Towards an Architecture Proposal for Federation of Distributed DES simulators

Unai Arronategui, José Ángel Bañares, and José Manuel Colom

Aragón Institute of Engineering Research (I3A)
University of Zaragoza, Zaragoza, Spain
{unai,banares,jm}@unizar.es

Abstract. The simulation of large and complex Discrete Event Systems (DESs) increasingly imposes more demanding and urgent requirements on two aspects accepted as critical: (1) Intensive use of models of the simulated system that can be exploited in all phases of its life cycle where simulation can be used, and methodologies for these purposes; (2) Adaptation of simulation techniques to HPC infrastructures, as a method to improve simulation efficiency and to have scalable simulation environments. This paper proposes a Model Driven Engineering approach (MDE) based on Petri Nets (PNs) as formal model. This approach proposes a domain specific language based on modular PNs from which efficient distributed simulation code is generated in an automatic way. The distributed simulator is constructed over generic simulation engines of PNs, each one containing a data structure representing a piece of net and its simulation state. The simulation engine is called *simbot* and versions of it are available for different platforms. The proposed architecture allows, in an efficient way, a dynamic load balancing of the simulation work because the moving of PN pieces can be realized by moving a small number of integers representing the subnet and its state.

Keywords: Simulation Federation · Distributed simulation · Dynamic load balancing.

1 Introduction

Complex systems require large scale simulations that can be very demanding in terms of computational resources. This requirement has produced a growing interest in the use of Cloud for distributed simulation. Moreover, the proliferation of IoT devices for sensing real world and connecting the physical and digital world have broaden the interest in that is called *pervasive*, or *ad hoc* distributed simulation [12]. It promotes the extensive use of simulation for closing the loop in control systems that reacts to changes in the environment.

Economic principles guide the conception and management of these complex systems when they are analyzed under the perspective of Resource Allocation System (RAS). RAS are discrete event systems in which a finite set of concurrent processes share in a competitive way a finite set of resources. Improving

the management of the own resources to support more efficient services with less cost, and to promote the interoperability between organizations for sharing resources and services is the object of study in different domains such as logistic, manufacturing, healthcare system, or cloud computing. It is essential to support decision making and providing high quality of services [17]. The synergic combination of simulation and formal models for functional, performance, and economical analysis are necessary for an efficient an reliable design and/or optimization.

In DES the model evolution happens at discrete points in time by means of simulation events. Large scale systems require distributed simulation to speedup the execution, and to federate the system simulator with other simulators specialized in different aspects interacting with the system under study such as users, external enviroments, or simply others well studied systems already running. A distributed DES simulation is performed through the partition of the simulation model in a set of logical processes (LPs) that interact exchanging time-stamped messages. Each LP ensures that all its internal events are processed in time stamp order.

However, important challenges has hampered the extensive use of distributed simulations, and therefore, the use of cloud computing by the simulation community [12]. Beyond an efficient management of computational resources for a distributed simulation, the modelling is the most costly task [6]. Most of the cost of developing a distributed simulation deals with the time required in specifying, trying it out, and tuning the simulation.

This paper continues our previous work on distributed simulation of discrete event systems [2] focusing on an holistic vision of the problem considering all facets that must be considered. The paper focuses on the role of languages in a MDE approach, proposes a micro-kernel providing services for distributed simulations, presents the algorithms for an efficient distributed interpretation of TPN models, and shows the architecture to federate a micro-kernel's system with other simulator engines and the environment.

2 Related work

Cloud Federation purpose is the interconnection of cloud computing environments of two or more service providers to increase their market share and provide a more efficient management of their resources by collectively load balancing traffic and accommodating spikes in demand [14]. Current solutions provides a seamless exploitation of heterogeneous distributed resources, and brokerage solutions to find the most suitable resource to run an application [5]. Using higher levels of abstraction, such as software as a services (SaaS), supposes a different perspective of federation based on the interoperability or ability of SaaS systems on one cloud provider to communicate with SaaS systems on another cloud provider [19]. At this level of abstraction, the focus is on functional aspects, reusing developed functionality, and the efficiency of resource management

is hidden to the developer. Additionally, semantic interoperability is the most important barrier to the adoption of SaaS systems in cloud computing.

