TECHNISCHE UNIVERSITÄT ILMENAU
Faculty of Computer Science and Automation
System and Software Engineering Department

Group Studies

Winter Semester 2013-2014

TimeNET implementation for Transforming UML State Machines into Stochastic Petri Nets for Energy Consumption Estimation of Embedded Systems

Andres Canabal
Matriculation No. 50981
Nikhil Rane
Matriculation No. 51384
Zhou Fan
Matriculation No. 51418

Prof. Dr.-Ing. habil. Armin Zimmermann
Dipl.-Ing. Dmitriy Shorin

Ilmenau, March 25, 2014

Contents

List of Figures

List of Tables

1 Introduction

Energy consumption in embedded and mobile systems is of prime concern today. With new and tiny embedded systems like wireless sensors which are bound to do enormous work with as low energy consumption as possible, this non functional property of the systems requires analysis and design. As stated in [SZM12], it is important to evaluate architectural and other design decisions in all phases of the development process based on a good prediction of the energy consumption of an embedded system. This document is based on implementation of the idea proposed in [SZM12], which concentrates on early design steps in major architectural decisions, that may lead to significant impact on the overall system's energy consumption. As UML (Unified Modeling Language) models are not well-defined semantically for a specification stochastic processes, [SZM12] proposes UML models to be transformed into a model for which analysis algorithms exist, specificly into SPNs (Stochastic Petri Nets) [BK96].

For implementing these ideas, TimeNET (Timed Net Evaluation Tool) [ZK07] is used, first with the creation of two basic UML models (operational and application), and then with a transformation into a SPN model. For better understanding an example is given, with specific detail on its creation and transformation into SPN.

2 Concepts

In this chapter, we introduce all models used, and the tools where this solution was implemented.

2.1 Petri Nets

A Petri Net (also known as a place/transition net or P/T net) is one of several mathematical modeling languages for the description of distributed systems. A Petri Net is a directed bipartite graph, in which the nodes represent transitions (i.e. events that may occur, signified by bars) and places (i.e. conditions, signified by circles). [Mur89]

Formerly used for the purpose of describing chemical processes, now Petri Nets have been found extended applications in many diverse areas like:

- Software design
- Workflow management systems
- Process modeling
- Data analysis
- Concurrent programming
- Reliability engineering
- Discrete process control
- Simulation

2.2 Stochastic State Machines

sSMs [TJZ07] (stochastic State Machines) are the combination of UML State Machines with the UML Profile for Schedulability, Performance and Time (SPT). Elements in sSMs include simple state, composite states, and events. States may have optional internal activities: entry, do and exit activities. Stereotypes from the SPT profile are used to annotate quantitative aspects. Two aspects related to the quantitative analysis are given in sSMs: timing and probability.

2.3 TimeNET

TimeNET is a graphical and interactive toolkit for modeling of Stochastic Petri Nets (SPNs) and Stochastic Colored Petri Nets (SCPNs). The project has been motivated by the need for powerful software for the efficient evaluation of timed Petri Nets with arbitrary firing delays. The current version is TimeNet 4.1 and runs on both Linux and Windows platforms. [ZK07]

TimeNET implements a standard discrete-event simulation for performance evaluation of SCPN models. The GUI is based on Java. Petri Net classes are defined by an extendable XML schema which affects the behavior of the graphical user interface. In short, a model is a well-formed XML document which is validated automatically.

3 Design

As explained in [SZM12], two models are used for the specification of the system, `Operational model` and `Application model`. As describe below, these models were added to TimeNET, based on sSM models. These models contain states with attributes, transitions between them, with `choices` and `join` states for flow description.

3.1 Operational Model

The `Operational models` specifies all run modes of a processor (microcontroller), the possible state changes, and their associated energy consumption (as well as transition times, if applicable). The details for this model could be referenced from data sheets and it has to be constructed only once for a specific microcontroller. Using `ResouceUsage` stereotype, a state of this model can specify the execution time (`execTime`) and power consumption (`powerPeak`), that represent how long the execution of the state will require (in seconds) and the necessary power required (in milliWatt). Depending on the aims, the maximum, minimum or an average value can be considered. After a `choice` state, a probability (`prob`) attribute must be given using the `GaStep` stereotype.

