Automated Software Engineering manuscript No.
(will be inserted by the editor)

A Petri Net Tool for Software Performance Estimation Based on Upper Throughput Bounds

Ricardo J. Rodríguez

Received: date / Accepted: date

Abstract Functional and non-functional properties analysis (i.e., dependability, security, or performance) ensures that requirements are fulfilled during the design phase of software systems. However, the Unified Modelling Language (UML), standard *de facto* in industry for software systems modelling, is unsuitable for any kind of analysis but can be tailored for specific analysis purposes through profiling. For instance, the MARTE profile enables to annotate performance data within UML models that can be later transformed to formal models (e.g., Petri nets or Timed Automatas) for performance evaluation. A performance (or throughput) estimation in such models normally relies on a whole exploration of the state space, which becomes unfeasible for large systems. To overcome this issue upper throughput bounds are computed, which provide an approximation to the real system throughput with a good complexity-accuracy trade-off. This paper introduces a tool, named `PeabraiN`, that estimates the performance of software systems via their UML models. To do so, UML models are transformed to Petri nets where performance is estimated based on upper throughput bounds computation. `PeabraiN` also allows to compute other features on Petri nets, such as the computation of upper and lower marking place bounds, and to simulate using an approximate (continuous) method. We show the applicability of `PeabraiN` by evaluating the performance of a building closed circuit TV system.

Keywords Petri net, UML, Software Performance, Optimization

R.J. Rodríguez
Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, María de Luna 1, 50018 Zaragoza, Spain
Tel.: (+34) 876 555531
Fax: (+34) 976 761914
E-mail: rjrodriguez@unizar.es

1 Introduction

Software systems are normally modelled with Unified Modelling Language (UML) OMG (2011b), the standard *de facto* in industry for software modelling. However, UML semantics is ambiguous [Fecher et al (2005)] and lacks for a support of formal validation (i.e., whether the system fulfils the requirements) and verification (i.e., whether the system fulfils internal correctness properties). Usually, verification and validation (V&V) are carried out in a system model focusing on the functional properties but avoiding non-functional properties. Examples of these non-functional properties are dependability, security or performance.

However, the system performance evaluation (or *throughput*, defined as completed jobs per unit of time), that is, to estimate how long the system takes to complete, is a primordial study in certain systems, mainly by its likely trade-off with other non-functional properties. For instance, let us suppose an embedded real-time distributed system of fire sensors deployed along a forest for early prevention of fire disasters. In this case, the system performance becomes crucial: the less the sensor activities take, the less battery consumption and thus a longer sensor useful life. This issue clearly evidences that the sooner the system performance is evaluated, the sooner it can be redesigned to get a better performance and save potential overruns. Note that the cost of redesigning a system in deployment has been quantified to be over the half of the total budget in certain domains [Randimbivololona (2001)].

UML can be tailored for specific purposes through *profiling* [Selic (2007); Lagarde et al (2007)]. A UML profile enriches UML models by extending its semantics. For instance, the Modelling and Analysis for Real-Time Embedded systems (MARTE) [OMG (2011a)] profile provides an analysis framework called Quantitative Analysis Model (GQAM) that enables performance specification in UML models. A UML model with performance annotations (that is, UML enriched with MARTE profile) can be transformed to a formal model where performance evaluation is carried out, such as Stochastic Petri Nets (SPN) [Molloy (1982)]. Petri nets (PN) [Murata (1989)], among other formal models, have been extensively studied in the literature as formal modelling language to evaluate UML models (the reader is referred to [Merseguer et al (2002); López-Grao et al (2004); Woodside et al (2005); Koch and Parisi-Presicce (2006); Bernardi and Merseguer (2007); Berardinelli et al (2009); Distefano et al (2011)], for naming a few).

However, a Petri net model of a software system can be so complex that exact performance evaluation becomes a highly complex computational task. The main reason for this complexity is the well-known state-space explosion problem. As a result, a task that requires an exhaustive state-space exploration becomes unachievable in a reasonable time for large systems. A way to overcome this problem is to estimate the performance by computing performance bounds [Campos and Silva (1992); Liu (1995); Rodríguez and Júlvez (2010); Rodríguez et al (2013)].

The computation of lower and upper throughput bounds gives an estimation about where the real system throughput is contained and can be a good approximation to future system performance improvement plans. This paper introduces an improvement of the tool `PeabraiN` [Rodríguez et al (2012)], which implements previously published algorithms to compute lower and upper throughput bounds [Campos and Silva (1992); Rodríguez and Júlvez (2010); Rodríguez et al (2013)] in a Petri net. These algorithms intensively use linear programming (LP) techniques for which polynomial complexity algorithms exist, so they offer a good trade-off between accuracy and computational complexity.

`PeabraiN` was initially presented in [Rodríguez et al (2012)], where a collection of modules for performance estimation and resource optimisation based on bounds computation for SPNs were introduced, as well as a SPN simulation analysis module based on exact Gillespie's stochastic simulation algorithm Gillespie (1976). This paper introduces the plug-in architecture of `PeabraiN` in more detail, and a complete description of its features. Besides, as case study we evaluate a building closed circuit TV system. The system is first modelled with UML, and annotated with MARTE to specify performance and resources data. Then, it is transformed to Petri net to carry a performance evaluation out using upper throughput bounds. We also show the benefit of applying this kind of computation instead of simulation techniques.

The plug-in architecture of `PeabraiN` allows a modular extension of the tool, which helps to easily enhance the tool features. `PeabraiN` was designed following structural and architectural patterns such as *Facade* and *Model-View-Controller*, respectively. `PeabraiN` includes several enhancements with respect to [Rodríguez et al (2012)], such as the computation of lower and upper bounds for the average marking of places and the computation of structural implicit places. The *Simulator* module has been also improved in several ways: First, it computes the average marking of places; second, it creates an external file of results to make easier the handling after experiments; and third, a new simulation method, an approximate stochastic simulation algorithm (using Tau-Leaping method [Gillespie (2007, 2008)]), has been implemented. Another interesting feature of `PeabraiN` is the command-line execution, which enables the automation of evaluation by using scripts, and integration into batch executions.

Contribution. In brief, the contribution of this paper is three-fold: First, we introduce a Petri net tool whose plug-in architecture enables a rapid prototype implementation of methods that use LP techniques on Petri nets. We exemplify it by extending `PeabraiN` with a module for computing structural implicit places; Second, an approximate simulation method faster than exact simulation is also introduced. Experiments show that this method performs better for nets with large number of resources; and third, we demonstrate how upper throughput bounds computation in a formal model, such as Petri nets,

becomes useful to estimate the performance of a complex software system modelled with UML.

The outline of this paper is as follows. Section 2 reviews the literature about tools for performance evaluation in UML/PN models. Section 3 introduces some preliminary concepts needed to follow the rest of the paper, such as Petri nets, performance estimation based on bounds, and UML profiles. The plug-in architecture of `PeabraiN` and its last enhancements are explained in more detail in Section 4. Section 5 introduces the case study: A UML model annotated with MARTE of a building closed circuit TV system is transformed to a Petri net to estimate its performance using upper throughput bounds. Finally, Section 6 states some conclusions and future work.

2 Related Work

We have reviewed the literature focusing on tools that enable to evaluate performance in UML models or Petri nets and the tools that allow to compute upper/lower throughput bounds in Petri net models.

UML models can be transformed to different formal models, such as Petri nets or other timed automatas. In this work, we focus on tools that transform a UML model to Petri nets [Distefano et al (2011)]. Some of these tools are ArgoPN [Delatour and de Lamotte (2003)], ArgoPerformance [Distefano et al (2011)] or ArgoSPE [Gómez-Martínez and Merseguer (2006)]. The PN models obtained after transformation with these tools can be handled with different PN tools (such as GreatSPN [Baarir et al (2009)] in the case of ArgoSPE) that compute the performance of the model through simulation, which becomes unfeasible for large systems due to the well-known state-space explosion problem. On the contrary, our tool enables not just simulation (exact and approximate methods) but also performance bound computation from which polynomial algorithms exist, thus offering a good complexity-accuracy trade-off.

To the best of our knowledge, the only tool for performance bound computation, is GreatSPN [Baarir et al (2009)], which computes lower and upper throughput bounds of transitions. GreatSPN is a tool for evaluation of performance in Petri nets widely used in the community. However, the programming language paradigm used for its implementation and its platform dependency makes very difficult to extend its functionalities, unlike `PeabraiN`.

MATLAB [The MathWorks (2010)] has also an optimisation package that can be used for LP computations. Nevertheless, it is a proprietary software and it depends of a proprietary software library (namely, MATLAB Component Runtime library) for a final software deployment. Besides, the representation of a PN in MATLAB is done in its mathematical form, what clearly makes the comprehension of the model at a glance harder. The extension of a MATLAB tool is neither not so straightforwardly.

Our tool, which is based on PIPE [Bonet et al (2007)] as we will show in the next section, has several benefits over the related tools: First, it allows an easy extension through modules; second, it uses the standard PN file format,

Petri Net Markup Language (PNML) [Hillah et al (2009)], so it allows an interchange of files between different PNML-compliant tools; third, it facilitates a user-friendly GUI editor; and finally, it is a multi-platform and open source tool, which enables to be executed in different environments and its functionalities can be improved/revised by the community. Finally, it is worth also mentioning that PIPE enables to import standard performance format files, such as Performance Model Interchange Format (PMIF) [Smith et al (2010)]. Therefore, a UML specification can be transformed to PMIF and then imported into PIPE [Lladó and Harrison (2011); Bonet and Lladó (2012)], which clearly extends the usability of this tool.

3 Preliminary Concepts

This section introduces concepts and references for the algorithms implemented in `PeabraiN`, as well as it presents the UML profiles.

3.1 Petri nets

A Petri net [Murata (1989)] is a 4–tuple $\mathcal{N} = \langle P, T, \mathbf{Pre}, \mathbf{Post} \rangle$, where P and T are disjoint non-empty sets of *places* and *transitions*, and **Pre** (**Post**) are the pre–(post–)incidence non-negative integer matrices of size $|P| \times |T|$. The *pre-* and *post-set* of a node $v \in P \cup T$ are respectively defined as ${}^\bullet v = \{u \in P \cup T | (u, v) \in F\}$ and $v^\bullet = \{u \in P \cup T | (v, u) \in F\}$, where $F \subseteq (P \times T) \cup (T \times P)$ is the set of directed arcs. A Petri net is said to be *self-loop free* if $\forall p \in P, t \in T\ t \in {}^\bullet p$ implies $t \notin p^\bullet$. *Ordinary* nets are Petri nets whose arcs have weight 1. The *incidence matrix* of a Petri net is defined as $\mathbf{C} = \mathbf{Post} - \mathbf{Pre}$.

A vector $\mathbf{m} \in \mathbb{Z}_{\geq 0}^{|P|}$ which assigns a non-negative integer to each place is called *marking vector* or *marking*. A *Petri net system*, or *marked Petri net* $\mathcal{S} = \langle \mathcal{N}, \mathbf{m_0} \rangle$, is a Petri net $\mathcal{N}$ with an *initial marking* $\mathbf{m_0}$.

