# Baboon, Framework Conducido por Red de Petri para Sistemas Reactivos Dirigidos por Eventos

Ing. Ariel Iván Rabinovich[1], Ing. Luis Orlando Ventre[1],Dr. Ing. Orlando Micolini[1]

[1]Laboratorio de Arquitectura de Computadoras, FCEFyN-Universidad Nacional de Córdoba
Av. Velez Sarfield 1601,CP-5000, Córdoba, Argentina
{ariel.rabinovich, luis.ventre, orlando.micolini}@unc.edu.ar

**Abstract.** Ensuring that a concurrent program is correct means that during its execution no thread suffers starvation, no set of threads falls into a deadlock and their successive states lead to the solution. To assert this, it is necessary to appeal to formal methods.
This paper proposes using a Petri Net as the logic of a reactive system, and its execution within a framework based on a concurrency monitor. It is put forward that the logical execution flow of the implemented system is driven by the execution of the model. Thus, the model transfers its properties to the system, which has been verified.

**Keywords:** Red de Petri, Sistema Reactivo, Programación Concurrente, Concurrencia, Modelo, Framework

## 1 Introducción

Este trabajo propone un framework para implementar sistemas críticos, reactivos (RS) y guiados por eventos (EDA) [1, 2]. Estos sistemas tienen que cumplir requerimientos no funcionales específicos debido a que interactúan con variables y eventos del propio sistema y del mundo exterior, donde los datos y eventos son heterogéneos y no determinísticos. Existe un gran número de propuestas dedicadas a estudiar cómo implementar estos sistemas a partir de máquinas de estados en diferentes lenguajes de programación en una amplia variedad de contextos de aplicación, tales como sistemas de control modernos [3], sistemas de control de producción [4], sistemas de control de misiones espaciales de la NASA [5], etc.

En particular, cuando se trata de sistemas paralelos y concurrentes, la generación de código a partir de máquinas de estados constituye una tarea compleja, debido a que muchos de los conceptos de especificación no están correctamente soportados, tales como eventos concurrentes, estados locales y globales, pseudo-estados históricos, o pseudo-estados de bifurcación [6, 7]. Puesto que éstos son aspectos centrales en los sistemas reactivos a implementar en este framework; es de gran importancia que el modelo los identifique. Dado que las redes de Petri (RdP) no autónomas [8] evidencian éstas características, son ejecutables [9], y escalables cuando se las expresa con la ecuación de estado extendida [10] se ha considerado que son el formalismo más conveniente.

La originalidad de éste trabajo es el diseño de un framework que integra la ejecución de una RdP no autónoma y las acciones con un monitor de concurrencia [11]. El framework ejecuta la red como el modelo lógico del sistema, basándose en la ecuación de estado extendida, de esta manera permite desarrollar RS confiables compuestos por una arquitectura probada y una lógica verificada formalmente.

La importancia del framework aquí presentado radica en la reusabilidad que se logra con la inclusión de *tópicos* para desacoplar la lógica y la política de las acciones. Reflejando la desagregación de las vistas funcionales y de comportamiento, con el fin de obtener un sistema resultante modular, simple, mantenible, validable y verificable formalmente. Para alcanzar estas características, la lógica es modelada con RdP no autónomas y la política no es integrada en la RdP. La política es determinada a partir de los conflictos presentes en la red [8].

Como antecedente a este trabajo se encuentra en [9] un estudio detallado de distintas propuestas que utilizan las RdP para la solución de RS y EDA. Asimismo, en [12] se ha desarrollado un procesador que ejecuta la RdP por hardware.

En este trabajo se ha extendido la propuesta realizada en [13], integrando en un framework las vistas del modelo con la metodología para el desarrollo de sistemas haciendo uso de la ecuación de estado generalizada. También, se pueden encontrar numerosos casos de simulación resueltos con RdP en [14].

En el siguiente apartado se presentan conceptos y notaciones fundamentales sobre RS y modelos. Luego, en las siguientes secciones, se describe en detalle la arquitectura desarrollada, los resultados obtenidos y finalmente las conclusiones.