As example, we present the model in Fig. 3.1, with the given attributes listed in Table **??**.

3.2 Application Model

The `Application model` captures effect of the controlling software. It describes which steps are taken and the amount of time spent in each state, along with transitions between them. Thus it is constituted by `Operational model` states used and their respective durations. An `Operational model` must be referenced from it, and all states must be linked to one

State	powerPeak	execTime	prob
A	10		
B	7	4	0.7
C	3	5	0.3
D	3	3	
E	4	5	
F	6		

Table 3.1: States and its attribute values in operational model

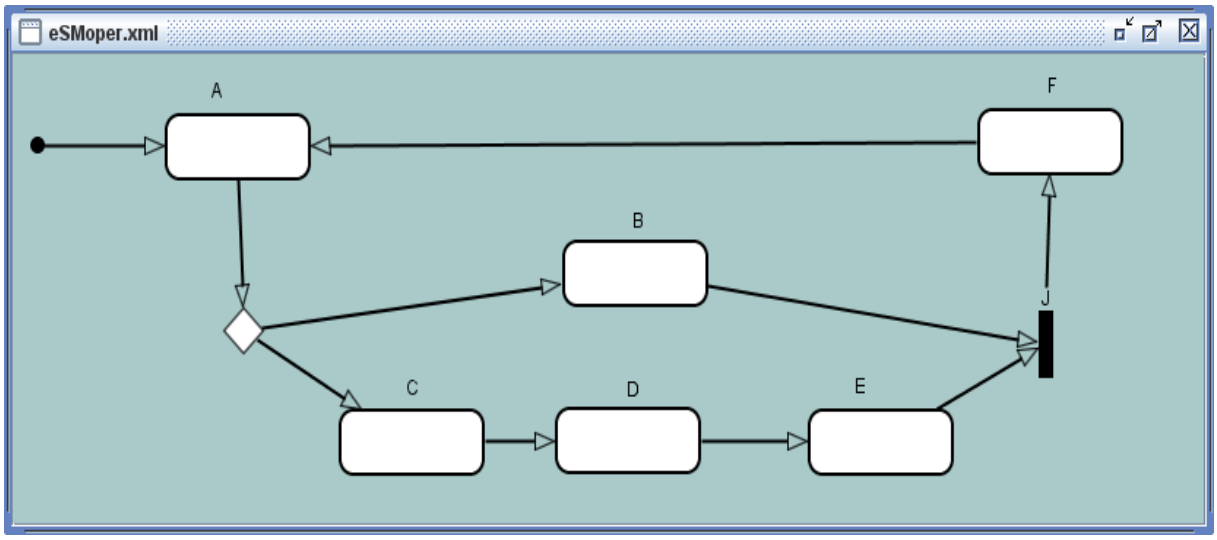

Figure 3.1: Operational model

State	execTime	prob
A1	10	
F1	2	
A2	10	
B2	4	0.7
C2	5	0.3
E2	5	
F2	2	

Table 3.2: States and its attribute values in application model

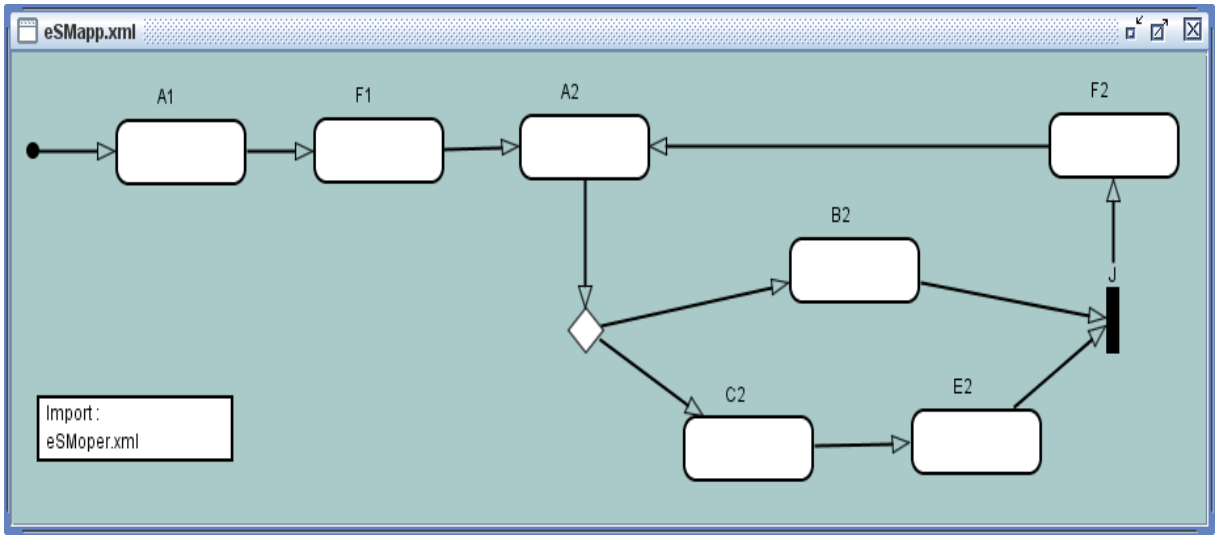

Figure 3.2: Application model

Application model state	Operational model state
A1, A2	A
B2	B
C2	C
E2	E
F1, F2	F

Table 3.3: Application model state and its corresponding operational model state

of the states in the `Operational model`. The reference is implemented by a `Import` element, and there is a name convention for the state reference, in which the name of a state in the `Application model` must have as prefix a name of a state in the referenced `Operational model`. This model can specify `execTime` with the `ResourceUsage` stereotype, and `prob` with `GaStep`, and it will override the one in the `Operational model` if it exists. Also, all transition in the `Application model` must be a valid transition in the referenced `Operational model`, although not nesesarly literal transitions. A transition in the `Application model` can represent more than one transition in the `Operational model`.

As example, we present the model in Fig. 3.2.

This `Application model` references the previous `Operational model` example. By the name convention we can state the relation betwen the states in the `Application model` and the states `Operational model`, described in Table 3.3.

Here we can see how a transition in the `Application model` may represent more than one transition in the `Operational model`. In the `Application model` example, there is a transition betwen **A1** and **F1**, but in the `Operation model`, there is not a direct transition betwen **A** and **F** states, but there is a possible path (even more than one) between them.