A transition $t \in T$ is *enabled* at marking $\mathbf{m}$ if $\mathbf{m} \geq \mathbf{Pre}(\cdot, t)$, where $\mathbf{Pre}(\cdot, t)$ is the column of **Pre** corresponding to transition t. A transition t enabled at $\mathbf{m}$ can *fire* yielding a new marking $\mathbf{m}' = \mathbf{m} + \mathbf{C}(\cdot, t)$ (*reached* marking). This is denoted by $\mathbf{m} \xrightarrow{t} \mathbf{m}'$. A sequence of transitions $\sigma = \{t_i\}_{i=1}^n$ is a *firing sequence* in $\mathcal{S}$ if there exists a sequence of markings such that $\mathbf{m_0} \xrightarrow{t_1} \mathbf{m}_1 \xrightarrow{t_2} \mathbf{m}_2 \ldots \xrightarrow{t_n} \mathbf{m}_n$. In this case, marking $\mathbf{m}_n$ is said to be *reachable* from $\mathbf{m_0}$ by firing σ, and this is denoted by $\mathbf{m}_0 \xrightarrow{\sigma} \mathbf{m}_n$. The enabling degree of a transition t enabled at a marking $\mathbf{m}_i$, denoted as $\mathbf{e}_{\mathbf{m}_i}(t)$, is the biggest integer number k such that $\mathbf{m}_i \geq k \cdot \mathbf{Pre}(\cdot, t)$. The *firing count vector* $\boldsymbol{\sigma} \in \mathbb{Z}_{\geq 0}^{|T|}$ of the firable sequence σ is a vector such that $\boldsymbol{\sigma}(t)$ represents the number of occurrences of $t \in T$ in σ. If $\mathbf{m_0} \xrightarrow{\sigma} \mathbf{m}$, then we can write in vector form $\mathbf{m} = \mathbf{m_0} + \mathbf{C} \cdot \boldsymbol{\sigma}$, which is referred to as the *linear* (or *fundamental*) *state equation* of the net.

Two transitions t, t' are said to be in *structural conflict* if they share, at least, one input place, i.e., ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$. Two transitions t, t' are said

to be in *effective conflict for a marking* $\mathbf{m}$ if they are in structural conflict and they are both enabled at $\mathbf{m}$. Two transitions t, t' are in *equal conflict* if $\mathbf{Pre}(\cdot, t) = \mathbf{Pre}(\cdot, t') \neq \mathbf{0}$. The equal conflict relation is an equivalence relation that partitions the set of transitions into equivalence classes ECS, called *equal conflict sets*. Transitions belonging to a given equal conflict set are in *extended free-choice conflict*.

A Petri net is said to be *strongly connected* if there is a directed path joining any pair of nodes of the graph. A *state machine* is a particular type of ordinary Petri net where each transition has exactly one input arc and exactly one output arc, that is, $|t^\bullet| = |^\bullet t| = 1, \forall t \in T$.

A *p-semiflow* is a non-negative integer vector $\mathbf{y} \geq \mathbf{0}$ such that it is a left anuller of the net's incidence matrix, $\mathbf{y}^\intercal \cdot \mathbf{C} = 0$. A p-semiflow implies a token conservation law independent from any firing of transitions. A *t-semiflow* is a non-negative integer vector $\mathbf{x} \geq \mathbf{0}$ such that is a right anuller of the net's incidence matrix, $\mathbf{C} \cdot \mathbf{x} = 0$. A p- (or t-)semiflow $\mathbf{v}$ is *minimal* when its support, $\|\mathbf{v}\| = \{i | \mathbf{v}(i) \neq 0\}$, is not a proper superset of the support of any other p- (or t-)semiflow, and the greatest common divisor of its elements is one.

A place $p \in P$ is said to be *implicit* in $\mathcal{S}$ if it never is the unique place to constraint the occurrence of its output transitions. That is, an implicit place p has enough tokens in any reachable marking of $\mathcal{S}$ such that the firing of its output transitions does not depends on marking of p. Thus, its removal does not affect the net system behaviour. A place p is *structurally implicit* in $\mathcal{N}$ if for any initial marking $\mathbf{m_0}$, the initial marking of p can be chosen such that p becomes implicit (i.e., p can be removed from the net without affecting the the net behaviour). Algebraically, a place p is structurally implicit if and only if there exits a p-semiflow $\mathbf{y}$ such that $\mathbf{y} \geq 0, \mathbf{y}(p) = 0$ and $\mathbf{y}^\intercal \cdot \mathbf{C} \leq \mathbf{C}(p, \cdot)$. A structurally implicit place p becomes implicit when its initial marking $\mathbf{m_0}(p) \geq \max(0, z)$, where z is solved by the following LPP [Garcia-Valles and Colom (1999):

$$\begin{aligned} z = \mathit{minimize}\ & \mathbf{y}^\intercal \cdot \mathbf{m_0} + \mu \\ \mathit{subject\ to}\ & \mathbf{y}^\intercal \cdot \mathbf{C} \leq \mathbf{C}(p, \cdot) \\ & \mathbf{y}^\intercal \cdot \mathbf{Pre}(\cdot, t) + \mu \geq \mathbf{Pre}(p, t), \forall t \in p^\bullet \\ & \mathbf{y}(p) = 0 \\ & \mathbf{y} \geq \mathbf{0}, \mu \geq 0 \end{aligned} \tag{1}$$

Definition 1 [Silva and Colom (1988) Let be $\langle \mathcal{N}, \mathbf{m_0} \rangle$ be a net system. The structural marking bound N of a given place p in $\mathcal{N}$ is

$$\begin{aligned} N(p) = \mathit{maximum}\ & \mathbf{m}(p) \\ \mathit{subject\ to}\ & \mathbf{m} = \mathbf{m_0} + \mathbf{C} \cdot \boldsymbol{\sigma} \\ & \mathbf{m} \geq \mathbf{0}, \boldsymbol{\sigma} \geq \mathbf{0} \end{aligned} \tag{2}$$

A *Stochastic Petri Net* system (SPN) is a pair $\langle \mathcal{S}, \delta \rangle$ where $\mathcal{S}$ is a Petri net system and $\delta : T \to \mathbb{R}^+$ is a positive real function such that $\delta(t)$ is the mean

of the exponential firing time distribution associated to each transition $t \in T$. If $\delta(t) > 0$, then transition t is a timed transition. Otherwise, i.e., $\delta(t) = 0$, transition t is an immediate one. It will be assumed that all transitions in conflict are immediate.

The average marking vector, $\overline{\mathbf{m}}$, in an ergodic Petri net system is defined as [Florin and Natkin (1989)]:

$$\overline{\mathbf{m}}(p) \underset{AS}{=} \lim_{\tau \to \infty} \frac{1}{\tau} \int_0^\tau \mathbf{m}(p)_u du \tag{3}$$

where $\mathbf{m}(p)_u$ is the marking of place p at time u and the notation $\underset{AS}{=}$ means *equal almost surely*.

Similarly, the steady-state throughput, χ, in an ergodic Petri net is defined as [Florin and Natkin (1989)]:

$$\chi(t) \underset{AS}{=} \lim_{\tau \to \infty} \frac{\boldsymbol{\sigma}(t)_\tau}{\tau} \tag{4}$$

where $\boldsymbol{\sigma}(t)_\tau$ is the firing count of transition t at time τ.

The vector of visit ratios expresses the relative throughput of transitions in the steady state. The visit ratio $\mathbf{v}(t)$ of each transition $t \in T$ normalised for transition t_i , $\mathbf{v}^{t_i}(t)$, is expressed as $\mathbf{v}^{t_i}(t) = \dfrac{\chi(t)}{\chi(t_i)} = \boldsymbol{\Gamma}(t_i) \cdot \chi(t),\ \forall t \in T$, where $\boldsymbol{\Gamma}(t_i) = \dfrac{1}{\chi(t_i)}$ represents the *average inter-firing time* of transition t_i and $\chi(t)$ is the steady-state throughput of transition t.

In general, the vector of visit ratios $\mathbf{v}$ depends on the structure of the net, on the routing rates, on the initial marking $\mathbf{m_0}$, and on the service times δ [Campos and Silva (1992)]. When the vector of visit ratios depends only on the structure of the net and on the routing rate, the Petri net is a *Freely Related T-semiflows* net (FRT-net) [Campos and Silva (1992)]. The range of FRT-nets is relatively broad. Examples of these kind of nets are mono-T-semiflow nets (i.e., nets having a single T-semiflow), choice-free nets, Process Petri nets [Tricas (2003)], and nets where every conflict is an equal conflict. It is known that the continuous time Markov chain associated to these nets is ergodic [Campos and Silva (1992)], what implies the existence of the above limits.

In a FRT-net, the vector of visit ratios $\mathbf{v}$ normalised for transition t_i, $\mathbf{v}^{t_i}$, can be calculated by solving the following linear system of equations [Campos and Silva (1992)]:

$$\begin{aligned} \begin{pmatrix} \mathbf{C} \\ \mathbf{R} \end{pmatrix} \cdot \mathbf{v}^{t_i} &= 0 \\ \mathbf{v}^{t_i}(t_i) &= 1 \end{aligned} \tag{5}$$

where $\mathbf{R}$ is a matrix containing the rates $\mathbf{r}(t)$ associated to transitions in equal conflict.

3.2 Performance Estimation Based on Bounds

A lower bound for the *average inter-firing time* of transition t_i, $\mathbf{\Gamma^{lb}}(t_i)$, can be computed by solving the following LP problem (LPP) [Campos and Silva (1992)]:

$$\begin{aligned} \mathbf{\Gamma}(t_i) \geq \mathbf{\Gamma^{lb}}(t_i) = \ & \textit{maximum } \mathbf{y^\intercal} \cdot \mathbf{Pre} \cdot \mathbf{D^{t_i}} \\ & \textit{subject to } \mathbf{y^\intercal} \cdot \mathbf{C} = \mathbf{0} \\ & \mathbf{y^\intercal} \cdot \mathbf{m_0} = 1 \\ & \mathbf{y} \geq \mathbf{0} \end{aligned} \tag{6}$$

where $\mathbf{\Gamma}(t_i)$ is the average interfiring time of transition t_i and $\mathbf{D^{t_i}}$ is the vector of *average service demands of transitions*, $\mathbf{D^{t_i}}(t) = \mathbf{s}(t) \cdot \mathbf{v}^{t_i}(t)$ (the vector of visit ratios $\mathbf{v}^{t_i}$ is normalised for transition t_i). In the sequel, we omit the superindex t_i in $\mathbf{D^{t_i}}$ for clarity.