# 2 Marco Teórico

## 2.1 Sistemas Reactivos

Un sistema reactivo típico presenta las siguientes características: interactúa continuamente con su entorno, utilizando entradas y salidas que son continuas o discretas. Las entradas y salidas son a menudo asíncronas, lo que significa que pueden arribar o cambiar valores de forma impredecible. El sistema debe ser capaz de responder a las interrupciones, es decir, a eventos de alta prioridad. Su funcionamiento y reacción a las entradas frecuentemente refleja los estrictos requisitos de tiempo [2]. La mayoría de los sistemas embebidos de tiempo real, control, supervisión, procesamiento de señales, protocolos de comunicación e interfaces hombre-máquina son sistemas reactivos.

La implementación de sistemas reactivos resulta compleja debido a que es imposible predecir o controlar el orden de llegada de los eventos externos. Esto implica la necesidad de estructuras de control y manejo de eventos, para asegurar el correcto funcionamiento según la lógica, política y acciones del sistema.

### 2.2 Modelado y Vistas de un Sistema Reactivo

La construcción de un modelo puede considerarse como una transición de ideas y descripciones informales a descripciones concretas que utilizan conceptos y terminología predefinida. En este trabajo se consideran las siguientes descripciones utilizadas en [1] para capturar la especificación del sistema:

La *vista funcional* captura el "qué": Describe las funciones, los procesos o los objetos del sistema, también llamados actividades, precisando así sus capacidades. Esta vista también incluye las entradas y salidas de las actividades, es decir, el flujo de información hacia y desde el entorno externo del sistema, así como la información que fluye entre las actividades internas.

La *vista de comportamiento* captura el "cuándo": describe el comportamiento del sistema a lo largo del tiempo, incluyendo la dinámica de las actividades, su control y comportamiento temporal, los estados y modos del sistema, y las condiciones y eventos que provocan el cambio de modos y otros sucesos. Por lo tanto, proporciona respuestas a preguntas sobre causalidad, concurrencia y sincronización. Es descripto por el *Gráfico de Estados* y en este trabajo por la RdP no autónoma.

La *vista estructural* captura el "cómo": describe los subsistemas, módulos u objetos que constituyen el sistema real y la comunicación entre ellos. Es la arquitectura del framework y todos los objetos que constituyen a las acciones. Se considera que esta vista es el modelo físico del sistema, incluye términos y nociones obtenidos del dominio de los problemas. Este modelo conduce al dominio de las soluciones.

## 3 Arquitectura de la Solución

El framework desarrollado en este trabajo, presenta una arquitectura reutilizable para el desarrollo de sistemas reactivos. Por lo que provee las interfaces necesarias para

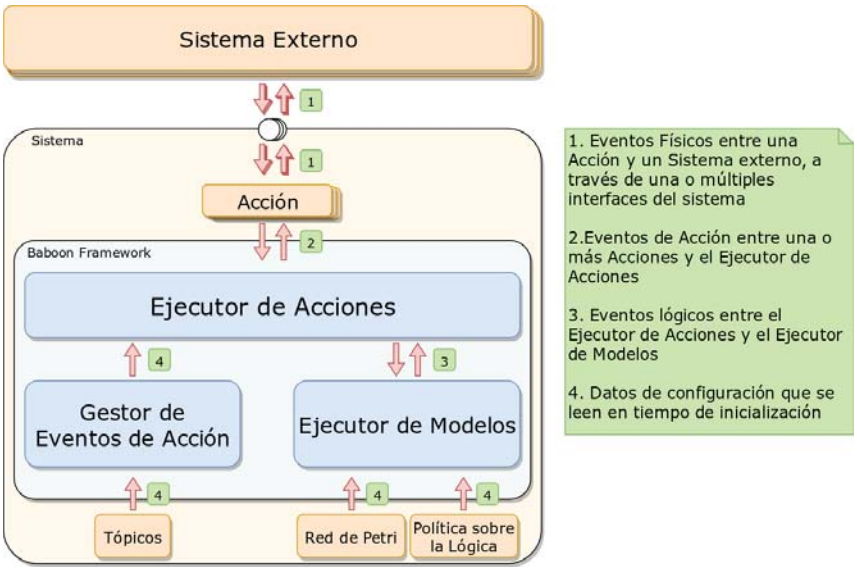



**Fig. 1.** Arquitectura de Baboon Framework (vista estructural)

interactuar con el modelo de comportamiento (RdP y política) y las acciones (vista funcional). En la Fig. 1 se observan los módulos principales de la arquitectura los que son descriptos a continuación, estos implementan las distintas vistas del modelo.