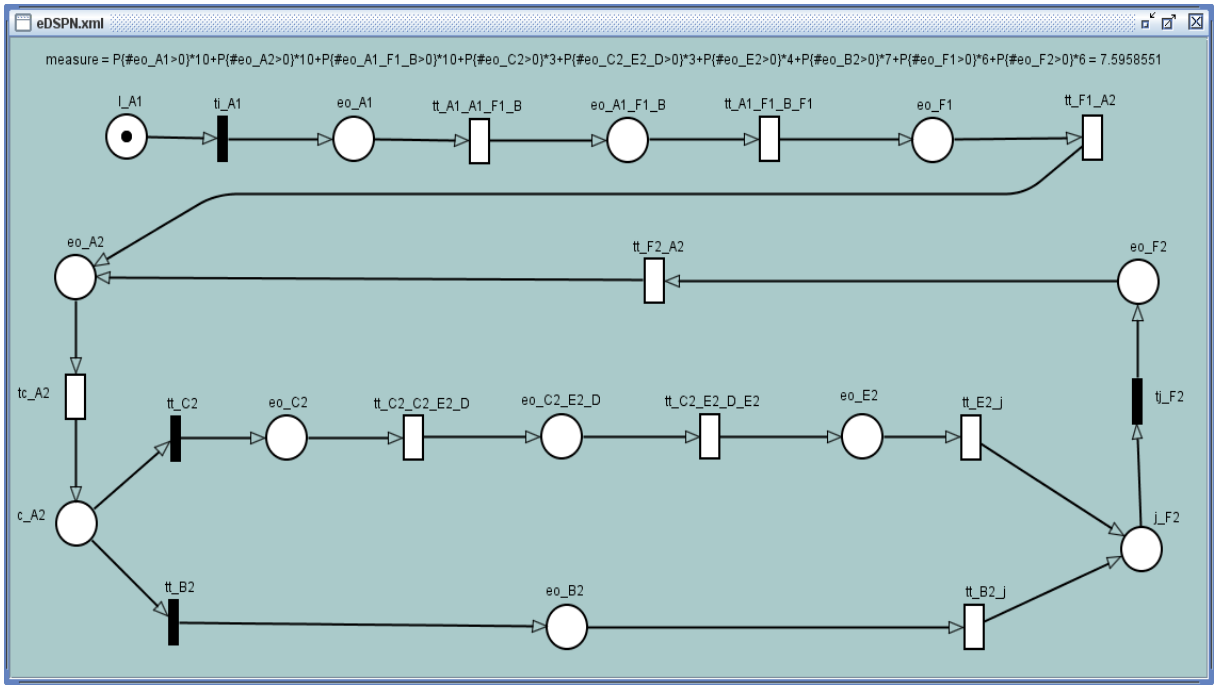

Figure 3.3: Transformed eDSPN model

3.3 Transformation

After the `Operational` and `Application` models are specified, a transformation into an `eDSPN` can be done. This transformation is implemented as a functionality in `Application model`. The aproach for this is, first combine both `Operational` and `Application` models into one, with all the information that is needed, and then transform each of its basic element into a basic element of the `eDSPN model`. Also a `measure` is created, with the function for the quantitative power consumption evaluation. All the prefix names used were chosen to keep coherence with similar transformation.

In Fig. 3.3, we present the output model of the transformation of previously shown examples. Further, we explain each of the steps for the transformation.

3.3.1 Combination of Models

The `Application model` is combined with the `Operational model` to create a complete model with all the information nedded. First, the ”missing states” are added. A missing state situation occurs when a transition in the `Application model` represents more than one transition in the `Operational model`. As we can seen in the given examples, there is a transition from states **C2** to **E2** in the `Application model`, but there is no direct transition from states **C** to **E** in the `Operational model`, so a state associated with the state **D** is added, this new state will have attributes from the `Operational` states, and a name composed by the beginning and end states of the `Application` model, plus the `Operational` state name. In the example it will be **C2_E2_D**, and it will have the attribute values from state **D**.

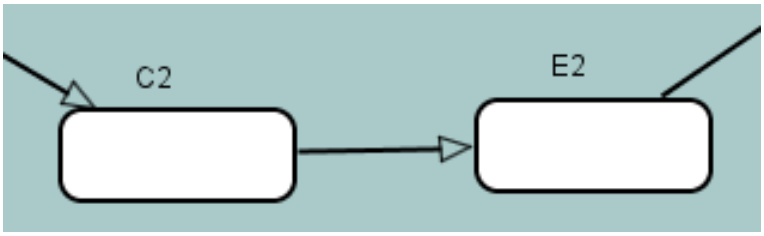

Figure 3.4: Transition with one missing state

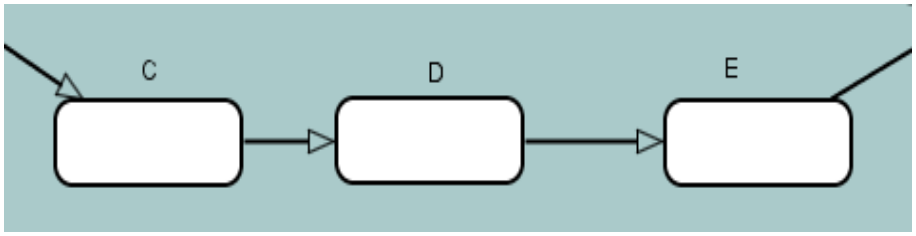

Figure 3.5: Transition in referenced operational model

Figures 3.4 and 3.5 illustrate this scenario.