As a side product of the solution of (6), $\mathbf{y}$ represents the *slowest* p-semiflow of the system, thus LPP (6) can also be seen as a search for the most constraining p-semiflow. This p-semiflow will be the one with highest ratio $\dfrac{\mathbf{y^\intercal} \cdot \mathbf{Pre} \cdot \mathbf{D}}{\mathbf{y^\intercal} \cdot \mathbf{m_0}}$. Therefore, an upper bound $\Theta(t_i)$ for the steady-state throughput can be calculated as the inverse of the lower bound for the average inter-firing time $\mathbf{\Gamma^{lb}}(t_i)$, that is, $\Theta(t_i) = \dfrac{1}{\mathbf{\Gamma^{lb}}(t_i)}$.

The LPP (6) can be applied when the vector of visit ratios $\mathbf{v}$ is known, as the computation of the vector of average service demands of transitions $\mathbf{D}$ is related to $\mathbf{v}$. After some manipulations, the LPP (6) can be transformed to its

dual [Chiola et al (1993)], which can be used to compute an upper throughput bound Θ for a transition t_i in a SPN:

$$
\begin{aligned}
\chi(t_i) \leq \Theta(t_i) = \ & \mathit{maximum}\ \chi(t_i) \\
& \mathit{subject\ to}\ \overline{\mathbf{m}} = \mathbf{m_0} + \mathbf{C} \cdot \sigma \\
& \sum_{t \in {}^\bullet p} \chi(t) \cdot \mathbf{Post}(p,t) \geq \sum_{t \in p^\bullet} \chi(t) \cdot \mathbf{Pre}(p,t),\ \forall p \in P \\
& \forall t \in T, \forall p \in {}^\bullet t:\ \chi(t) \leq \frac{\overline{\mathbf{m}}(p)}{\delta(t) \cdot \mathbf{Pre}(p,t)} \\
& \forall t \in T, {}^\bullet t = \{p\} : \chi(t) \cdot \delta(t) \geq \frac{\overline{\mathbf{m}}(p) - \mathbf{Pre}(p,t) + 1}{\mathbf{Pre}(p,t)} \\
& \forall t \in T, {}^\bullet t = \{p_1, p_2\} : \chi(t) \cdot \sigma(t) \cdot \mathbf{Pre}(p_1,t) \geq \\
& \qquad \overline{\mathbf{m}}(p_1) - \mathbf{Pre}(p_1,t) + \\
& \qquad 1 - N(p_1) \cdot \left(1 - \frac{\overline{\mathbf{m}}(p_2) - \mathbf{Pre}(p_2,t) + 1}{N(p_2) - \mathbf{Pre}(p_2,t) + 1}\right) \\
& \forall t_j, t_k \in ECS,\ r_k \cdot \chi(t_j) = r_j \cdot \chi(t_k) \\
& \mathbf{m_0}, \sigma \geq \mathbf{0}, \chi(t) \geq 0,\ \forall t \in T
\end{aligned}
\tag{7}
$$

Similarly, LPP (7) can be also used for computing the upper marking bound of a given place p in $\mathcal{N}$, considering $\mathbf{m}(p)$ as objective function (instead of $\chi(t_i)$).

3.3 UML Profiles

UML [OMG (2011b)], the current standard modelling language for the industry and the software engineering research community, can be tailored for specific purposes through profiling [Selic (2007); Lagarde et al (2007)]. A UML profile is a UML extension to enrich UML model semantics defined in terms of **stereotypes** (they are concepts in the target domain that will be added to the UML model), **tags** (the attributes of the stereotypes), and **constraints** (they are formulae that apply to stereotypes and UML elements to extend their semantics).

UML profiling has been a very active research field in the last years. As a matter of fact, several UML profiles can be found targeting at different domains and at analysis of non-functional properties (e.g., performance, dependability, or security). Examples of these UML profiles are *Modelling and Analysis of Real-Time and Embedded systems* (MARTE) [OMG (2011a)], *Dependability Analysis and Modelling* (DAM) [Bernardi et al (2011)] and *Security Analysis and Modelling* (SecAM) [Rodríguez et al (2010)]. These profiles enables to specify performance, dependability and security properties within UML models, respectively. In this paper, we use the MARTE profile to specify

the activity timing in a UML model and to specify the number of resources in a system. Namely, we use the MARTE analysis frameworks called Generic Quantitative Analysis Model (GQAM) and the Generic Resource Modelling (GRM), which provide stereotypes to these goals (`gaStep`, `hostDemand` tagged value; `resource`, `resMult` tagged value, respectively).

4 The `PeabraiN` Tool: Architecture and Features

`PeabraiN`, which stands for *"Performance Estimation bAsed (on) Bounds (and) Resource optimisAtIon (for Petri) Nets"*, is made of a set of modules compliant with PIPE-tool modules [Bonet et al (2007)]. In this section, we firstly introduce its architecture in detail and then the features provided by the tool.

4.1 Architecture

`PeabraiN` is programmed with Java and integrated on PIPE [Bonet et al (2007)]. Aside from PIPE library dependencies, it depends on other libraries to perform its functionality. These libraries are related to computational operations in matrices (JAMA) [NIST (2012)], probability distribution functions (SSJ) [Université de Montréal (2014)], and LP solver-specific interface for Java (Java ILP) [JavaILP (2013)].

Fig. 1 depicts the UML Class Diagram (UML-CD) of `PeabraiN`. It has been designed following the Model-View-Controller (MVC) architectural pattern, and its closed layered architecture is composed by three layers. Each layer corresponds with a component in MVC. The main classes added in the last release are depicted in orange colour. These classes are explained in more detail in Section 4.3.

The *data layer* contains classes representing the information needed for the provided functionalities to execute, in terms of LP problems (i.e., the constraints related to the LP problem, the definition of optimisation function, variable types, etc.). The *Facade* structural pattern has been followed in this layer to handle and minimise the classes complexity. The *intermediate layer* contains the classes that implement the algorithms and features provided by `PeabraiN`. Finally, the *GUI layer* encloses the classes related to graphic interfaces for collecting/showing from/to the user input data and results of the functionalities.

The integration of `PeabraiN` in PIPE is depicted in Fig. 2. PIPE allows to be extended through modules that must implement the *IModule* interface. Note that the integration follows an open architecture, as upper layered classes communicate with non-immediate lower layered classes. Namely, Fig. 2 shows how the PIPE-data layer and `PeabraiN`-data layer are related. The *PetriNet-Model* class, which is a matrix representation of the current PN model in PNML (PIPE format), is created by each one of the `PeabraiN` modules. Let

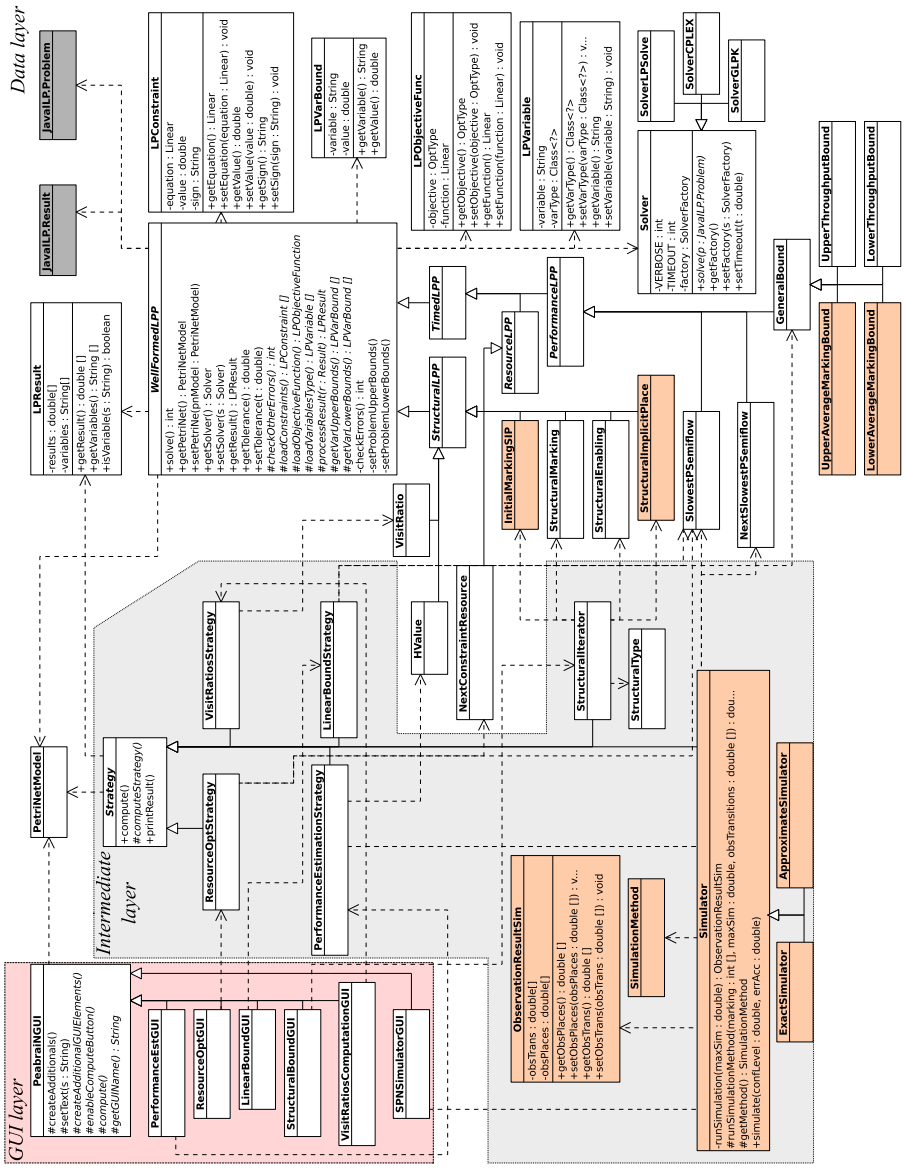


Fig. 1 UML Class diagram of `PeabraiN`.

us remark that our modules do not use PNML as data layer because the module algorithms work with the matrix representation as we are mainly solving LPPs.

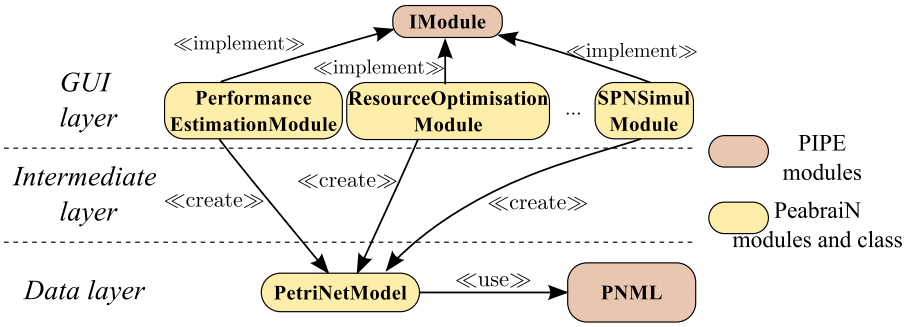


Fig. 2 Integration of `PeabraiN` in the PIPE tool.