### 3.1 Ejecutor de Modelos

Las responsabilidades son: interpretar la RdP y ejecutarla según la política.
**Interpretar la RdP.** La responsabilidad de este submódulo, es transformar la descripción del modelo en un equivalente ejecutable. El formato aceptado por este submodulo es PNML [15], con él se generan las estructuras de datos para ejecutar la red mediante la ecuación de estado generalizada.
**Ejecutar el Modelo.** Si se interpreta la evolución de la RdP como una traza compuesta por una alternancia de eventos, transiciones y estados, y se asume que en cada estado se realiza una acción, y siendo que la RdP representa la lógica del sistema, estas acciones estarán coordinadas por el disparo de las transiciones habiéndose establecido una correspondencia entre los estados y las acciones.

Un conflicto emerge cuando dos o más hilos intentan hacer uso concurrente del mismo recurso para ejecutar acciones que, por la naturaleza del sistema, no pueden darse simultáneamente. Estos conflictos se resuelven aplicando una política sobre la lógica. La aplicación de la política determina la transición al estado siguiente.

Las responsabilidades de este submódulo son: computar el estado siguiente ante un evento a partir del estado actual, resolver los conflictos entre disparos y gestionar la concurrencia de acceso a la lógica y política.

Este framework está diseñado para soportar la programación de la política en tiempo de ejecución, por lo que gestiona la lógica y política de forma desacoplada.

### 3.2 Gestor de Eventos de Acción

**Eventos.** La ejecución del sistema resultante se realiza mediante el intercambio de eventos entre los distintos módulos. Estos eventos son de tres tipos:
*Eventos Lógicos.* Están inherentemente asociados a las evoluciones del modelo. Son los relacionados con las transiciones y las guardas de la RdP.
*Eventos Físicos.* Suceden en el mundo físico y representan sucesos del dominio del problema. Se comunican a través de las interfaces del software de usuario.
*Eventos de Acción.* Eventos de comunicación entre el framework y el software de usuario. Se corresponden al inicio o fin de una acción, desacoplan la vista funcional de la vista de comportamiento. El framework captura estos eventos para redirigir el flujo de ejecución del programa, logrando la inversión de control.
**Tópicos.** Un tópico es un conjunto ordenado de eventos lógicos necesarios para realizar una acción. Su responsabilidad es relacionar las acciones con los eventos lógicos, por lo cual se implementa con una estructura que contiene cuatro campos, que son: *name*, identificador unívoco; *permission*, una lista ordenada de transiciones para solicitar permiso de ejecución; *setGuardCallback*: una lista ordenada de conjuntos de guardas para gestionar la ejecución; *fireCallback*, una lista ordenada de transiciones para el aviso de finalización de ejecución.

En tiempo de inicialización, el Gestor de Eventos de Acción carga su configuración desde el archivo de tópicos provisto por el usuario.