Simulation Federations supposes a pragmatic approach to promote reusability and solve semantic interoperability in the domain of distributed simulations. Try to solve semantic interoperability for SaaS in general is an ambitious tasks. The High Level Architecture (HLA) is an architecture framework for distributed simulation that solves the interoperability and reusability of heterogeneous simulations [21]. A federated simulation conforms to the HLA standard and implements the interfaces specified in the standard to participate in a distributed simulation execution. To solve semantic interoperability, all federated simulations share a common specification of data communication. The federation object model (FOM) specifies object attributes and interactions, and during the simulation all joined federated shall interact with a broker using a Publish/Subscribe Pattern. However, computational resources are hidden to the programmer, and the HLA framework does not provide any mechanism to prevent imbalances. Federation migration become a fundamental mechanism for large-scale distributed simulations [3].

Distributed simulation is a consolidated discipline that faces unprecedented levels of complexity and scale in many fields [11]. Current challenges are presented in [12], which include the analysis of conservative and optimistic strategies in the cloud that has been the focus of recent works. Among the most important challenges to translate distributed simulation to the cloud is the definition of modelling languages that can be easily translated to efficient parallel and distributed simulation code. The purpose of a MDE approach is to model at the higher level of abstraction to increase productivity, and the role and semantic of languages used for modelling and supporting the MDE approach are relevant [21]. The strategy is to model the application with domain-specific languages (DSL). The use of formal models can play an important role in MDE approaches, and PN has shown to be a suitable formalism for specifying DESs. PN has been applied to different domains, providing different level of abstractions for modelling domains such as workflows, business processes, manufacturing, health systems or communication networks. The possibility to automatize the analysis using software tools has been extensively used for proposing good partitioning algorithms and estimate the lookahead in distributed simulations [10, 16, 9].

3 Language-based view of MDE for developing distributed simulation applications

An holistic methodological approach based on formal models for the development of applications over cloud resources was presented in [20]. This MDE approach manages 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. In the case of DES on the cloud, sharing resources implies interferences caused by the limited isolation of

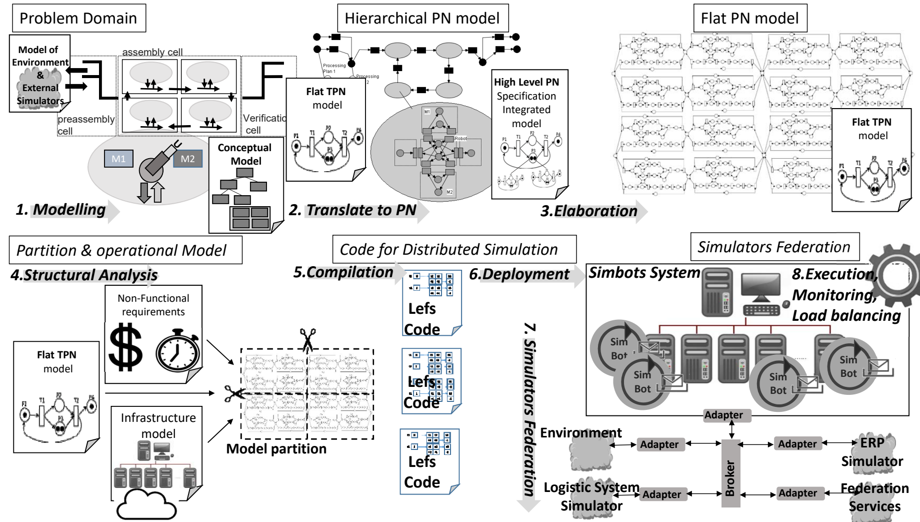

Fig. 1: MDE approach for developing a distributed simulation applications.

virtualization technologies, and with high coupled components it is clear that the execution speed is bounded by the slowest component. The impact of performance variability of resources and the incorporation of mechanism for load balancing are essential in the case of distributed simulation on the cloud. Figure 1 shows the stages of the MDE approach presented in [2]: 1) **Modelling** with a DSL language that provides the basic, usually graphical, primitives/modules that composes an application in the specific domain. Interactions with the environment or external simulators are also modelled. 2)The **modular** construction gives rise to a *hierarchical PN* model. 3) An **elaboration** process translates this high level PN specifications into a *flat model*, 4) The structural analysis of the flat model in combination with an utility function, which combines the speedup of simulations and the cost of computational resources, provides an initial **partition** of the model, 5) Model partitions are **compiled** into efficient code based on the idea of linear enabling function (LEF), 6) *Partitions* are deployed in the **system of simbots**, and finally, 7) in the case of interaction with the environment or other simulators, the simbots system is **federated** with them.