4.2 Plug-in Architecture

`PeabraiN` was designed following also a plug-in architecture to facilitate its extension. This architecture makes easier to add more functionalities that make use of LP techniques on PNs. In fact, this is the major benefit of this architecture: It allows a rapid prototype implementation of any algorithm that deals with Petri nets and LP problems. Let us explain how `PeabraiN` can be easily extended by an example.

Recall that to check whether a place is (structurally) implicit can be solved by an LPP (see LPP (1)). Suppose that a user wants to add this feature to `PeabraiN`. The first thing to do is to create a class (termed as *StructuralImplicitPlace*) that extends the abstract class *WellFormedLPP*, and fill in the methods properly. Recall that the Petri net is represented in matrix form by the *PetriNetModel* class, and the methods of *StructuralImplicitPlace* represent the LPP (1) variables, constraints, objective function, etc. After that, an intermediate layer class must be created, *SIPStrategy*, extending the *Strategy* abstract class. This class must properly create and communicate with the new class created (*StructuralImplicitPlace*). After creating it, it needs to fill in the arguments needed by *StructuralImplicitPlace* (if any) and then invoke to *solve()* method. Finally, a class should be created in the GUI layer by extending the *PeabrainGUI* abstract class (let us suppose this class is named as *StructuralImplicitPlaceGUI*). As last step, the integration with PIPE must be performed. Thus, a new module named *StructuralImplicitPlaceModule* implementing *IModule* and using *PetriNetModel* is created.

Fig. 3 illustrates interactions between the user, PIPE and `PeabraiN` when executing this new feature considering the above mentioned classes. Once the *StructuralImplicitPlaceModule* is instanced, it creates a *PetriNetModel* object, taking as parameter the current PNML displayed by PIPE. After that, it creates a *SIPStrategy* and a *StructuralImplicitPlaceGUI*, which take as input parameter the recently created *PetriNetModel* object. The GUI also collects the input data provided by the user (i.e. the index of place to check, *idxPlace*), and then whether such a place is structurally implicit is checked. The last release of `PeabraiN` includes indeed this feature, but the intermediate and GUI

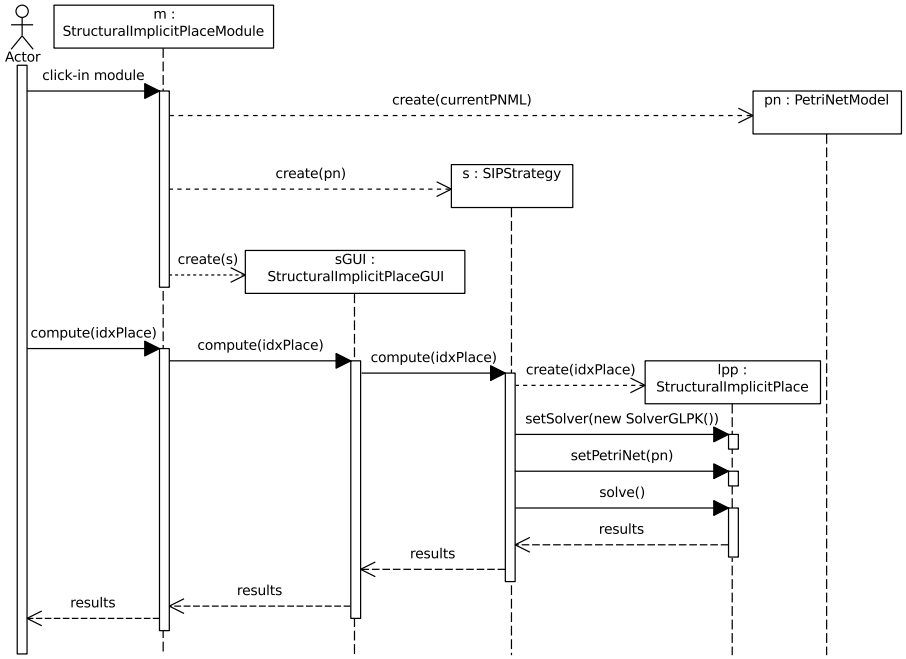


Fig. 3 UML Sequence Diagram for executing *Structural Implicit Place* module.

layered classes are integrated into *StructuralIterator* and *StructuralBoundGUI*, respectively, as it shares some characteristics with other structural properties (mainly, the input parameters can be collected using the same GUI).

4.3 Tool Modules

In the following, the `PeabraiN` modules and what they provide are introduced.

Performance Estimation. It uses an iterative algorithm based on LP problems [Rodríguez and Júlvez (2010); Rodríguez et al (2013)] for computing upper throughput bounds. This iterative approach is applicable for some subclasses of Petri nets (namely, Marked Graphs [Murata (1989)] and Process Petri nets [Tricas (2003)]).

Resource Optimisation. This module enacts an optimal distribution of resources in a shared-resource PN for a given budget and resource costs, trying to enhance the system performance as much as possible was implemented. The theory behind this method was formerly introduced in [Rodríguez et al (2013)]. This method is currently applicable for Process Petri nets [Tricas (2003)].

Visit Ratio Computation. Section 3.1 describes how the vector of visit ratios v of a PN, normalised for a transition $t \in T$ can be computed. This module computes whether the vector of visit ratios has a single solution, otherwise reports that there exists more than one solution, and it cannot be computed.

Linear Bound. This module encompasses several computations related to linear bounds, such as the computation of slowest p-semiflow (see LPP (6)) or the computation of lower(upper) throughput bounds (see LPP (7)) [Campos and Silva (1993)]. Recall that the PN structure needs to fulfil a set of conditions so that the computation of performance bounds has some sense. These conditions are: (i) the PN must be structurally live, (ii) structurally bounded, (iii) have a home state and (iv) its vector of visit ratios must have a unique solution. Our tool is limited to automatically check the latter property, as verifying some of these properties in general nets are NP-decidability problems [Esparza and Nielsen (1994)]. The latter conditions are also applicable to the computation of the slowest p-semiflow of a PN since it is indeed an upper performance bound for the real system performance.

This module also enables to compute the lower and upper average marking bound for a place in a Petri net, using LPP (7) but changing the optimisation function as required.

Structural Marking Bound *and* **Structural Enabling Bound**. The structural marking of a place $p \in P$, and the structural enabling of a transition $t \in T$, can as well be computed by LP problems [Campos and Silva (1993)]. These LP problems are valid for any kind of PN.

Structural Implicit Places. This module allows to compute the structural implicit places of a net and the initial marking such that these places become implicit and thus they can be removed from the net. Both features can be solved by using LPP, as previously explained (see Section 3.1).

This module has been used as example of how to extend the plug-in architecture of `PeabraiN`. The classes implementing these features have been highlighted (orange) in Fig. 1. Note that these new classes belong to data layer. As we have previously said, more logic has been added to some intermediate and GUI layer classes to make use of these features.

GSPN Simulation Analysis. This module incorporates two different stochastic simulation methods for average throughput and average marking computation in SPNs: An exact (discrete) method using the *first reaction method* [Gillespie (1976)], and an approximate (continuous) method using the *Tau-Leaping method* [Gillespie (2007, 2008)].

The exact method is based on the Gillespie's stochastic simulation algorithm (SSA) [Gillespie (1976)]. It performs a set of replications of the simulation, and estimates the average throughput with a given confidence interval level and error accuracy. This algorithm uses a Monte Carlo procedure for numerically generating enabled transitions to be fired [Gillespie (1992)].

Algorithm 1 shows the pseudo-code of the discrete SSA implemented. As input, it needs two parameters: The Petri net $\mathcal{S}$ to be simulated and the maximum system time t_{max} to be simulated. As output, it produces a vector χ of average throughput of transitions. As side product, the vector of average marking of places $\overline{\mathbf{m}}$ is also obtained.

Input: $\mathcal{S}, t_{max}$
Output: χ

1 Set simulated time $t = 0$
2 Set a vector $\mathbf{f}$ of firing count of transitions, i.e., $\mathbf{f} = \mathbf{0}, |\mathbf{f}| = |T|$
3 Set a vector χ of average throughput of transitions, i.e., $\chi = \mathbf{0}, |\chi| = |T|$
4 Set a vector $\overline{\mathbf{m}} \in \mathbb{R}^{|P|}, \overline{\mathbf{m}} = \mathbf{0}$, of average marking of places
5 Fire all enabled immediate transitions until no other immediate transitions are enabled
6 Compute the set $\mathcal{A}$ of timed transitions that are enabled
7 **if** $\mathcal{A} = \emptyset$ **then**
8 | Raise error message, as no system evolution is possible
9 **else**
10 | Get next timed transition $a \in \mathcal{A}$ to be fired in a randomly time t'
11 | Update the average marking of places in ${}^\bullet a$
12 | **repeat**
13 | | $t = t + t'$
14 | | Fire transition $a \in \mathcal{A}$, and increment $\mathbf{f}(a)$ in one unit
15 | | Fire all enabled immediate transitions until no other immediate transitions are enabled
16 | | Compute the set $\mathcal{A}$ of timed transitions that are enabled
17 | | Get next timed transition $a \in \mathcal{A}$ to be fired in a randomly time t'
18 | | Update the average marking of places in ${}^\bullet a$
19 | **until** $t \geq t_{max}$ *or* $\mathcal{A} = \emptyset$
20 | **foreach** $e \in \mathbf{f}$ **do**
21 | | $\chi(e) = \mathbf{f}(e)/t$
22 | **end**
23 **end**

Algorithm 1: Discrete Stochastic Simulation Algorithm implemented in `PeabraiN`.

Steps 1 – 4 initialise the parameters used in the algorithm, namely the simulated time, the vector of firing count of transitions, the vector of average throughput of transitions, and the vector of average marking of places. Then, all enabled immediate transitions are fired until no other immediate transition is enabled (step 5). Step 6 computes the set $\mathcal{A}$ of timed transitions that are enabled. Then, if no timed transition is enabled, an error message is reported indicating that the system cannot evolve, and suggests that initial marking might be revised. Otherwise, a transition $a \in \mathcal{A}$ is picked up for firing at time t' (step 10). Next step updates the average marking of places contained in ${}^\bullet a$. This transition is selected by applying the standard inversion generating method of Monte Carlo theory [Gillespie (1992)]. Steps 12 – 19 are the simulation loop. The simulated time is incremented in t' units (step 13), and transition a is fired and its firing count is incremented (step 14). Then, previous steps 5, 6, 10 and 11 are repeated. The iteration loop ends either when

the maximum of simulated time is reached, or when there is no any timed transition to be fired. Finally, steps 20 – 22 compute the average throughput of each transition by dividing the number of firing between the total simulated time.

`PeabraiN` provides other simulation method that implements Tau-Leaping method [Gillespie (2007, 2008)], an approximate way of accelerating the SSA in which each time step advances the system through possibly many events at the same time. In terms of Petri nets, each step the enabled timed transitions are fired as much as possible, as indicated by a Poisson distribution.