**Traducción de Eventos de Acción a Eventos Lógicos.** Los eventos de acciones y los eventos lógicos deben vincularse y este vínculo puede cambiar a lo largo del desarrollo, implementación y mantenimiento del sistema. Por esto es necesario implementar una relación programable entre los mismos. Además se requiere soportar los casos donde la cardinalidad entre eventos de acciones y eventos lógicos es diferente. Para implementar esta relación en el framework se han incluido los *tópicos*. Estos le informan en tiempo de inicialización al ejecutor de modelos cómo hacerlo evolucionar ante el comienzo y el fin de cada acción, respetando el orden en que deben darse las evoluciones. Los tópicos constan de los siguientes campos: el campo *permission* es una lista de eventos lógicos para soportar el requerimiento de ejecutar una acción que requiere múltiples eventos. El campo, *fireCallback* es una lista de eventos lógicos opcionales que se desencadenan al finalizar la ejecución de una acción sólo si están las condiciones dadas para que sucedan (transiciones sensibilizadas). El campo, *SetGuardCallback* contempla el caso de las tareas complejas. Al ser una lista de listas de guardas, cada lista de guardas se establece en la finalización de la sub-tarea correspondiente dentro de la ejecución de una tarea compleja.

La decisión de introducir las guardas en los tópicos permite la toma de decisión dinámica de las acciones, esto relaciona la lógica con el estado de las entidades físicas, lo que mejora la capacidad de expresión del modelo.

### 3.3 Suscriptor de Acciones

Este módulo brinda al usuario las interfaces para construir una acción y suscribirla al tópico correspondiente. Esto se hace durante la fase de inicialización del sistema.

El *Ejecutor de Acciones* utiliza estas suscripciones para lanzar los eventos de acción, y ejecutar las secciones correspondientes del programa.

### 3.4 Ejecutor de Acciones

Este módulo se relaciona con las acciones y el ejecutor de modelos. Con las acciones a través de los eventos de acciones para coordinar su ejecución y con el ejecutor de modelos, a través de los eventos lógicos, para sincronizar con el modelo el comienzo y fin de las acciones.

**Acción.** Una acción es un componente de software ejecutable, desarrollado por el usuario del framework que realiza una actividad del dominio del problema. Estructuralmente está compuesta por un objeto y un método perteneciente a ese objeto. A su vez contiene todos los parámetros necesarios para realizar la llamada a este método.

El método de una acción se clasifica según dónde se origina su ejecución:

*Suceso o Happening.* su ejecución se origina por un evento físico que ingresa al sistema y la llama explícitamente. El ejecutor de acciones captura la llamada

(utilizando AOP) y consulta al modelo si debe permitir o no la ejecución. Debe estar anotada como *@HappeningController*
*Tarea o Task.* su ejecución se origina sin la necesidad de un evento físico, es decir internamente desde el ejecutor de acciones.[5]. La responsabilidad de permitir o rechazar la ejecución es del modelo y de su ejecutor. Debe estar anotada como *@TaskController.*
*Tarea Compleja o ComplexSequentialTask.* Si una tarea está compuesta por N tareas menores que naturalmente se realizan secuencialmente, existe la posibilidad de agruparlas en una Tarea Compleja. El framework provee las interfaces para crear una tarea compleja, agregandolas.

### 3.5 Proveedor de Guarda

Si el modelo construido con la RdP emplea guardas, debe existir en el programa un método por cada guarda con el fin de calcular el valor que el framework actualiza. Este método es el proveedor de la guarda y es anotado como *@GuardProvider("nombre_guarda")*. Retorna un valor booleano y es registrado en el tópico de la acción (esto se realiza por reflexión en tiempo de inicialización).

Se debe considerar que la guarda es usada por la RdP, por lo cual se abstrae al usuario de consultarla manualmente durante la ejecución del programa.

### 3.6 Ejecución de una Acción

Para cumplir las restricciones del sistema, al ejecutar una acción, se realizan la siguiente secuencia de pasos: el ejecutor de acciones captura el evento de acción, luego lo traduce a eventos lógicos de permiso mediante el tópico, seguidamente se emiten los eventos lógicos de permiso hacia el monitor para luego si el modelo permite la ejecución de la acción emitir un evento lógico de permiso concedido hacia el ejecutor. De otra manera queda la petición a la espera de las condiciones. A continuación el ejecutor recibe el evento de permiso concedido, se ejecuta la acción de usuario para realizar la actividad y finaliza la acción de usuario. Para concluir se emiten los eventos lógicos de finalización de ejecución al ejecutor del modelo: primero los de guardas y luego los de transiciones

## 4 Diseño de un Programa con Baboon Framework

Baboon Framework está diseñado para desarrollar sistemas con POO. Las principales características del diseño obtenidas con este framework son: los objetos no contienen lógica, sino que ésta está en la RdP; la concurrencia es transparente al usuario y es gestionada por el *Ejecutor de Modelos* y la interacción entre objetos se hace por el intercambio de eventos realizados por el ejecutor de acciones.