Focusing on the semantics aspects of a language for simulation, the translation of model specifications to meaningful distributed code must preserve the behaviour. Leaving out aspects of hierarchy, composition, or abstraction levels, the dimensions that should be considered in a system are the static structure and the dynamic behaviour. These aspects has been traditionally considered as separated models: static models represent concepts, attributes, relations, and conceptual hierarchies such as UML class diagrams; and dynamic models are presented specifying sequence of actions (workflows), transitions systems (statecharts), and protocols of interaction using events, states, and transition states.

An important research work has been developed trying to integrate models that represent these facets of the system [1, 13].

Our main hypothesis to define our design principles for DES simulation applications are the following: **(1)** We propose only **events**, and **event dependencies** as the minimum required to represent and manage the system behaviour. The identification of a minimal set of basic primitives/concepts to represent the time flow mechanisms that control the generation of a models behavior over time will facilitate the interoperability of simulators and the minimum information required that must be migrated to support the load balancing of simulation work. **(2)** A **model execution** based on its interpretation separates the model specification from the simulator, which is essential for scaling simulations [22]: The model is not wired with the simulator, which enables the portability of the model to other simulators and rise interoperability at a high level of abstraction. Additionally, balancing load works can be facilitated between simulators interpreting the same simulation code. **(3)** Algorithms and methods from distributed programming techniques can be integrated independently of the model interpreter, which facilitates **reusing models** and federation of simulators. **(4)** Dependencies and structural information in combination with event logs (**event sourcing** [8]) are relevant for qualitative and quantitative analysis. An analysis of these structural information is relevant for developing an initial partition, and evaluating the number of resources required to simulate a model in an estimated time, and the cost related with these decisions. However, workload varies in time during simulations. It is required combining structural information with the monitoring and recording of every state change as events in an event log.

4 Simbots: Distributed simulation micro-kernels

In order not to be locked-in specific simulation services to be able to use heterogeneous cloud infrastructures, and even embed these simulation services into IoT devices, it is needed a core invariant portion of DES simulation services that can be executed in heterogeneous devices. The use of micro-kernels specialized in simulation avoids to develop entirely the systems from scratch [18]. We will call *simbots* to our micro-kernels implementing LPs. A simbot is an *actor* defined as an lightweight process that communicates with other simbots through message passing. The actor model was originally constructed for distributed computations and has well-known successful implementations such as the Erlang language, and more recently frameworks like the Akka event-driven middleware [15]. The success of the actor model to afford scalability is the lacks of shared memory between actors, which only interact by means of asynchronous immutable messages. Actors are isolated from each another and are thread safe.

The **architecture of a simbot** is presented in Figure 2. Messages are sent asynchronously to the simbot's **mailbox**, and these are retrieved from the mailbox with a receive statement or pattern-matching construction that filter events, control messages and LEFs code received from other simbots. The **Communication Interface** (CI) ensures that internal or external events of the simbot

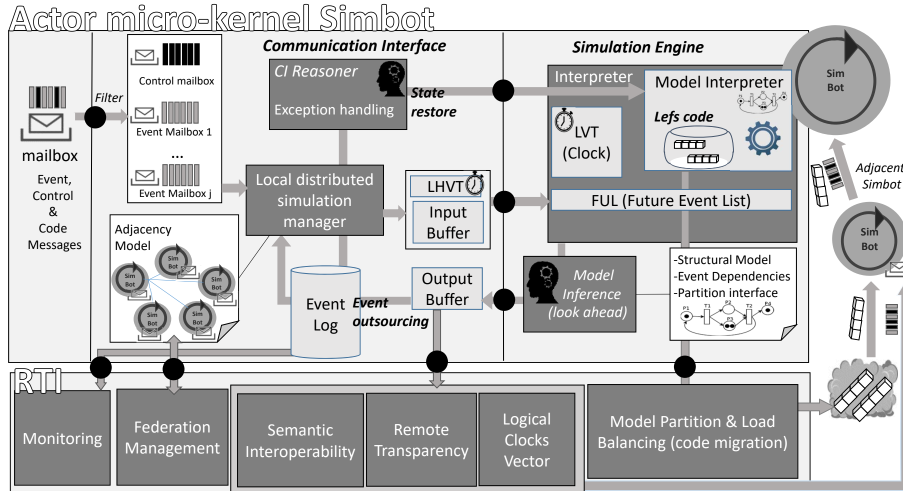

Fig. 2: Simbot: Architecture of a micro-kernel for distributed simulations.