This process can be more complex if there is more than one possible path. As shown in Fig. 3.2, there is a transition from **A1** to **F1** in the `Application model`, but two possible paths to make this transition with the referenced states in the `Operational model`.

Figures 3.6 and 3.7 detail the scenario.

This is solved choosing the least energy consumption path. To accomplish it, each state is given a value which is equal to its `powerPeak` times its `execTime`, and then a Dijkstra algorithm is executed to compute the optimal path. In the example,

Path A $\rightarrow$ B $\rightarrow$ F = 7$*$4 = 28
Path A $\rightarrow$ C $\rightarrow$ D $\rightarrow$ E $\rightarrow$ F = 3$*$5 + 3$*$3 + 4$*$5 = 43

Hence, first path will be choosen, and a state **A1_F1_B** will be added.

After all missing states are added, the transformation of each of the states into eDSPN elements takes place.

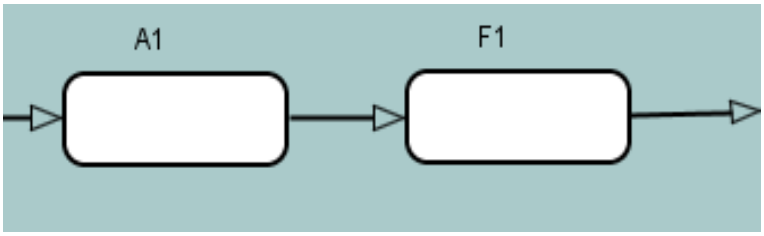

Figure 3.6: Transition with multiple possible paths

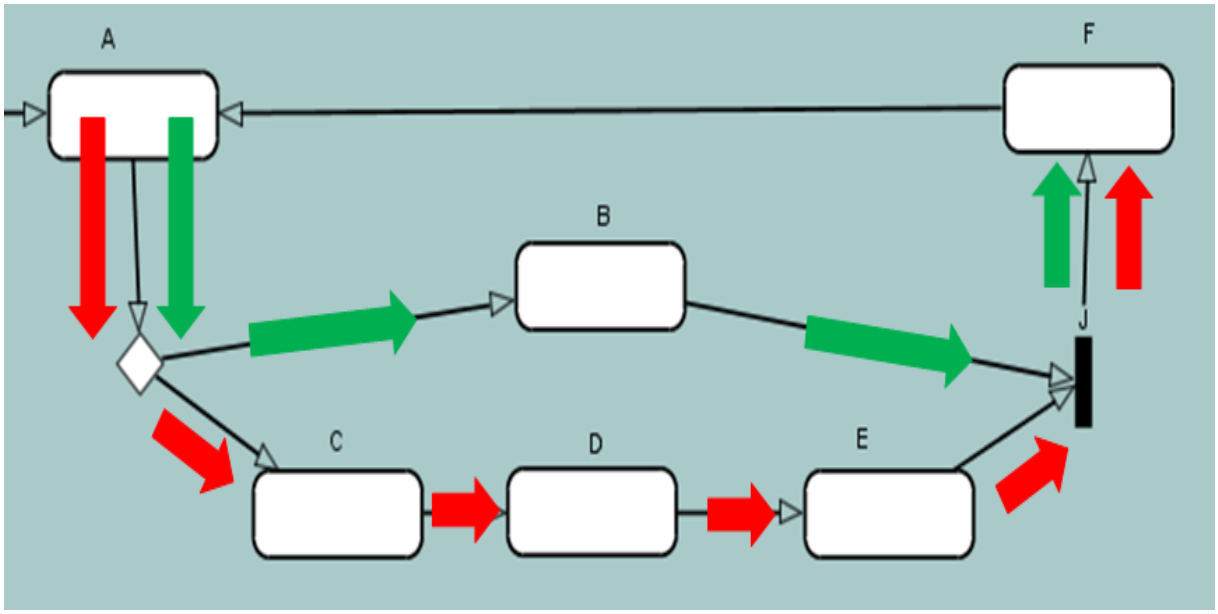

Figure 3.7: Transition with multiple possible paths in operational model

3.3.2 Transformation of a State

A state in the model is transformed into a place in the eDSPN, with the name of the state with the prefix `eo_`, and `InitialMarking` of `0`.