Algorithm 2 introduces this simulation method. As input, it needs the Petri net to be simulated, and the maximum of simulated time. As output, it produces a vector χ of average throughput of transitions. As side product, the vector of average marking of places $\overline{\mathbf{m}}$ is also obtained. Note that both algorithms use the same input, and produce the same output.

Input: $\mathcal{S}, t_{max}$
Output: χ

1 Set simulated time $t = 0$
2 Set a vector $\mathbf{f}$ of firing count of transitions, i.e., $\mathbf{f} = \mathbf{0}, |\mathbf{f}| = |T|$
3 Set a vector χ of average throughput of transitions, i.e., $\chi = \mathbf{0}, |\chi| = |T|$
4 Set a vector $\overline{\mathbf{m}} \in \mathbb{R}^{|P|}, \overline{\mathbf{m}} = \mathbf{0}$, of average marking of places
5 Fire all enabled immediate transitions until no other immediate transitions are enabled
6 Compute the set $\mathcal{A}$ of timed transitions that are enabled
7 **if** $\mathcal{A} = \emptyset$ **then**
8 | Raise error message, as no system evolution is possible
9 **else**
10 | **repeat**
11 | | Compute τ such that leap condition is fulfilled
12 | | Fire each transition in $\mathcal{A}$ a randomly generated number of times, and increment $\mathbf{f}$ accordingly
13 | | Update the average marking of places in $\mathcal{A}$ accordingly
14 | | $t = t + \tau$
15 | | Fire all enabled immediate transitions until no other immediate transitions are enabled
16 | | Compute the set $\mathcal{A}$ of timed transitions that are enabled
17 | **until** $t \geq t_{max}$ *or* $\mathcal{A} = \emptyset$
18 | **foreach** $e \in \mathbf{f}$ **do**
19 | | $\chi(e) = \mathbf{f}(e)/t$
20 | **end**
21 **end**

Algorithm 2: Tau-Leaping Stochastic Simulation Algorithm implemented in `PeabraiN`.

Steps 1 – 9 are equivalent to the same steps at Algorithm 1. Both algorithms mainly differ in the iteration loop. Here, the iteration loop starts in step 11, where a time τ that fulfils the leap condition is computed. The leap condition ensures that such a τ is small enough that no propensity function changes by a significant amount [Gillespie (2007)]. Given a marking $\mathbf{m}$, such a value τ is

computed as:

$$\tau = \min_{\forall a \in \mathcal{A}} \left(\frac{\max(\epsilon \cdot \sum_{\forall a \in A} \delta(a), 1)}{|\sum_{p \in \mathcal{B}_a} \mathbf{C}(p,a) \cdot \delta(a) \cdot \mathbf{e}(a)|}, \frac{\max(\epsilon \cdot \sum_{\forall a \in A} \delta(a), 1)^2}{|\sum_{p \in \mathcal{B}_a} \mathbf{C}(p,a)^2 \cdot \delta(a) \cdot \mathbf{e_m}(a)|} \right) \quad (8)$$

where $\mathcal{B}_a = \{{}^{\bullet\bullet}a\} \bigcup \{a^{\bullet\bullet}\}$, $\delta(a)$ is the mean of the exponential firing time distribution associated to a, and $\mathbf{e_m}(a)$ is the enabling degree of transition a.

After computing τ, each enabled timed transition a fires in $[t, t+\tau)$ a number of times n which is a Poisson random variable with mean (and variance) $\delta(a) \cdot \mathbf{e}(a) \cdot \tau$, i.e., $n = \mathcal{P}(\delta(a) \cdot \mathbf{e}(a) \cdot \tau)$, and the vector of firing counts is accordingly incremented (step 12). Similarly, step 13 increments accordingly the average marking of places in ${}^{\bullet}a, \forall a \in \mathcal{A}$. Then, step 14 increments the simulated time in τ units. As before, previous steps 5 and 6 are repeated. The iteration loop ends either when the simulated time reaches the maximum simulated time, or the set of enabled transitions is empty. As in Algorithm 1, the final steps (18 – 20) compute the average throughput of each transition.

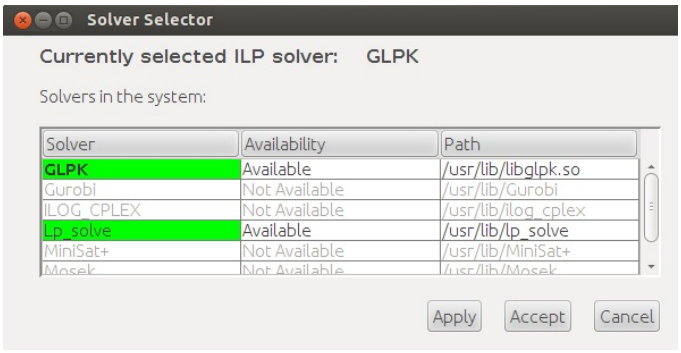
Algorithm 2 outperforms Algorithm 1 for net systems where the population (i.e., the number of initial tokens) is large, as in each iteration step the enabled timed transitions are fired more than just one time. Of course, the simulation results produced by Algorithm 2 may be less exact than the ones computed by Algorithm 1. Note that enabled timed transitions are fired at a time, regardless these firings may enable subsequent immediate transitions. Thus, timed transitions could be fired multiple times at once which would change the net behaviour. This is an interesting issue that needs further study. We aim at studying the structure of nets whose behaviour may be influenced by this simulation method.

In the next section we compare the performance and accuracy of both algorithms in a case study. As future work, we aim at evaluating them with a large set of Petri nets.

4.4 Extended Features

In this section, we summarise other features that `PeabraiN` provides and are used by all modules. Namely, `PeabraiN` allows to select the LP solver to be used and to execute all module using the command-line interface. The first feature could be used, for instance, to evaluate the performance of LP solvers in the computation of some Petri net properties; while the latter feature enables to batch executions without user interaction, and to concatenate the execution of different modules.

Linear Programming Solver Selector. A `PeabraiN` *Settings* option has been added into PIPE to allow a user to select the LP solver to be used in LP computations. To this aim, we have followed a *Model-View-Controller* architectural



(a)

```
Default = GLPK
Solver7 = MiniSat+,/usr/lib/MiniSat+
Solver6 = SAT4J,/usr/lib/SAT4J
Solver5 = GLPK,/usr/lib/libglpk.so
Solver4 = Mosek,/usr/lib/Mosek
Solver3 = Gurobi,/usr/lib/Gurobi
Solver2 = ILOG_CPLEX,/usr/lib/ilog_cplex
Solver1 = Lp_solve,/usr/lib/lp_solve
```

(b)

Fig. 4 (a) LP solver selector GUI and (b) configuration file to dynamically generate solver list.

pattern for the design of the solver selector window, which allows to the user to select the LP solver to be used by `PeabraiN` modules. This window has been integrated into the PIPE menu, thus allowing an easily configuration from the main PIPE window. Besides, the LP solver list is dynamically generated and its content relies on a configuration file[1] that can be easily modified (following a given syntax) by any user. This new GUI and a example of configuration file are given in Fig. 4(a) and (b), respectively. We have also used the *Singleton* creational pattern to ensure a unique instance of the LP solver across all modules. This enhances the performance of `PeabraiN`, as the same solver can be concurrently used by different modules, thus saving the LP loading penalty time.

`Command-Line Interface.` `PeabraiN` also provides a way to execute any module via a command-line interface (CLI). This feature enables to batch computations in Petri nets without the need of user interaction, and even to easily integrate `PeabraiN` modules and results with other tools.

Fig. 5 depicts the UML-CD of `PeabraiN` Command-Line Interface modules. Note that the integration with `PeabraiN` modules are done through module strategies (left-side classes of the figure). We have followed the same design rules as with `PeabraiN`. Thus, the *Facade* structural pattern and a plug-in architecture have been used: When a new module needs CLI support, a developer only needs to extend *PeabraiNCLI* class and fill in the required methods

[1] Named ".solverselector.conf", and located at the home folder of the user.

properly. Moreover, this new functionality was designed under the idea of minimising as much as possible the modification of current code when adding CLI support for other modules. Therefore, the *PipeCLI*, which is the one currently executed by a user when executing `PeabraiN` through a console, automatically loads the modules that can be executed by checking a predefined CLI module path.

The support of these dynamic CLI modules rises other dynamic features, such as the dynamic generation of help messages, which report about the usage of each module with its corresponding module parameters. To this aim, the parameters are collected by the *MyParamValue* class, which defines a set of methods and properties for input parameters. Recall that a module has some mandatory (and common) parameters, such as the PNML file and the LP solver to be used by the module, and other specific parameters, such as the identification label of a place or a transition in the case of structural marking bound and structural enabling bound modules, respectively.

4.5 Tool Remarks

In this last part, we summarise the kind of Petri nets where the methods provided by `PeabraiN` can be applied. Exact simulation methods are valid to any kind of Petri net subclass (e.g., a simple net). However, approximate simulation methods are only applicable to nets where only immediate transitions are in conflict. Besides, this simulation method is recommended for nets with a high number of tokens since accuracy error is higher when population is low. Similarly, lower/upper throughput (marking) bound computation can be done in any kind of Petri net using the LPP (7). Iterative upper throughput bound algorithms, and resource optimisation, are valid for Petri nets where the vector of visit ratios is computable. Likewise, computing slowest p-semiflow using LPP 6 is only possible when the vector of visit ratios is known. Recall that the vector of visit ratios is easily computable when it depends only on the net structure and on the routing rates. Otherwise, the reachability set of a net is needed to compute it.

Tool Availability. `PeabraiN` is released under GNU GPL version 3 license, and further information about tool requirements and installation steps, tool binaries and sources can be found in the project web page `http://webdiis.unizar.es/GISED/?q=tool/peabrain`.

