Colored Petri Nets Design Studio

Ronald Picard
CS 6388 Model-Integrated Computing
Vanderbilt University
Nashville, TN, USA
ronald.s.picard@vanderbilt.edu

Bernard Serbinowski
CS 6388 Model-Integrated Computing
Vanderbilt University
Nashville, TN, USA
bernard.serbinowski@vanderbilt.edu

Colored Petri Nets are a directed graph modeling approach utilized to reason about race conditions and deadlocks within concurrent systems. We present a Design Studio approach to develop, simulate, and analyze Colored Petri Nets. Our implementation consists of a modeling environment developed within WebGME, and a series of Python plugins for the simulation and analysis of Colored Petri Nets. The target audience for this paper and tool are for those seeking to reason about dead locks and race conditions within concurrent systems.

Keywords—Colored Petri Nets, WebGME, Meta-Modeling, Domain Specific Modeling, Design Studio, Simulation, Analysis

I. Introduction

Petri nets were originally developed by Carl Adam Peri in 1962 and were the subject of his dissertation. [1] Since that time, Petri nets have been extended into Colored Petri Nets (CPNs) which are a backwards compatible specification that provide additional functionality. Colored Petri Nets have been applied to a wide variety of applications including: office automation, work-flows, flexible manufacturing, programming languages, protocols and networks, hardware structures, real-time systems, performance evaluation, operations research, embedded systems, defense systems, telecommunications, Internet, e-commerce and trading, railway networks, and biological systems. [1] The Colored Petri Net Design Studio described in this paper is focused on concurrent systems.

II. Colored Petri Nets

A. Basic Colored Petri Net Concepts

A Colored Petri Net (abbrev. CPN) is a collection of directed arcs connecting places to transitions and transitions to places. Places may hold colored tokens. Colors represent a type of token, and a number of colored tokens represent a value from that color set. Arcs either connect from a place to a transition or from transition to a place (i.e. a map from a set of colors to integer values which describe how many tokens a place can contribute to the threshold of the transition). Arcs from a place to a transition contain a capacity (i.e a map from a set of colors to integer values which describe how many tokens are added to a place when the transition fires). Arcs from a transition to a place contain a weight (i.e. set of colored tokens). Each transition has a minimum threshold requirement (i.e. a map from a set of colors to integer values which describe how many tokens are required for the transition to be enabled). At this point we note that the above is the simplest form of a CPN. One can extend CPN Arcs to have expressions such as relative number of tokens and so on. We have added one such extension, which is a maximum number of tokens for a transition, after which the transition again becomes disabled. Transitions may either be enabled (able to fire) or disabled (not able to fire) depending on if the threshold requirement is satisfied by the number of colored tokens in the originating places and their ability to contribute those tokens by the connected arc capacities. Only one enabled transition is fired at every step of a simulation. If a transition fires, tokens are removed from the originating places according to the capacities of the associated arcs respectively and tokens are placed in connected places according to arc weights connected to those places. Refer to Figure 1 for an illustration of a basic Petri Net (assume 1 token color).

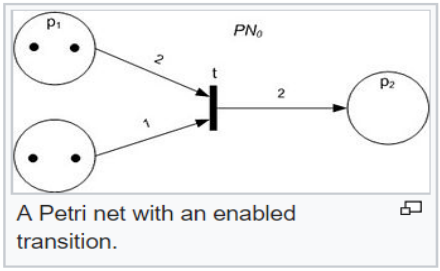

Figure 1: Petri Net [2]