3.3.3 Transformation of a Transition from Initial State to a State

The Initial State, with the transition to another state is transformed into a place with the prefix `I_` and the name of the state and `InitialMarking` of `1`, followed by an immediate transition with the prefix `ti_` connecting the other state.

3.3.4 Transformation of a Choice State

A `choice` state is transformed into a place with the prefix `c_` and the name of the state where this `choice` is coming from, and `InitialMarking` set to `0`. An `exponential transition` will be added from the place this `choice` is coming, with prefix `tc_` and the name of the state associated, and a `delay` set to the execution time of that state. Also, an `immediate transition` will be added for each outgoing transition from the `choice`, each with the prefix `tt_` folowed by the name of the state the transition ends in, and the `weight` set to the probability of that state.

3.3.5 Transformation of a Join State

A `join` state is transfomed into a place with the prefix `j_` followed by the name of the state the `join` ends in, and `InitialMarking` set to `0`. An `immediate transition` is added to that state, with the prefix `tj_` followed by the name of that state, and a `weight` set to `1`. Also, an `exponential transition` will be added for every transition coming to this `join`, with the prefix `j_` followed by the name of the state it starts in and the postfix `_j`, with the `delay` set to the execution time of that state.

3.3.6 Adding the measure

After all the transformations take place, the performance measure is added. This measure is the sum of the probability of each state from the initial model times the power consumption of it. As shown in the example, if in the model there was a state **A1**, with a `powerPeak` of `10`, a term in the measure will be `P{eo_A1 >0} * 10` . Not all states will add a term, only the ones with a power consumption attribute, so the pseudo states like `initial` state, and any `join` or `choice` state will not add a term; but any missing state added will.

The measure expression for our example will be,

P{#eo_A1 >0}∗10 + P{#eo_A2 >0}∗10 + P{#eo_A1_F1_B >0}∗10 +
P{#eo_C2 >0}∗3 + P{#eo_C2_E2_D >0}∗3 + P{#eo_E2 >0}∗4 +
P{#eo_B2 >0}∗7 + P{#eo_F1 >0}∗6 + P{#eo_F2 >0}∗6

3.4 eDSPN Model

eDSPNs stands for **extended Deterministic and Stochastic Petri Nets** and is of interest in the area of real-time systems. The term was introduced by German and Lindemann[GL94]. It is an extended version of DSPNs (Deterministic and Stochastic Petri Nets) which include constant timing and exponentially distributed timing. Timed transitions in eDSPNs are either exponential or immediate, which makes it especially capable of working on real-time system.

As explained, an eDSPN model is created from the transformation of the previous `Operational` and `Application` models.

3.4.1 Stationary Analysis

One important function we can now run with this eDSPN model is the **Stationary Analysis**. This functionality will compute the measure expression, giving us a value regarding the power consumption of the whole system. By default, these values are in milliWatt (mW), it could be W, kW, etc. as well. The measure is a calculated value of power consumption of the system. In the example explained above, the result is 7.5958551, which denotes that the power consumption of the system equals 7.5958551 mW. If there is a change in the power consumption, different results would be generated. In this way, the most power efficient system configuration can be found.

