23–50 (Jan 2011)
6. Chiola, G., Ferscha, A.: Distributed simulation of petri nets. IEEE Concurrency (3), 33–50 (1993)
7. D'Angelo, G., Marzolla, M.: New trends in parallel and distributed simulation: From many-cores to cloud computing. Simul. Model. Prac. Theory 49, 320–335 (2014)
8. De Grande, R.E., Boukerche, A.: Dynamic balancing of communication and computation load for HLA-based simulations on large-scale distributed systems. J. Parallel Distrib. Com. 71(1), 40–52 (2011)
9. Djemame, K., Gilles, D.C., Mackenzie, L.M., Bettaz, M.: Performance comparison of high-level algebraic nets distributed simulation protocols. Journal of Systems Architecture 44(6-7), 457–472 (1998)
10. Fujimoto, R.M., Bagrodia, R., Bryant, R.E., Chandy, K.M., Jefferson, D., Misra, J., Nicol, D., Unger, B.: Parallel discrete event simulation: The making of a field. In: 2017 Winter Simulation Conference (WSC). pp. 262–291 (Dec 2017)
11. Fujimoto, R.M., Perumalla, K., Park, A., Wu, H., Ammar, M.H., Riley, G.F.: Large-scale network simulation: how big? how fast? In: Proc. MASCOTS 2003. 11th IEEE/ACM Int. Symp. Modeling, Analysis and Simulation of Computer Telecommunications Systems. pp. 116–123 (Oct 2003)
12. Fujimoto, R., Bock, C., Chen, W., Page, E., Panchal, J.: Research Challenges in Modeling and Simulation for Engineering Complex Systems. Simulation Foundations, Methods and Applications Series, Springer International Publishing (2017)
13. Fujimoto, R., Park, A., Huang, J.C.: Towards flexible, reliable, high throughput parallel discrete event simulations. In: Recent Advances in Modeling and Simulation Tools for Communication Networks and Services, pp. 257–278. Springer (2007)
14. Fujimoto, R.M., Malik, A.W., Park, A.: Parallel and distributed simulation in the cloud. SCS M&S Magazine 3, 1–10 (2010)
15. García, F., Villarroel, J.: Decentralized implementation of real-time systems using time petri nets. application to mobile robot control. IFAC Proceedings Volumes 31(4), 11–16 (1998)

16. Haller, P.: On the integration of the actor model in mainstream technologies: The scala perspective. In: Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions. pp. 1–6. AGERE! 2012, ACM, New York, NY, USA (2012)
17. Malik, A., Park, A., Fujimoto, R.: Optimistic synchronization of parallel simulations in cloud computing environments. In: IEEE International Conference on Cloud Computing, 2009. CLOUD'09. pp. 49–56. IEEE (2009)
18. Marsan, M.A., Balbo, G., Bobbio, A., Chiola, G., Conte, G., Cumani, A.: The effect of execution policies on the semantics and analysis of stochastic petri nets. IEEE Trans. Soft. Eng. 15(7), 832–846 (Jul 1989)
19. Medel, V., Arronategui, U., Bañares, J.Á., Colom, J.M.: Distributed simulation of complex and scalable systems: From models to the cloud. In: GECON'16. pp. 304–318. Springer (2016)
20. Merino, A., Tolosana-Calasanz, R., Bañares, J.Á., Colom, J.M.: A specification language for performance and economical analysis of short term data intensive energy management services. In: GECON'15. pp. 147–163. Cham (2016)
21. Moreno, R.P., Tardioli, D., Salcedo, J.L.V.: Distributed implementation of discrete event control systems based on petri nets. In: Proc. IEEE Int. Symp. Industrial Electronics. pp. 1738–1745 (Jun 2008)
22. Muro-Medrano, P.R., Bañares, J.A., Villarroel, J.L.: Knowledge representation-oriented nets for discrete event system applications. IEEE Trans. Systems, Man, and Cybernetics, Part A 28(2), 183–198 (1998)
23. Nicol, D.M., Mao, W.: Automated parallelization of timed petri-net simulations. J. Parallel Distrib. Com. 29(1), 60–74 (1995)
24. Perumalla, K.S.: μsik - a micro-kernel for parallel/distributed simulation systems. In: Workshop on Principles of Advanced and Distributed Simulation (PADS'05). pp. 59–68 (June 2005)
25. Piedrafita, R., Villarroel, J.L.: Performance evaluation of petri nets centralized implementation. the execution time controller. Discrete Event Dynamic Systems 21(2), 139–169 (Jun 2011)
26. Schriber, T.J., Brunner, D.T., Smith, J.S.: How discrete-event simulation software works and why it matters. In: Proceedings of the Winter Simulation Conference. pp. 3:1–3:15. WSC '12, Winter Simulation Conference (2012)
27. Shekhar, S.e.a.: A simulation as a service cloud middleware. Annals of Telecommunications 71(3), 93–108 (2015)
28. Thomas, G.S., Zahorjan, J.: Parallel simulation of performance petri nets: extending the domain of parallel simulation. In: 1991 Winter Simulation Conference Proceedings. pp. 564–573 (Dec 1991)
29. Tolk, A.: Engineering Principles of Combat Modeling and Distributed Simulation. Wiley Publishing, 1st edn. (2012)
30. Tolosana-Calasanz, R., Bañares, J.Á., Colom, J.M.: Model-driven development of data intensive applications over cloud resources. Futur. Gener. Comp. Syst. (2018)
31. Topçu, O., Durak, U., Oğuztüzün, H., Yilmaz, L.: Distributed Simulation: A Model-Driven Engineering Approach. Simulation Foundations, Methods and Applications, Springer International Publishing (2016)
32. Zehe, D., Knoll, A., Cai, W., Aydt, H.: SEMSim cloud service: Large-scale urban systems simulation in the cloud. Simulation Modelling Practice and Theory 58, 157 – 171 (2015)
33. Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems. Academic press (2000)