are processed in time stamp order. Section 4.1 explains the synchronization algorithm in detail. In the figure, we can observe that the *local simulation manager* defines a logical horizontal virtual time ($LVHT$) and feeds the simulation engine with the events received from neighbour simbots, executing a simulation step with the local simulation engine until the local virtual time (LVT) reaches the $LVHT$. Every internal and external event is stored in order in an *event log* using a pattern called *event sourcing*, which allows to restore the state from disk on failure recovery, such as the case of an out of order event received. The *CI reasoner monitors* the simbot and orchestrates the recovery of a failure, a change in the adjacent topology, or a load balancing of code with neighbour simbots.

The **Simulation Engine** interprets the model. Figure 5 presents the algorithm to interpret a TPN model represented with LEFs code. The interpreter can be replaced by another interpreter simulating different models using the same interface between the *CI* and the interpreter.

Different dynamic load balancing approaches have been proposed for distributed simulation [7]. The main objective is to minimize the delay generated by redistributing the load and migrating the code of a federate to a destination physical resource. LEFs code facilitates the migration of code between simbots with lightweight messages. Load balancing services based on the movement of code between interpreters is only possible if they use the same codification. It introduces different levels of federation with different layers of interoperability. Finally, the interpreter has a *model inference reasoner* that use the structural information to calculate the lookahead to allow neighbour simbots to advance their LVTs. It also uses the PN state equation (an algebraic computation) to compute , in an efficient way, a restored state in case of failure recovery.

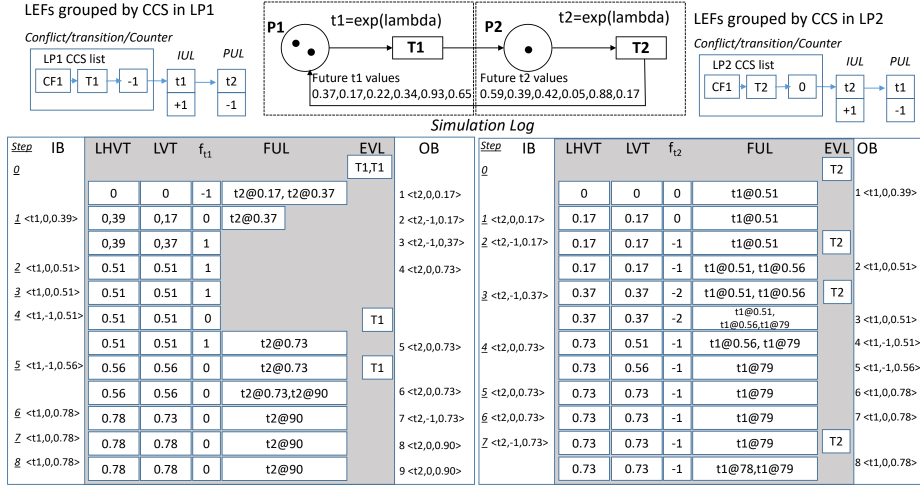

Fig. 3: Distributed simulation log.

The bottom of Figure 2 presents a middleware, the **Runtime Interface** (*RTI*) in HLA terminology, providing the services required for supporting different levels of simulation service federation. The *federation management component* knows when federates join or leave the federation, and manages the way to keep the system running when the topology of the federation changes. Following a multi-layered federation scheme, in the case of a simbot system the *remote transparency* component, manages pair-wise communications directly between pairs of Simbots. Every Simbot and transition has a virtual address. If for example, the simbot system participates as a federate in an HLA federation, the *Semantic Interoperability* component translates the simbots events and the enabling of transitions into HLA object attributes and interactions . The *Federation Management* component deals with the HLA Federation, and the *Remote Transparency* components interchange messages by means of the event bus of an HLA middleware following the publish-subscribe pattern (see Figure 6). Finally, the *Model-Partition* and *monitoring components* support the mechanism for load balancing between simbots , and can participate in the migration of simulation code to physical resources in a federation. The *Logical Clocks Vector* component allows to capture order and causal precedence of events to orchestrate the simulation and the management of control events in the case of failure or code migration.