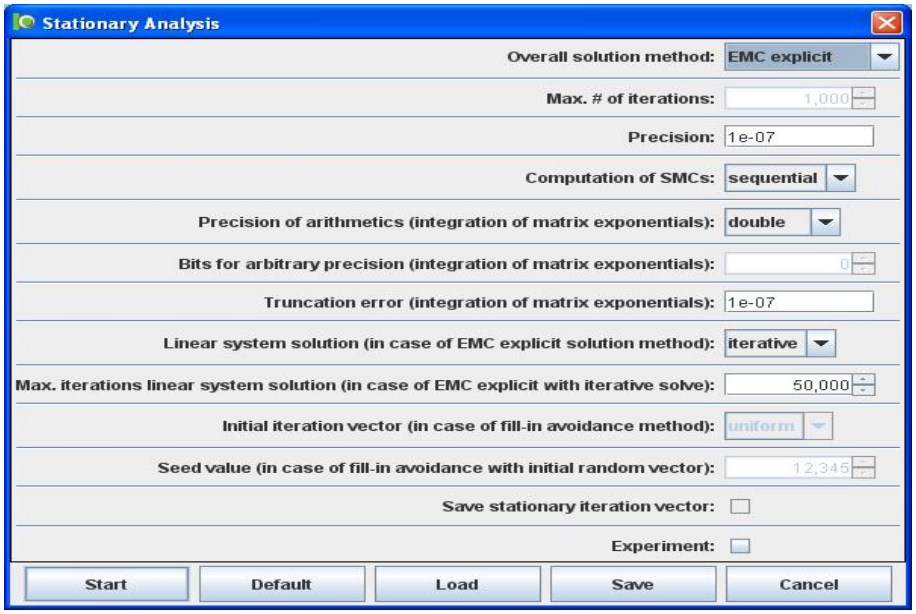

Figure 3.8: Parameters for stataionary analysis

3.4.2 Results

The parameters passed for running Stationary Analysis are shown in Fig. 3.8. The results are generated in a file with extension ".RESULTS" and contains following values:

measure = 7.5958551
THROUGHPUT RESULTS OF TIMED TRANSITIONS
(Throughput of immediate transitions have not been computed and are denoted by 0.0)
0.051813446509885
0.036269417919162
0.051813468504943
0.015544042690313
0.000000015093159
0.000000022457713
0.000000007364554
0.015544036251070
0.015544040619743
0.000000000000000
0.000000000000000
0.000000000000000
0.000000000000000

4 Implementation

As state before, all this functionality is implemented in TimeNET. The `Operational` and the `Application` models were added as models to the tool as `eSMoper` and `eSMapp` respectively. Both models are based on the `sSM` model. Basically they are a clone of this model, with restrictions on the components they can have, and some modifications in the functionality. `sSM` allows some elements that were removed, like `Composite State`, `Final State` and so on. Also, the `Import` element was added to the `eSMapp` model, to be able to reference the `Operational model`. The **transform to eDSPN** function is removed from the `eSMoper` model, and modified in the `eSMapp` model to fulfill the specification. A utility `util` package was added to this model, where the Dijkstra algorithm is implemented.

4.1 Code

The code for implementing this functionality has been added in namely two packages:

- netclasses.eSMapp
- netclasses.eSMoper

Both these packages are a part of `Netclasses` project in TimeNET repository.

The code for Dijkstra's algorithm lies in `netclasses.eSMapp.util` package of `Netclasses` project in TimeNET.

4.2 Steps to create an Operational Model

As explained in earlier chapters, the `Operational model` models the hardware part of the system. This type of model can be created from TimeNET GUI by clicking on **File** → **New** and then selecting **eSMoper** from the dropdown menu. This is shown in Fig. 4.1.

After this, a blank model appears with all elements explained in Chapter 2 except `Import`. The user can now create an `Operational model` with all states, join, decision points, etc. and save it for use in `Application model`.

4.3 Steps to create an Application Model

An `Application model` is created in the same way as the `Operational model`. Except, in the dropdown, the user needs to select **eSMapp** instead of **eSMoper**.