Los pasos del proceso de diseño de un programa con Baboon Framework son: primero se determinan los objetos y acciones que forman parte del sistema, luego se

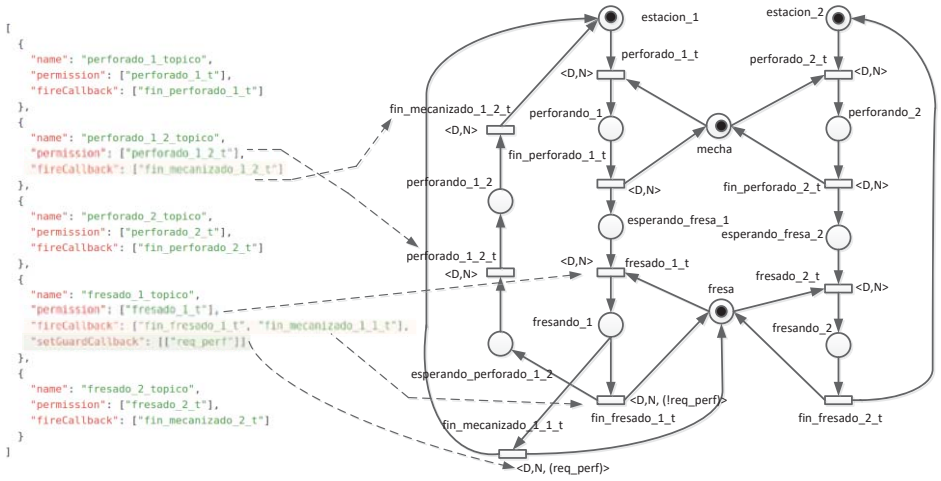
construye y valida la lógica del sistema con una RdP, a continuación se generan los tópicos y se etiquetan las acciones del sistema, en el paso siguiente se construyen los proveedores de guardas, seguidamente se define una política en las transiciones en conflicto, se escribe la inicialización del programa y finalmente se optimiza la cantidad de hilos haciendo uso de las tareas complejas.

## 5 Caso de Estudio

Como caso de estudio se presenta el diseño del software de un centro de mecanizado automatizado con las siguientes características: una máquina con dos estaciones de trabajo donde la primera estación puede realizar hasta dos procesos distintos. La segunda estación sólo hace un mecanizado simple.

El flujo de trabajo de la estación 1 decide en tiempo de ejecución cuál mecanizado realizar. Para esto emplea la guarda *req_perf* (requiere perforación), ambas estaciones comparten las herramientas (mecha y fresa) por lo que no las pueden utilizar simultáneamente. Si una herramienta está ocupada, la estación de trabajo que la requiera debe esperar hasta que dicha herramienta se encuentre disponible.

En la en la Fig. 2(a) se observan los tópicos del sistema, mientras que en la Fig. 2(b) se observa la RdP que modela al sistema.




(a) Tópicos del Caso de Estudio (b) RdP del Caso de Estudio

**Fig. 2.** Relación entre la RdP y los tópicos

En la Fig. 2(a) se observa, que para la definición de los tópicos se utiliza JSON, donde en la definición de cada artefacto se respeta su pertenencia, es decir, que para nombrar un tópico se usa la palabra tópico, para el método de interés se usa la palabra método, mientras que para los eventos se distingue su naturaleza con una F, A o L según sea un evento físico, de acción o lógico.

En la Fig. 2 se advierte la relación entre cada segmento de la RdP con el tópico y sus componentes correspondientes. Estas relaciones se muestran con las flechas de líneas de guiones.