4.1 Distributed interpretation of a LEF code

Compilation translates a transition in Timed Place/Transition nets specification into an event dependency network based on the idea of ***Linear Enabling Function (LEF)*** [4]. A LEF allows to characterize when a transition is enabled

(an event can occur) with a simple linear function fo the marking. A LEF of a transition t is a function $f_t : \mathbf{R}(\mathcal{N}, \mathbf{m_0}) \longrightarrow \mathbb{Z}$, that 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 $T1$ in the net of Figure 3, the LEF is: $f_{T1}(\mathbf{m}) = 1 - (\mathbf{m}[P1]), \forall \mathbf{m} \in \mathbf{R}(\mathcal{N}, \mathbf{m_0})$, where $\mathbf{m_0}$ is the initial marking depicted. Observe that at $\mathbf{m_0}$, the value of $f_{T1}(\mathbf{m_0}) = -1 <= 0$ and $f_{T2}(\mathbf{m_0}) = 0 <= 0$, i.e. both transitions are enabled at $\mathbf{m_0}$. More details in [2]. A LEF codification translate the specification to code with a **minimal workload** that can be migrated to support the load balancing of simulation.

To explain a distributed simulation based in the interpretation of LEFs, we reproduce the TPN example presented by Ferscha in [10], where $T1$ represents the occurrence of a *machine failure* event, and $T2$ the repair rate. Figure 3 shows the codification associated to each transition, t': (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 List** ($IUL(t')$). Set of transitions whose LEFs must be updated after the occurrence of t' containing the corresponding Updating Factor to be sent; and (5) **Projected Updating List** ($PUL(t')$). Set of transitions whose LEFs must be updated after the occurrence of t' containing the corresponding Updating Factor to be sent. The firing of transitions represents internal events of the simulation engine, and using the event dependency information, it is possible to update the enabling of internal transitions and external transitions. The distributed simulation is coordinated by means of the interchange between simbots of time-stamped messages, which represents how to update the enabling of external transitions when a transition fires.

To exploit parallelism, the model example is partitioned in two LPs: LP_1 and LP_2. *Future Lists* in each partition represent respectively the sequence of exponentially distributed random times ($exp(\lambda = 0.5)$ for $T1$ and $T2$. The bottom of Figure 3 shows the logs (events recording) of a conservative distributed simulation. Gray boxes shows the variables of the simulation interpreters of distributed simbots executing LP1, and LP2, and white sides shows the input and output buffers, and the steps executed by the respective distributed simulation managers. Figure 4 shows a conservative distributed *simulation manager* algorithm sketch that invokes the interpreter algorithm, and Figure 5 shows the algorithm that implements a step of the *simulation interpreter*. The *simulation interpreter* is executed until the LVT reaches the $LHVT$.

For simplicity, the **simulation manager** in Figure 4 shows only the reactive behaviour when an event is received in an actor-like style. The **Start** message (`line 1-10`) initializes the *simulation manager*. It initializes the Event List (EVL) that contains the initial list of enabled internal transitions, it also initializes to zero all time stamps received from adjacent simbots (`Adj`), and it

```
1: when Start() is received                                   ▷ Initialize Simbot
2: VT ← 0; FUL ← {};
3: for all (t ∈ PUL_ext) do
4:     Adj[t] ← 0;
5:     t ! < 0, lookahead(t) >
6: end for
7: for all (t ∈ LEFs) do
8:     if (f_t(M) ≤ 0) then insert(EL, t);
9:     end if
10: end for
11:
12: when Event(t,UF,ts) is received                           ▷ Event Received
13: Adj[t] ← ts;
14: insert-FUL (t, UF, ts);
15: if allReceived(Adj) then
16:     LVTH=min(Adj);
17:     Simulate(ts);
18:     for all (t ∈ PUL_ext) do
19:         if (t ∈ FUL) then
20:             t ! < UF, ts >; remove-FUL(t);
21:         else
22:             t ! < 0, lookahead(t) >;
23:         end if
24:     end for
25: end if
```

Fig. 4: Generic algorithm sketch of the distributed simulation manager.

sends the lookahead value to each transition in PUL_{ext}, which contains the list of transitions in adjacent simbots that can be affected by transition fires in the simbot.