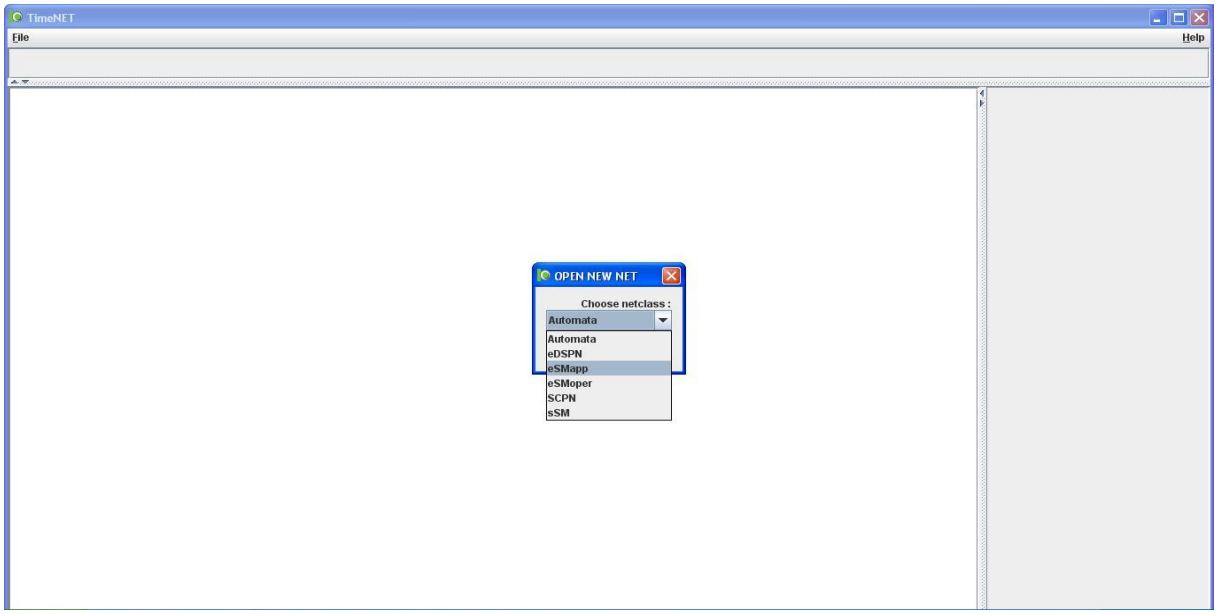

Figure 4.1: Creating Models in TimeNET

4.4 Steps to run stationary analysis

For performing a stationary snalysis of an eDSPN model, the TimeNET tool provides following simple steps:

1. Open the eDSPN XML file in TimeNET
2. Once open, click on **Evaluation** menu in the menu bar and select **Stationary Analysis**
3. A dialog box for settings appears; enter values as desired for various precision levels and hit **Start**
4. A new dialog box appears with **Analysis Output**

A sample output for the results is shown in Section 3.4.2.

5 Conclusions

TimeNET now contains two new classes, `eSMoper` and `eSMapp`, based on `sSM` class. These classes can be used to describe the hardware behaviour and specific application of a system, as the `Operational` and `Application` models, in which the user can specify the execution time and energy consumption of each state.

With these models, a Petri Net with all the relevant information from the combination of both previous models can be created. During this transformation, missing paths and states in the `Application model` will be taken into account, and if there is more than one possible path, the most energy efficient will be chosen. In this new model, a measure will be created, that summarize the energy consumption of the whole system. This measure can be then evaluated with `Stationary Analysis`, and with it, the most power efficient system configuration can be found.

Bibliography

[BK96] Falko Bause and Pieter S. Kritzinger. Stochastic petri nets. In *Stochastic Petri Nets*, pages 163–171. 1996.

[GL94] Reinhard German and Christoph Lindemann. Analysis of stochastic petri nets by the method of supplementary variables. In *Performance Evaluation*, page 317–335. May 1994.

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, volume 77, pages 541 – 580, April 1989.

[SZM12] Dmitriy Shorin, Armin Zimmermann, and Paulo Maciel. Transforming UML State Machines into Stochastic Petri Nets for Energy Consumption Estimation of Embedded Systems. In *The Second IFIP Conference on Sustainable Internet and ICT for Sustainability*, Pisa, Italy, October 2012.

[TJZ07] J. Trowitzsch, D. Jerzynek, and A. Zimmermann. A Toolkit for Performability Evaluation Based on Stochastic UML State Machines. In *Proceedings of the 2Nd International Conference on Performance Evaluation Methodologies and Tools*, ValueTools '07, Nantes, France, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[ZK07] Armin Zimmermann and Michael Knoke. *TimeNET 4.0: A Software Tool for the Performability Evaluation with Stochastic and Colored Petri Nets*. Technische Universität Berlin, August 2007.