5 Case Study: A Building CCTV System

In this section, we study the performance of a Building Closed Circuit TV System (CCTV), inspired by the system introduced in [Petriu and Woodside

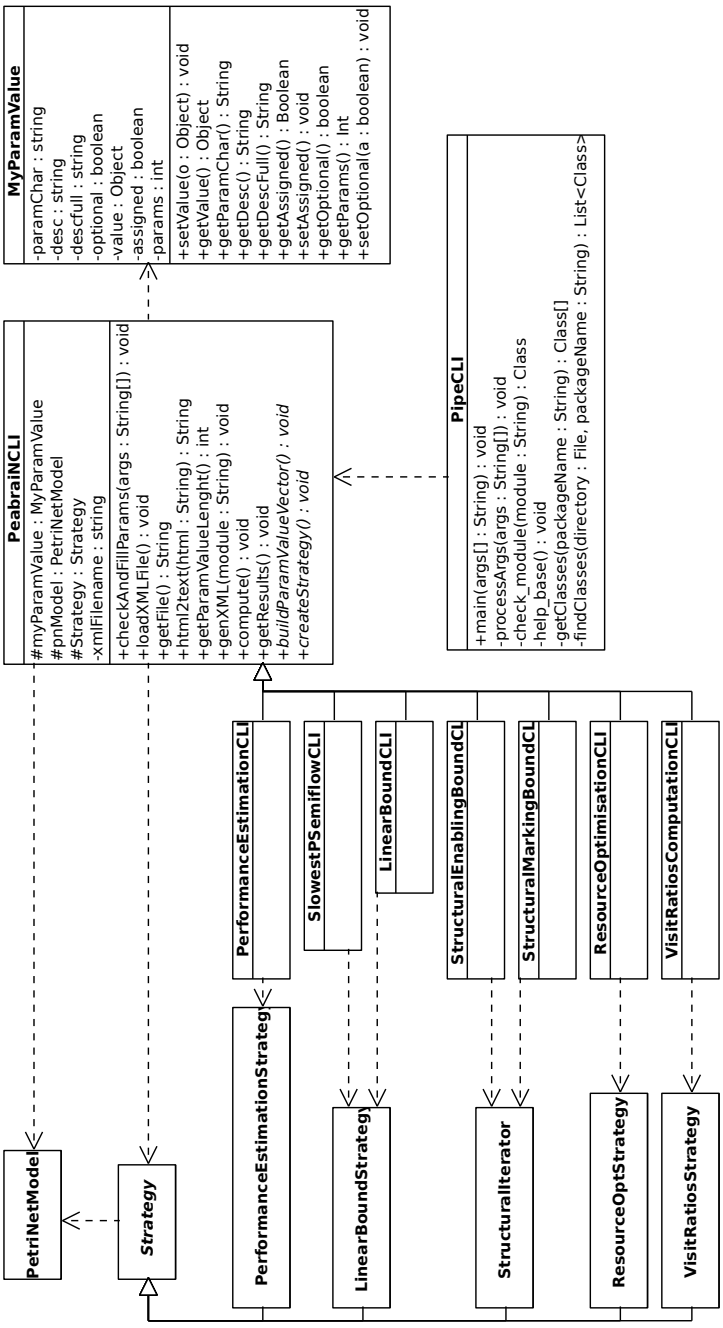


Fig. 5 UML Class diagram of `PeabraiN` Command-Line Interface.

(2003); Woodside et al (2005)]. The CCTV collects video images from a set of cameras, stores them for forensics analysis if needed, and analyses them in real-time to report suspicious behaviours. The system is first modelled with UML and annotated with MARTE [OMG (2011a)] profile, and then converted to a Petri net (PN). Lastly, its performance is estimated using upper throughput bound computation under different scenarios. We also simulate the system with `PeabraiN` and other existing PN tools to validate the results.

5.1 System Description

Fig. 6 depicts the UML deployment diagram (UML-DD) of CCTV, including the hardware resources (depicted as cubes) and their network links (arrows between cubes, or cubes in the case of intranets). The architecture of the CCTV is composed by a database server and an application server, both connected through an intranet. The database server deploys a component, *DatabaseManager*, in charge of interacting with the database, represented as an storage resource (*HardDisk*). The application server embraces three different elements: The access controller, the video acquisition and the alarm controller. The video acquisition controller is composed by four different resources: *VideoController*, *AcquireProc*, *StoreProc* y *BufferManager*. All resources are annotated with MARTE [OMG (2011a)] to express its multiplicity (`resource` stereotype, and `resMult` tagged value). We assume that the system is able to run in parallel with several instances of each resource (but the *VideoController*), which outperforms a non-parallel execution.

In this paper, we focus on the UML Sequence Diagram (UML-SD) of acquire/store video scenario depicted in Fig. 7. The *VideoController* iterates acquiring and storing video frames for each camera deployed in the system. The number of cameras to iterate has been set to a variable $\$nCameras$ for sensitive analysis. Each image frame is processed by an instance of *AcquireProc* resource. This resource allocates a memory buffer to process the image. The memory buffer handling is performed by an instance of *BufferManager* resource. Once the memory buffer is allocated, the image frame is transferred over the network (represented by an external operation with a throughput depending on the number of packets $\$nP$ transmitted) and sent to *StoreProc* resource to be stored into the database. An instance of *StoreProc* resource gets the image and writes it into the database. After that, it analyses the image and raises an alarm when some suspicious behaviour is detected. The image analysis can be carried out using a behavioural profiling based on multiple observations method to achieve a 91% of success [Bouma et al (2013)]. Suspicious behaviour rate has set to $\$ssRate$ input parameter, and will be used for sensitive analysis. Once the image has been analysed, the previous acquired memory buffer is released, and image processing for the camera is finished.

The acquire (release) of a resource has been indicated through the `gaAcqStep` (`gaRelStep`) stereotype (see MARTE annotations in Fig. 7), also denoting the number of resources acquired (released). To avoid cluttering we only show the

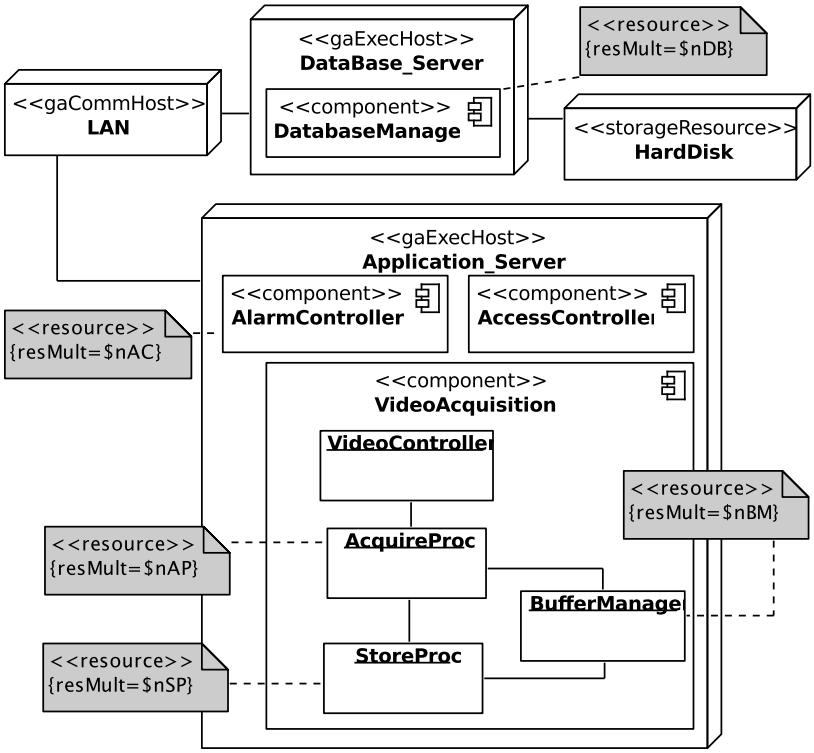


Fig. 6 CCTV UML Deployment diagram (annotated with MARTE).

first acquire (release) of resource *AcquireProc*. The rest of resources are annotated in the same way. Activities are annotated with `gaStep` stereotype to specify how long takes, on average, each activity using the tagged-value `hostDemand`.

Description as a Petri net. Figure 8 shows the PN obtained from the transformation of the UML-SD depicted in Fig. 7. The transformation from UML to PN is documented in [Distefano et al (2011)], and can be carried out by several tools, such as ArgoPN [Delatour and de Lamotte (2003)], ArgoPerformance [Distefano et al (2011)] or ArgoSPE [Gómez-Martínez and Merseguer (2006)] (see Section 2). In this paper, ArgoSPE tool[2] has been chosen since the output format is compatible with GreatSPN tool [Baarir et al (2009)], a PN tool later used for simulate the system. A parser from GreatSPN input net format to `PeabraiN` has also been implemented. Note that ArgoSPE is useful in this context since software engineers usually work with UML diagrams and this tool transforms them to PNs. Nevertheless, any of the other tools could be used in conjunction with `PeabraiN` to obtain a PN model from a UML model. Note that any system transformed to Petri net could be analysable

[2] Available at `https://argospe.tigris.org`.

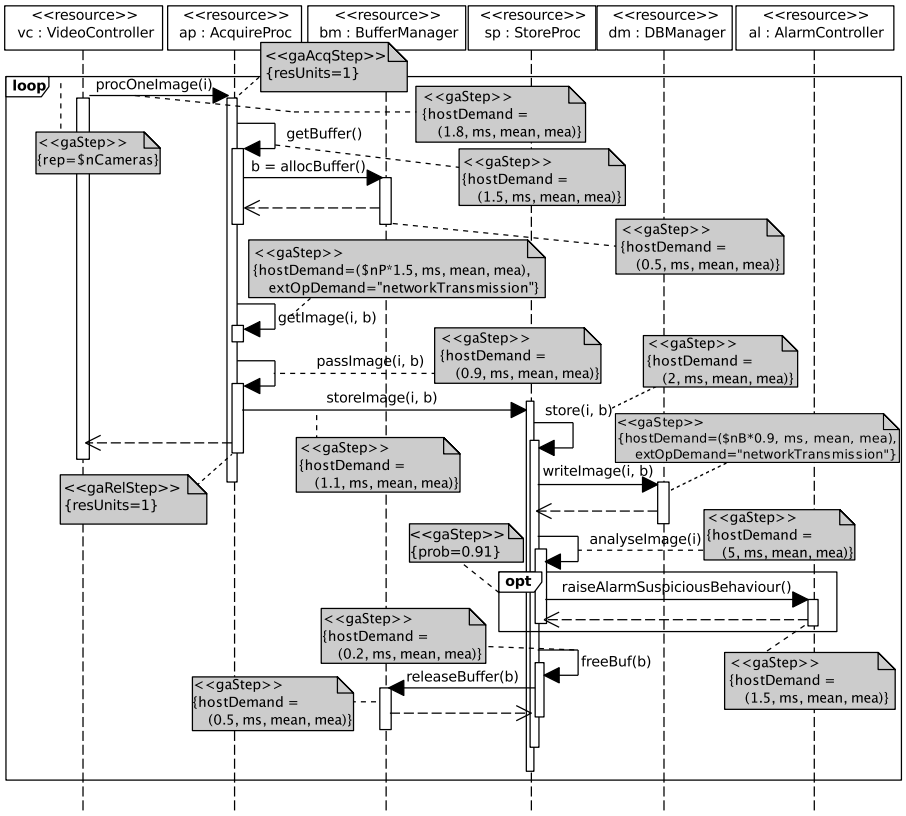


Fig. 7 CCTV Acquire/Store Video scenario.