When an **Event** message is received, the *simulation manager* executes a simulation step of the *simulation interpreter* algorithm in Figure 5. The first step of the *simulation manager* is to update the received time stamp of transition in `Adj` (`line 13`), and it translates the external event of the *Input Buffer* (IB) to the *Future Updating List* (FUL) (`line 14`), which plays the role of the future event list in an event-driven simulation. The function *insert-FUL()* maintains events ordered by time stamp. Every event in the FUL has a pointer to the transition to be updated, the updating factor $UF(t' \rightarrow t)$ delivered by each fired transition ($t \in (t'^{\bullet})^{\bullet}$), and the *time* at which the updating must take effect. In the log shown in the figure, UFs are removed from the FUL to avoid redundant information. Following, the event processing checks that a message has been receive from each adjacent simbot (`allreceived(Adj)`, `line 15`). In this case, the minimum time stamp (`ts, line 16`) is used as $LHVT$ to execute a **simulation step** of the interpreter. After this, messages are inserted in the *Output Buffer* (OB) for each transition in PUL_{ext}. Then, the *simulation manager* empties the OB and sent asynchronous messages to all adjacent Simbots to allow them advance their simulations trusting not to receive messages with smaller timestamp in the future. When transitions in adjacent simbots must be updated, the message contains the $t'^{\bullet}$ identifier, the UF and the ts (`line 20`). In other case, the message contains a zero UF value, and the lookahead time stamp (`line 22`). The algorithm can be improved by reducing the number of messages [11].

```
1:  procedure Simulate(LVHT)
2:    while (LVT <= LVTH) do
3:      if (head-FUL.time > clock) then VT ← head-FUL.time          ▷ Update Virtual Time
4:      end if
5:      while (head-FUL.time = VT) do                               ▷ Update Event List
6:        t ← head-FUL.pt; f_t(M) := f_t(M) + head-FUL.UF;
7:        if (f_t(M) ≤ 0) then insert(EL, t);
8:        end if
9:        head-FUL ← pop(FUL);
10:     end while
11:     EVL ← Sort(EVL,CCS, Strategy);          ▷ prioritizes transitions in conflict int EVL
12:     for all (t' ∈ EL) do                                 ▷ Fires enabled transitions
13:       if (f_t'(M) ≤ 0) then                       ▷ Checks transition is enabled yet
14:         for all (t ∈ IUL(t')) do
15:           f_t(M) ← f_t(M) + UF(t' → t);
16:           if (t = t'andf_t(M) ≤ 0) then                      ▷ Avoids race conditions
17:             insert-FUL (t, 0, τ(t) + clock);
18:           end if
19:         end for
20:         for all (t ∈ PUL(t')) do
21:           insert-FUL (t, UF(t' → t), τ(t') + clock);
22:         end for
23:       end if
24:     end for
25:   end while
26: end procedure
```

Fig. 5: Simulation interpreter of a LEFs-coded TPN

When the *simulation manager* executes a conservative strategy, the model requires to exploit the **lookahead** information to speedup the simulation. The lookahead comes directly from the net structure [10]. To calculate the lookahead for every external transition, the sample use the precomputed *future list* of random firing times. The lookahead of a transition is calculated as the minimum time-stamp of events that reference this transition in the FUL, and the times that result for the addition of the times in the future list and the LVT taking into account a number of times equal to the enabling degree of transition.

The **simulation interpreter** in Figure 5 executes the interpreter until the $LVHT$. First, it advances the LVT until the time stamp of the first event in the FUL (`line 3-4`). Then, the algorithm updates all LEF values with UF in the events of the FUL that has the current LVT, and inserts enabled transitions in the EVL (`lines 5-10`). *head-FUL* is a pointer to FUL, *pop(FUL)* pops and returns the head of FUL, and we access the fields of events in FUL using the dot notation. Following, the algorithm deals with all enabled transitions in *coupled conflict sets (CCS)* (`line 11`), solving conflicts by sorting enabled transitions according to some defined strategy. A CCS is a structural transitive relation that goups transitions that share some previous input place. Then, the interpreter takes all enabled transitions in the EVL firing enabled transitions in order (`lines 12-24`), solving in this way conflicts.

For every enabled transition, the algorithm immediately applies IUF updating factors, which represents removing tokens from previous places, once a transition occurs (`lines 14-19`), but insert events in PUL, which represent that tokens will be appear in posterior places at future clock time (`lines 20-22`).