### 5.1 Código del Usuario

En la Fig. 3 se observa una plantilla para la clase Fresadora. Esta clase contiene las acciones relacionadas al fresado de una pieza. En la Fig. 2 las flechas con guiones muestran cómo estas acciones están suscritas a un tópico, cuyos eventos están coordinados por la RdP.

```
public class Fresadora {
  // campos y otro métodos de la clase
  @TaskController
  public void fresado1Metodo() {
  // algoritmo de fresado sobre el mundo físico
  // emisión de eventos físicos con resultados en el mundo real
  }
  @GuardProvider("req_perf")
  public boolean requierePerforacion() {
    boolean resultado = false;
    // algoritmo para definir si se requiere hacer una perforación
    return resultado;
  }
}
```

**Fig. 3.** Clase de Acción

```
public class CentroDeMecanizado implements BaboonApplication {
  private Fresadora fresadora;
  @Override
  public void declare() { // tomar los recursos que requiera el sistema
    fresadora = new Fresadora();
    try {
      BaboonFramework.createPetriCore("/ruta/a/mi/red/de/petri.pnml",
        petriNetType.PLACE_TRANSITION, PoliticaPersonalizada.class);
      BaboonFramework.addTopicsFile("/ruta/a/mis/topicos.json");
    } catch (BadPolicyException | BadTopicsJsonFormat | NoTopicsJsonFileException e) {
      // manejar la excepción y terminar programa
    }
  }

  @Override
  public void subscribe() {
    try {
      // suscribir las las acciones a los tópicos usando BaboonFramework.subscribeControllerToTopic
      // crear tareas complejas usando BaboonFramework.createNewComplexTaskController
      // agregar tareas a una tarea compleja usando BaboonFramework.appendControllerToComplexTaskController
      BaboonFramework.subscribeControllerToTopic("fresado_1_topico", fresadora, "fresado1Metodo");
    } catch(NotSubscribableException e) {
      // manejar la excepción y terminar programa
    }
  }
}
```

**Fig. 4**. Clase de Inicialización

En la Fig. 4 se muestra un prototipo de la clase principal del programa, donde se le provee al framework el modelo de RdP, el archivo de tópicos y se realizan las suscripciones de acciones a tópicos.

## 6 Conclusiones

En Baboon, se extendió el trabajo desarrollado en [13] utilizando la ecuación de estado extendida de la RdP con lo cual se soportan todos los tipos de brazos. En consecuencia se obtiene mayor capacidad de expresión en la lógica de los sistemas a modelar frente a las POPN. Se desacopló la vista funcional de la vista de comportamiento, haciendo uso de los tópicos lo que simplifica la comprensión y mantenibilidad del sistema. Se ha mantenido el formalismo de la RdP no autónoma integrada en el framework, por lo tanto la lógica del sistema conserva la verificación formal realizada en el modelo. De acuerdo a los casos de estudio implementados son destacables las siguientes características alcanzadas con el framework: verificable, comprobable, mantenible, modular, basado en una arquitectura probada y además se logró reducir el tiempo de desarrollo, debido a que no se requiere codificar la lógica.

## 7 Resultados

Los casos implementados en [16, 17] fueron re-diseñados con el framework Baboon desarrollado en este trabajo, los que son: un sistema de control de acceso inteligente y un sistema marcapasos. Asimismo se implementó el sistema de celda flexible de producción aquí presentado.

Los resultados obtenidos han sido los siguientes: dado que el framework interpreta los modelos de RdP expresados en lenguaje PNML es directo implementar la lógica a partir del modelo realizado con un simulador. Esto facilitó el desarrollo y el mantenimiento de los sistemas.

Puesto que las propiedades de las RdP fueron verificadas formalmente con el simulador (interbloqueo, inanición, acotada, vivacidad, etc.), las acciones han sido desacopladas y no contienen lógica, y la política solo resuelve los conflictos sin modificar la lógica; éstas propiedades se han mantenido en el sistema resultante.

El uso del framework implementa la inversión de control: con lo que se obtuvo un flujo robusto y probado, y una arquitectura reutilizable.