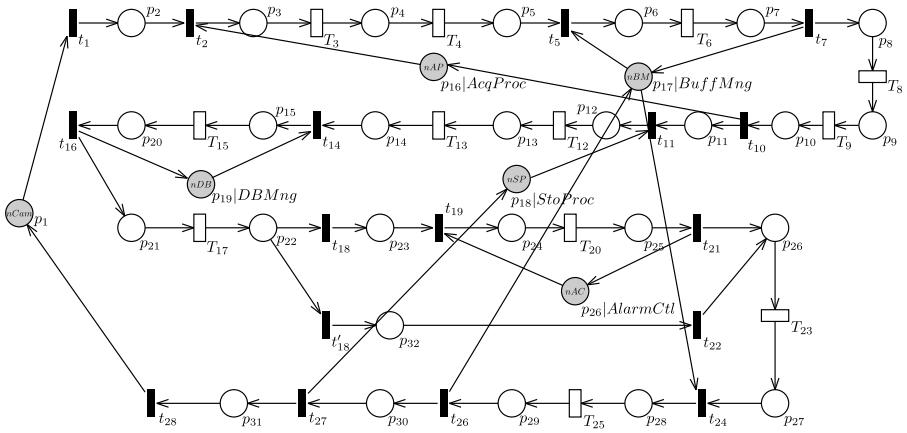


Fig. 8 Petri net of the CCTV. Resource places are depicted in light grey.

Transition	Activity	Value(s)
T_3	*procOneImage*	1.8ms
T_4	*getBuffer*	1.5ms
T_6	*allocBuffer*	0.5ms
T_8	*getImage*	12ms (assuming $\$nP = 8$)
T_9	*passImage*	0.9ms
T_{12}	*storeImage*	1.1ms
T_{13}	*store*	2ms
T_{15}	*writeImage*	7.2ms (assuming $\$nB = 8$)
T_{17}	*analyseImage*	5ms
t_{18}	*$suspiciousProb*	0.91
t'_{18}	*(1 - $suspiciousRate)*	0.09
T_{20}	*raiseAlarm*	2ms
T_{23}	*freeBuff*	0.2ms
T_{25}	*releaseBuff*	0.5ms

(a) Delay of net transitions

Place	Parameter	Value(s)
p_1	No. of cameras	$\{50, 100, 150\}$
p_{16}	Acquire process instances	$\{10, 20, 30\}$
p_{17}	Buffer manager instances	$\{10, 20, 30\}$
p_{18}	Store process instances	$\{10, 20, 30\}$
p_{19}	Database manager instances	5
p_{33}	Alarm controller instances	2

(b) Initial marking of net places

Table 1 CCTV experiment settings.

with `PeabraiN`. Of course, depending on the net structure some features may not apply (see Section 4.5).

Each resource annotated in Fig. 6 is represented by a (light-grey highlighted) place in the PN: p_1 (number of cameras), p_{16} (acquire process), p_{17} (buffer manager), p_{18} (store process) and p_{19} (database manager), and p_{33} (alarm controller). Table 1 summarises the number of instances of each resource, since they will be represented by tokens in the respective places.

The acquire (release) of a resource is transformed into an immediate transition with an input (output) arc. For example, transition t_23 represents the acquire of the *Acquire Process*, while t_{10} represents the release of such a resource. The activities, self-messages in Fig. 7, are transformed into an exponential transition in the PN with its corresponding duration (given in Table 1).

In the sequel, we analyse the properties of the PN obtained after transformation from the UML-SD of Fig. 7. The net is composed of 33 places and 29 transitions. In fact, the PN is a *process Petri net* [Tricas (2003)], a subclass of PN where the analysis methods introduced in [Rodríguez et al (2013)] to compute improved upper throughput bounds and resource optimisation apply. The net has six minimal p-semiflows: $\mathbf{y}_1 = \{p_{1-15}, p_{20-32}\}$, $\mathbf{y}_2 = \{p_{3-10}, p_{16}\}$, $\mathbf{y}_3 = \{p_6, p_7, p_{17}, p_{28}, p_{29}\}$, $\mathbf{y}_4 = \{p_{12-15}, p_{18}, p_{20-30}, p_{32}\}$, $\mathbf{y}_5 = \{p_{15}, p_{19}, p_{20}\}$, $\mathbf{y}_6 = \{p_{24-26}\}$. For the sake of readability, we have shorten and grouped the place names. All places are covered by some p-semiflow, therefore the net is bounded. Similarly, the net has two minimal t-semiflows: $\mathbf{x}_1 = \{t_{1-21}, t_{23-28}\}$ and $\mathbf{x}_2 = \{t_{1-28}, t'_{18}\}$. All transitions are covered by some t-semiflow, there it

might be bounded and live. Transitions t_{18}, t'_{18}, are in an extended free-choice conflict, while transitions t_5, t_{24}, are in structural (but not free-choice) conflict. When both transitions are in an effective conflict for a given marking $\overline{\mathbf{m}}$, the transition to be fired is randomly chosen.

We compute the structural implicit places using the new feature of `PeabraiN`. The net has four implicit places, which match with the places representing resources (but the number of cameras) within the system: p_{16} (acquire process), p_{17} (buffer manager), p_{18} (store process) and p_{19} (database manager), and p_{33} (alarm controller). In fact, the initial marking for each one of these places such that they become structurally implicit is equal to the initial marking of p_1 (number of cameras in the system). That is, if the initial marking of the former places are greater than or equal to the initial parking of p_1, then they can be removed without effect to the behaviour of the Petri net.

5.2 Experiments and Discussion

In the sequel, we validate `PeabraiN`'s new features by evaluating the CCTV under different workloads. We set the number of resources as expressed in Table 1: The number of cameras will vary between 50, 100 and 150, while the number of resource instances in the application server varies between 10, 20 and 30 (but the alarm controller, which remains equal to 2). Similarly, the database manager instances is set to 5.

The experiments have been run in a machine with Intel Pentium 4 CPU 3.60GHz 2048KiB cache, 3GiB RAM 533MHz, running a 32-bit Linux distribution. The net depicted in Fig. 8 has been simulated with the two simulation methods provided by `PeabraiN` (exact, approximate). The set of experiments has been executed with two different confidence level and accuracy configurations: $95\% - 5\%$, and $99\% - 1\%$.

Tables 2 and 3 summarise the throughput and execution time results of the set of experiments performed, respectively. Throughput results are taken from the throughput of transition t_1. The first column indicates the number of cameras, while the second the number of other resource instances in the application server. Then, the results given by `PeabraiN` using both simulation methods, as well as its execution time (in seconds), is shown. Lastly, the percentage of approximate simulation result with respect to exact simulation result is shown.

The results of Table 2 show that approximate simulation method performs better when the number of tokens is high. In fact, the results are almost the same with the most populated net. It also remarkable that the simulation parameters (confidence level and accuracy) do not a have a big impact in the results. In all cases, the difference remains at $\pm 0.5\%$.

Similarly, the execution of approximate simulation method clearly outperforms the execution of exact simulation method. Table 3 shows that approximate simulation method executes near to half a time the execution time of exact simulation method, for all cases under study.

No. of cameras	Other resource instances	PeabraiN (exact)	PeabraiN (approximate)	%
Confidence interval 95%, *accuracy* 5%				
50	10	0.5788341	0.6130577	5.91%
	20	0.6933366	0.6956743	0.34%
	30	0.6942851	0.69478	0.07%
100	10	0.5818307	0.6154501	5.78%
	20	0.6981312	0.69637	−0.25%
	30	0.6972403	0.6957046	−0.22%
150	10	0.5839596	0.6169496	5.65%
	20	0.6973815	0.6995541	0.31%
	30	0.6958464	0.6958727	0.00%
Confidence interval 99%, *accuracy* 1%				
50	10	0.5799806	0.612705	5.64%
	20	0.6967369	0.6958847	−0.12%
	30	0.6952727	0.6961634	0.13%
100	10	0.5812165	0.6148568	5.79%
	20	0.6963155	0.6929288	−0.49%
	30	0.6963199	0.6939515	−0.34%
150	10	0.5846464	0.6176673	5.65%
	20	0.6971197	0.6964626	−0.09%
	30	0.6963669	0.6956921	−0.10%

Table 2 CCTV experiment: Throughput results.

No. of cameras	Other resource instances	PeabraiN (exact)	PeabraiN (approximate)	%
Confidence interval 95%, *accuracy* 5%				
50	10	206.529	98.692	−52.21%
	20	216.059	109.268	−49.43%
	30	220.462	114.761	−47.95%
100	10	182.443	99.581	−45.42%
	20	219.214	108.902	−50.32%
	30	220.079	109.632	−50.19%
150	10	181.053	101.071	−44.18%
	20	218.096	110.519	−49.33%
	30	220.4	110.857	−49.70%
Confidence interval 99%, *accuracy* 1%				
50	10	179.754	98.978	−44.94%
	20	220.374	114.931	−47.85%
	30	218.702	113.891	−47.92%
100	10	181.305	101.368	−44.09%
	20	215.844	108.217	−49.86%
	30	217.812	109.141	−49.89%
150	10	182.096	100.06	−45.05%
	20	219.847	111.187	−49.43%
	30	223.386	110.701	−50.44%

Table 3 CCTV experiment: Execution time results (in seconds).

In brief, the approximate simulation method is recommended for highly populated nets (i.e., Petri nets with a huge number of tokens). About the simulation parameters, a high confidence level and low accuracy error are recommended since the difference on execution time when changing them is negligible.

Let us compute the upper throughput bound of the most populated net in the experiments (that is, 150 cameras and 30 other resources). The slowest p-semiflow is $\mathbf{y}_5 = \{p_{15}, p_{19}, p_{20}\}$, with a throughput of 0.694445. The computation of this bound takes 0.2830 seconds. Note that this upper throughput bound is almost identical to the simulation result, but the execution time is less than half a second, instead of 220.4 seconds (or 223.38 seconds, with confidence level 99%, accuracy error 1%).

The slowest p-semiflow is also given feedback about what is the most constraining resource/part in the system. Thus, whether a system designer would need to optimise this system, s/he has two choices: either to increment the number of instances of database manager (since the place representing such a resource is contained in the support of $\mathbf{y}_5$), or to improve the function *writeImage* represented by transition T_{15} (see Table 1).

Therefore, upper throughput bounds computation provides to software designers rapid information about what functions and resources can be optimised in order to improve system performance.

6 Conclusions and Future Work

Software systems modelling with UML, standard *de facto* in the software development industry, can be enriched with UML profiles such as MARTE to specify performance data (e.g., duration of activities, transmissions, etc.) within UML models. These annotated UML models can be transformed to formal models to enhance the analysis capabilities. In this paper, we consider Petri nets as formal model obtained from UML models enriched with MARTE annotations. Existing tools for performance (or throughput) evaluation in Petri nets rely either on the exploration of the whole state space, or either on simulating the net. However, these methods are unfeasible for large systems either because of the well-known state-space explosion appears, or simulation does not converge in a reasonable time. To overcome these issues, upper throughput bounds can be computed. This paper introduces `PeabraiN`, a Petri net tool that enables, among other features, the computation of upper (and lower) throughput bounds.