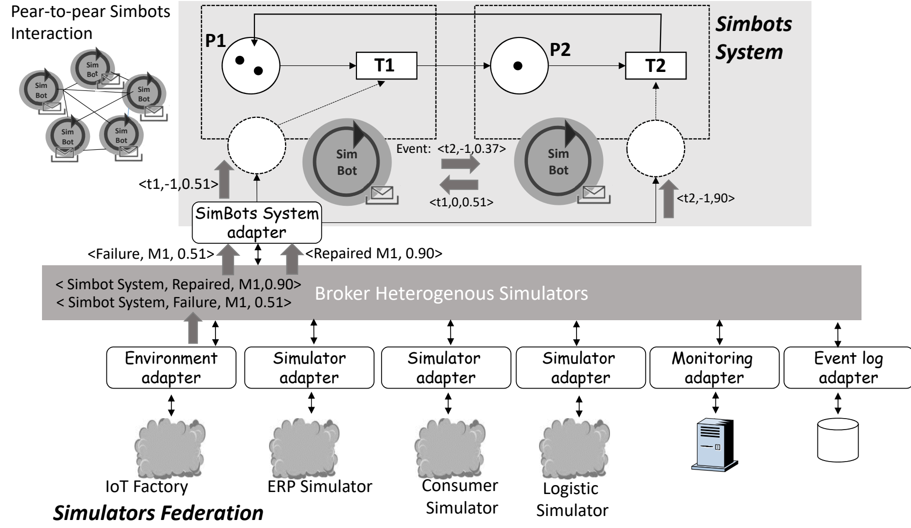

Fig. 6: Architectural approach for federation of heterogeneous simulators.

Then, all LEFs values of transitions in the FUL with the same time-stamp of the current LVT are checked, and enabled transitions are inserted in the EVL. As it was highlight in [2], LEFs make unnecessary the representation and updating of the marking of the PN model, and the construction of the marking of the PN after the occurrence of a sequence of transitions can be easily done collecting a log containing the occurrence of transitions each one labelled with the simulation time. The occurrence sequence and the net state equation (an algebraic computation) can be used to compute the reached marking from the initial one.

Only a mailbox is associated with a simbot, which allow to adapt the algorithm to support dynamic topologies of simbots, or use optimistic strategies incorporating an exception handling mechanism.

5 Architectural approach for federating simulators

The need to build more scalable and interactive simulations considering all involved aspects (devices, humans and environment), require the expertise from several groups to be combined, and to consider more realistic scenarios. It forces to reuse legacy simulation components, and use component with varying degrees of fidelity depending of the required precision of results.

Reusability implies a different architectural approach to have heterogeneous cooperating simulators of Cloud services. Reuse is regarded by many organizations as the top driver for the adoption of SOA, which is fulfilled under the mediation of a brokering structure. This is the underlying idea of the HLA standard for distributed simulation, which provides services for information exchange

and synchronization between simulations that together form a federation. Figure 6 shows the simbots system federated with heterogeneous simulators through a broker, and adapters that translate events and interactions coming from the environment or external simulators to simbot events. Observe that it is required to define the interface with the environment, which is represented in the figure with the dotted places.

Figure 6 presents a layered architecture, with the top layer supported by distributed simulation micro-kernels, which efficiently provide a distribute interpretation of the model using a pear-to-pear interaction, and dynamic load balancing with a minimal workload. The botton layer focus on the reuse under the mediation of a brokering middleware. It opens opportunities for collaborations an alliances at different levels: a more close collaboration to share simulation workload using simbots, and more open collaborations reusing different simulators.

6 Conclusions and Future work

The paper has proposed an architecture to reduce the economic costs of the simulation task for two reason: (1) The use of models in different phases of the lifecycle allow to plan good strategies for the efficient use of resources by means of a previous analysis of the model and to customize the simulation according to the structure of the model to be simulated; (2) The implementation of a distributed simulation to take advantage of the availability of resources, and making an efficient use of the resources by the dynamic partitioning of the model to be simulated.

Moreover, the paper proposed an additional line of economic costs reduction by the connection of several existing simulators running in heterogeneous platforms. This connection try to reduce costs of model construction delegating some parts of the model to those included in existing simulators and interpreting this delegated parts as the environment of our system. To do that in the paper a mechanism for the federation of DES simulators is proposed and integrated in the simbots designed.