El framework ha sido diseñado y construido considerando la relevancia de la simplicidad para su uso por lo que las interfaces de las APIs son consistentes, existe documentación online detallada y el proceso de construcción y gestión de dependencias están automatizados.

El código fuente es libre, licenciado bajo Apache 2.0 y disponible online en: https://github.com/airabinovich/java_petri_concurrency_monitor, https://github.com/juanjoarce7456/baboon.

Como resultado se destaca la escalabilidad de la lógica. Puesto que la lógica esta implementada a partir de la ecuación de estado extendida y ésta es una ecuación matricial (4 matrices de enteros de dimension: plazas * transiciones + 1 matriz de

floats con lógica temporal de dimensión: transiciones * 3). Para el caso de un sistema de 1000 plazas por 1000 transiciones se requieren 4 MB.

## 8 Trabajo Futuro

A continuación se mencionan las principales líneas de investigación sobre las que se continúa a partir de este trabajo: permitir cambios en la lógica en tiempo de ejecución, extender la ejecución a RdP coloreadas con el objetivo de explotar su capacidad de expresión manteniendo la escalabilidad del sistema resultante y re-implementar el proyecto utilizando otras arquitecturas como: web services, librería, driver, etc.

## Bibliografía

1. Harel, D., Politi, M.: Modeling reactive systems with statecharts: the STATEMATE approach. McGraw-Hill, Inc. (1998)
2. Wieringa, R.J.: Design methods for reactive systems: Yourdon, statemate, and the UML. Elsevier (2003)
3. Wen, J.T., Mishra, S.: Intelligent Building Control Systems: A Survey of Modern Building Control and Sensing Strategies. Springer (2017)
4. Zhou, M., Wu, N.: System modeling and control with resource-oriented Petri nets. (2018)
5. Siewert, S.: Real time embedded components and systems. Cengage Learning (2016)
6. Niaz, I.A.: Automatic code generation from UML class and statechart diagrams (2005)
7. Niaz, I.A., Tanaka, J.: Mapping UML statecharts to java code. In: IASTED Conf. on Software Engineering, pp. 111-116 (2004)
8. David, R., Alla, H.: Discrete, continuous, and hybrid Petri nets, Springer Science (2010)
9. Micolini, O.: PhD thesis Arquitectura asimétrica multicore con procesador de Petri. vol. Doctor. Facultad de Informática, La Plata, Argentina (2015)
10. Micolini, O., Cebollada, M., Eschoyez, M., Ventre, L.O., Schild, M.: Ecuación de estado generalizada para redes de Petri no autónomas y con distintos tipos de arcos. In: XXII Congreso Argentino de Ciencias de la Computación (2016)
11. Peter A. Buhr, M.F.: Monitor Classification. ACM Computing Surveys 27, 63-107 (1995)
12. Micolini, O., Daniele, E.N., Ventre, L.O.: Modular Petri Net Processor for Embedded Systems. pp. 199-208. Springer International Publishing (2017)
13. Micolini, O., Caro, M.F., Furey, I., Cebollada, M.: Generación de Código de Sistemas Concurrentes a partir de Redes de Petri Orientadas a Procesos. In: XLIII Jornadas Argentinas de Informática e Investigación Operativa (43JAIIO)-XV Argentine Symposium on Technology (2014)
14. Universität Hamburg, https://www.informatik.unihamburg.de/TGI/PetriNets/applications/
15. PNML Org , http://www.pnml.org/grammar.php
16. Micolini, O., Ventre, L.O., Ludemann, M., Viano, J.I.R., Bien, C.C.: Case Study of Reactive and Embedded System Design Modeled with Petri Nets. In: 2018 IEEE International Conference on Automation/XXIII Congress of the Chilean Association of Automatic Control (ICA-ACCA), pp. 1-7. IEEE (2018)
17. Micolini, O., Ventre, L.O., Ludemann, M.: Methodology for design and development of Embedded and Reactive Systems Based on Petri Nets. In: 2018 IEEE Biennial Congress of Argentina (ARGENCON), pp. 1-7. IEEE (2018)