`PeabraiN` is a collection of PIPE-modules to simulate and compute several Petri net properties that can be compute using Linear Programming problems. Some of these properties are, for instance, upper (and lower) throughput bounds, slowest p-semiflow or structural marking (and enabling). `PeabraiN` also enables to compute structural implicit places of a net, and lower and upper average marking bound for places. Besides, it allows to select the Linear Programming solver to be used, and incorporates a command-line interface to

easily batch computations and collect results. Apart from a exact simulation method, it implements an approximate method to simulate Stochastic Petri nets using the Tau-Leaping method.

This paper introduces the plug-in architecture of `PeabraiN` and how it can be easily extended, as well as the modules and features provided by the tool. By means of a case study that models a building closed circuit TV system we also show the benefit of using upper throughput bounds to evaluate the performance in large systems where the exact performance computation becomes unfeasible, in terms of execution time. In summary, the usage of upper throughput bounds provides a software designer in a fast way information about what are the slowest methods in the system, in which initial improvement efforts should be targeted to obtain a better system performance.

As future work, we plan to integrate `PeabraiN` with UML CASE tools. This integration brings several benefits into the foreground: First, formal methods can be automatically adopted in the design and analysis of UML models; and second, it easily provides feedback to the software engineers about what the most constraining resources/slowest activities in the system are, and then more effort should be done to optimise them. We also aim at extending the tool with more features related to Petri nets. We are at the moment implementing a web service using AJAX and PHP for remotely executing `PeabraiN`.

Acknowledgements The author would like to thank Iván Pamplona, an undergraduate student who helped in the implementation of `PeabraiN` enhancements. This work was partially supported by the EU Horizon 2020 research and innovation programme under grant agreement no. 644869 (DICE) and by the Spanish MICINN project CyCriSec (TIN2014-58457-R).

References

Baarir S, Beccuti M, Cerotti D, De Pierro M, Donatelli S, Franceschinis G (2009) The GreatSPN tool: recent enhancements. SIGMETRICS Perform Eval Rev 36(4):4–9

Berardinelli L, Bernardi S, Cortellessa V, Merseguer J (2009) UML Profiles for Non-functional Properties at Work: Analyzing Reliability, Availability and Performance. In: Boskovic M, Gasevic D, Pahl C, Schatz B (eds) 2nd International Workshop on Non-functional System Properties in Domain Specific Modeling Languages (NFPinDSML), CEUR, CEUR, Denver, Colorado, USA, vol 553

Bernardi S, Merseguer J (2007) Performance evaluation of UML design with Stochastic Well-formed Nets. Journal of Systems and Software 80(11):1843–1865

Bernardi S, Merseguer J, Petriu D (2011) A Dependability Profile within MARTE. Journal of Software and Systems Modeling 10(3):313–336

Bonet P, Lladó CM (2012) Importing PMIF Models into PIPE2 Using M2M Transformation. In: Proceedings of the 3rd ACM/SPEC International Con-

ference on Performance Engineering (ICPE), ACM, New York, NY, USA, pp 245–246

Bonet P, Lladó C, Puigjaner R, Knottenbelt W (2007) PIPE v2.5: A Petri Net Tool for Performance Modelling. In: Proceedings of the 23rd Latin American Conference on Informatics (CLEI), Costa Rica

Bouma H, Vogels J, Aarts O, Kruszynski C, Wijn R, Burghouts G (2013) Behavioral profiling in CCTV cameras by combining multiple subtle suspicious observations of different surveillance operators. In: SPIE Defense, Security, and Sensing, International Society for Optics and Photonics, pp 87,451A–87,451A

Campos J, Silva M (1992) Structural Techniques and Performance Bounds of Stochastic Petri Net Models. Lect Notes Comput Sc 609:352–391

Campos J, Silva M (1993) Embedded Product-Form Queueing Networks and the Improvement of Performance Bounds for Petri Net Systems. Perform Evaluation 18(1):3–19

Chiola G, Anglano C, Campos J, Colom J, Silva M (1993) Operational Analysis of Timed Petri Nets and Application to the Computation of Performance Bounds. In: Proceedings of the 5th International Workshop on Petri Nets and Performance Models (PNPM), IEEE Computer Society Press, Toulouse, France, pp 128–137

Delatour J, de Lamotte F (2003) ArgoPN: a CASE Tool Merging UML and Petri Nets. In: Proceedings of the 3rd International Workshop on New Developments in Digital Libraries (NDDL) and the 1st International Workshop on Validation and Verification of Software for Enterprise Information Systems (VVEIS), pp 94–102

Distefano S, Scarpa M, Puliafito A (2011) From UML to Petri Nets: The PCM-Based Methodology. IEEE Transactions on Software Engineering 37(1):65–79

Esparza J, Nielsen M (1994) Decidability Issues for Petri Nets - a survey. Bulletin of the EATCS 52:244–262

Fecher H, Schönborn J, Kyas M, Roever WP (2005) 29 New Unclarities in the Semantics of UML 2.0 State Machines. In: Lau KK, Banach R (eds) Formal Methods and Software Engineering, Lecture Notes in Computer Science, vol 3785, Springer Berlin Heidelberg, pp 52–65

Florin G, Natkin S (1989) Necessary and Sufficient Ergodicity Condition for Open Synchronized Queueing Networks. IEEE Trans Softw Eng 15(4):367–380

Garcia-Valles F, Colom J (1999) Implicit Places in Net Systems. In: Proceedings of the 8th International Workshop on Petri Nets and Performance Models, pp 104–113

Gillespie D (1992) Markov Processes: An Introduction for Physical Scientists. Academic Press

Gillespie DT (1976) A General Method for Numerically Simulating the Stochastic Time Evolution of Coupled Chemical Reactions. J Comput Phys 22(4):403–434

Gillespie DT (2007) Stochastic Simulation of Chemical Kinetics. Annu Rev Phys Chem 58:35–55

Gillespie DT (2008) Simulation Methods in Systems Biology. In: Formal Methods for Computational Systems Biology, Lecture Notes in Computer Science, vol 5016, Springer Berlin Heidelberg, pp 125–167

Gómez-Martínez E, Merseguer J (2006) ArgoSPE: Model-Based Software Performance Engineering. In: International Conference of Application and Theory of Petri Nets, pp 401–410

Hillah LM, Kindler E, Kordon F, Petrucci L, Tréves N (2009) A primer on the Petri Net Markup Language and ISO/IEC 15909-2. Petri Net Newsletter 76:9–28

JavaILP (2013) Java Interface to ILP Solvers library. Online, `http://javailp.sourceforge.net/`

Koch M, Parisi-Presicce F (2006) UML Specification of Access Control Policies and their Formal Verification. Software and System Modeling 5(4):429–447

Lagarde F, Espinoza H, Terrier F, Gérard S (2007) Improving UML Profile Design Practices by Leveraging Conceptual Domain Models. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, ACM, New York, NY, USA, ASE'07, pp 445–448

Liu Z (1995) Performance Bounds for Stochastic Timed Petri Nets. In: Proceedings of the 16th ICATPN, Springer-Verlag, pp 316–334

Lladó CM, Harrison PG (2011) A PMIF with Petri Net Building Blocks. In: Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering (ICPE), ACM, New York, NY, USA, pp 103–108

López-Grao JP, Merseguer J, Campos J (2004) From UML Activity Diagrams to Stochastic Petri Nets: Application to Software Performance Engineering. In: Proceedings of the 4th International Workshop on Software and Performance (WOSP), ACM, New York, NY, USA, pp 25–36

Merseguer J, Campos J, Bernardi S, Donatelli S (2002) A Compositional Semantics for UML State Machines Aimed at Performance Evaluation. In: Proceedings of the 6th International Workshop on Discrete Event Systems (WODES), IEEE Computer Society, Washington, DC, USA, WODES '02, pp 295–302

Molloy M (1982) Performance Analysis Using Stochastic Petri Nets. IEEE Trans Comput C-31(9):913–917

Murata T (1989) Petri Nets: Properties, Analysis and Applications. In: Proceedings of the IEEE, vol 77, pp 541–580

NIST (2012) JAMA : A Java Matrix Package. Online, `http://math.nist.gov/javanumerics/jama/`

OMG (2011a) A UML profile for Modeling and Analysis of Real Time Embedded Systems (MARTE). Object Management Group, document formal/11-06-02

OMG (2011b) Unified Modelling Language: Superstructure. Object Management Group, version 2.4, formal/11-08-05

Petriu D, Woodside C (2003) Performance Analysis with UML. In: Lavagno L, Martin G, Selic B (eds) UML for Real, Springer US, pp 221–240

Randimbivololona F (2001) Orientations in Verification Engineering of Avionics Software. In: Informatics, Lecture Notes in Computer Science, vol 2000, Springer Berlin/Heidelberg, pp 131–137

Rodríguez RJ, Júlvez J (2010) Accurate Performance Estimation for Stochastic Marked Graphs by Bottleneck Regrowing. In: Proceedings of the 7th European Performance Engineering Workshop (EPEW), Springer, Lecture Notes in Computer Science, vol 6342, pp 175–190

Rodríguez RJ, Merseguer J, Bernardi S (2010) Modelling and Analysing Resilience as a Security Issue within UML. In: Proceedings of the 2nd International Workshop on Software Engineering for Resilient Systems (SERENE), ACM, London, United Kingdom, pp 42–51

Rodríguez RJ, Júlvez J, Merseguer J (2012) PeabraiN: A PIPE Extension for Performance Estimation and Resource Optimisation. In: Proceedings of the 12th International Conference on Application of Concurrency to System Designs (ACSD), IEEE, pp 142–147

Rodríguez RJ, Júlvez J, Merseguer J (2013) On the Performance Estimation and Resource Optimisation in Process Petri Nets. IEEE Transactions on Systems, Man, and Cybernetics: Systems 43(6):1385–1398

Selic B (2007) A Systematic Approach to Domain-Specific Language Design Using UML. In: 10th IEEE Int. Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), IEEE Computer Society, Santorini Island, Greece, pp 2–9

Silva M, Colom J (1988) On the computation of structural synchronic invariants in P/T nets. In: Rozenberg G (ed) Advances in Petri Nets 1988, Lecture Notes in Computer Science, vol 340, Springer Berlin Heidelberg, pp 386–417

Smith CU, Lladó CM, Puigjaner R (2010) Performance Model Interchange Format (PMIF 2): A comprehensive approach to Queueing Network Model interoperability. Performance Evaluation 67(7):548–568

The MathWorks (2010) Matlab, `http://www.mathworks.com/`. Version R2010a

Tricas F (2003) Deadlock Analysis, Prevention and Avoidance in Sequential Resource Allocation Systems. PhD thesis, Dpto. de Informática e Ingeniería de Sistemas, Universidad de Zaragoza

Université de Montréal (2014) Stochastic Simulation in Java library. Online, `http://www.iro.umontreal.ca/~simardr/ssj/indexe.html`

Woodside M, Petriu DC, Petriu DB, Shen H, Israr T, Merseguer J (2005) Performance by Unified Model Analysis (PUMA). In: Proceedings of the 5th International Workshop on Software and Performance, ACM, New York, NY, USA, WOSP '05, pp 1–12