A compiler for ordinary Timed PN and a prototype in Akka of the distributed Simbot actor has been developed. The use of ordinary Timed PN in the modeling of large complex DES can lead to models of unmanageable size. Inmediate future work includes the use of high-level models that support modularity and hierarchy, and the implementation of a compiler that explores the top-down design hierarchy and builds an interconnection table until it reaches the building blocks of the design: events, and event dependencies.

The partition of the resulting flat model, the deployment of compiled code, and the development of mechanism to support the monitoring and load balancing redistribution of code between adjacent actors are the inmediate steps to to show experimentally the adequacy of the architectural proposal for an efficient distributed interpretation of the model.

Acknowledgments This work was co-financed by the Aragonese Government and the European Regional Development Fund "Construyendo Europa desde Aragón" (COSMOS research group, ref. T35_17D); and by the Spanish program "Programa estatal del Generación de Conocimiento y Fortalecimiento Científico y Tecnológico del Sistema de I+D+i ", project PGC2018-099815-B-100.

References

1. Agha, G.A., De Cindio, F.: Concurrent object-oriented programming and petri nets: advances in petri nets. No. 2001, Springer Science & Business Media (2001)
2. Bañares, J.Á., Colom, J.M.: Model and simulation engines for distributed simulation of discrete event systems. In: International Conference on the Economics of Grids, Clouds, Systems, and Services. pp. 77–91. Springer (2018)
3. Boukerche, A., Grande, R.E.D.: Optimized federate migration for large-scale hla-based simulations. In: Proc. 12th IEEE/ACM Int. Symp. Distributed Simulation and Real-Time Applications. pp. 227–235 (Oct 2008)
4. 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)
5. Carlini, E., Dazzi, P., Mordacchini, M.: A holistic approach for high-level programming of next-generation data-intensive applications targeting distributed heterogeneous computing environment. Procedia Computer Science 97, 131 – 134 (2016), http://www.sciencedirect.com/science/article/pii/S1877050916321068, 2nd International Conference on Cloud Forward: From Distributed to Complete Computing
6. Chandy, M.K.: Event-driven applications: Costs, benefits and design approaches (2006)
7. 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)
8. Debski, A., Szczepanik, B., Malawski, M., Spahr, S., Muthig, D.: A scalable, reactive architecture for cloud applications. IEEE Software 35(2), 62–71 (2017)
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. Ferscha, A.: Tutorial on parallel and distributed simulation of petri nets. In: Performance Tools95, Heidelberg, Germany, Sept. (1995)
11. 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)
12. Fujimoto, R.M.: Research challenges in parallel and distributed simulation. ACM Trans. Model. Comput. Simul. 26(4), 22:1–22:29 (May 2016)
13. Gómez, A., Merseguer, J., Di Nitto, E., Tamburri, D.A.: Towards a uml profile for data intensive applications. In: proceedings of the 2nd International Workshop on Quality-Aware DevOps. pp. 18–23. ACM (2016)
14. Haile, N., Altmann, J.: Evaluating investments in portability and interoperability between software service platforms. Future Gener. Comput. Syst. 78(P1), 224–241 (Jan 2018)

15. 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)
16. Nicol, D.M., Mao, W.: Automated parallelization of timed petri-net simulations. J. Parallel Distrib. Com. 29(1), 60–74 (1995)
17. Paščinski, U., Trnkoczy, J., Stankovski, V., Cigale, M., Gec, S.: Qos-aware orchestration of network intensive software utilities within software defined data centres. Journal of Grid Computing 16(1), 85–112 (Mar 2018), `https://doi.org/10.1007/s10723-017-9415-1`
18. 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)
19. Rezaei, R., Chiew, T.K., Lee, S.P., Aliee, Z.S.: A semantic interoperability framework for software as a service systems in cloud computing environments. Expert Systems with Applications 41(13), 5751 – 5770 (2014)
20. Tolosana-Calasanz, R., Bañares, J.Á., Colom, J.M.: Model-driven development of data intensive applications over cloud resources. Futur. Gener. Comp. Syst. (2018)
21. Topçu, O., Durak, U., Oğuztüzün, H., Yilmaz, L.: Distributed Simulation: A Model-Driven Engineering Approach. Simulation Foundations, Methods and Applications, Springer International Publishing (2016)
22. Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems. Academic press (2000)