In Figure 1 the Petri Net before firing is shown on top, and the corresponding Petri Net after firing is shown on bottom. The circles represent places, the arrows represent arcs and the line in the center represents a transition. Each dot in a place represents a single token the place currently holds. The two arcs connecting the places to the transition have capacities of 2 and 1 as illustrated by the values placed in the middle of the arc. The arc connecting the transition to the place has a weight of 2 as illustrated by the value placed in the middle of the arc. The transition has a threshold value of the variable t associated with it, however this should be replaced with an integer value in order to simulate it. Since the graphic contains the variable t we will assume that it is replaced with the value 2 for this example to make sense. Since the transition has a threshold has a value of 2 (as we just defined) it is enabled in the top diagram because both originating places have 2 tokens. Thus, we could fire this transition. When fired, 2 tokens would be taken from the top place, as the capacity of that arc is 2, while only 1 token would be taken from the bottom place, as the capacity of that arc is only 1. Subsequently, 2 tokens would be added to the place on the right, as the weight of that arc is 2. Furthermore, in subsequent rounds the transition would not be enabled. While the capacity of the arcs and the threshold does not change, the number of tokens the places have does, which means they are now incapable of contributing tokens to meet the required threshold of 2. The result of the firing can be seen in the bottom diagram. Since the top originating place had 2 tokens and its associated arc had a capacity of 2, 2 tokens were removed from the place. Since the bottom originating place had 2 tokens, but the associated arc only had a capacity of 1, only one token was removed, leaving one token remaining. The weight associated with the arc connecting from the transition to the destination place is 2. Therefore, upon firing 2 tokens are placed in the destination place as shown in the bottom figure.

B. Formal Definitions for Colored Petri Nets

Bulleted below is a formal definition of Colored Petri Net.

- A Colored Petri net is a tuple M=(P, T, A, S, N, E, G, I) [3]
    - P is a set of places (a state with colored tokens) [3]
    - T is a set of transitions (a collection point for activating an action) [3]
    - A is a set of arcs (links places to transitions and vice versa) [3]
    - S is set of color sets defined within CPN model [3]
    - N is a node function (defines what arcs link which places to transitions and vice versa) [3]
    - E defines what the restrictions on arcs are (capacity and weight) [3]
    - G defines the values on Transitions (activation requirements) [3]
    - I is an initial state for the Petri Net [3]

Identify applicable funding agency here. If none, delete this text box.

C. Basic Concurrent Systems Concepts

Concurrent systems provide an advantage by using parallel processing (i.e. multiple tasks being performed concurrently on multiple processors/cores) to decrease computational time. However, with those benefits come risks such as race conditions and deadlocks.

- A Race condition occurs when a process A requires process B to have arrive at state S before process A arrives as state S in order to perform correctly, however process A ends up arriving at state S before process B arrives.
- A Deadlock occurs when process A is in state S1 and needs access to state S2, but state S2 is locked by process B, who needs access to state S1, which is locked by process A.

The above conditions are difficult to reason about, detect, and debug, because they are not guaranteed to be replicable as they depend upon careful timing conditions which may vary between runs and be disrupted by debugging. Therefore, in order to avoid such conditions, a more careful analysis is required and CPNs are can be used for this.

D. Basic WebGME Concepts

WebGME is a Meta-Modeling plugin-based web-framework. It is used to graphically design a meta-model (partially UML2.0 based) for an engineering domain of interest, and then create domain specific models from the meta-model within the same framework. It has a JavaScript and Python plugin-based environment in which a user can write their own plugins. The JavaScript or Python plugins for WebGME may be written to simulate, analyze, and visualize the domain specific models or to interpret the models into a common exchange formats (e.g. XML) to be imported into 3rd party tools. The designed domain-specific modeling environments is referred to as a Design Studio. We utilized WebGME to develop a Colored Petri Net design, simulation, and analysis tool.

III. COLORED PETRI NETS DESIGN STUDIO

A. Design Studio

A Design Studio provides a framework for the modeling and analysis of an engineering domain. [4] The Colored Petri Net Design studio provides a framework for which user can design, simulate, and analyze a concurrent network to reason about deadlocks and race conditions.

First, we present the Colored Petri Net Meta-Model specification within the framework. Second, we present a Domain Specific Model which is created from Meta-Model component instances. Third, we present a series of Python plugins for simulation, analysis, and visualization.

IV. Colored Petri Nets META-MODEL

A. *Meta-Model Specification*

Refer to Figure 2 for the Colored Petri Net Meta-Model designed in WebGME. As seen, everything inherits from the FCO parent. The Colored Petri Net has a one-to-many containment relationship with the abstract Petri Object. This provides a Colored Petri Net Instance with access to places, transitions, and arcs.

The Colored Petri Net meta-node contains several attributes. InitialState, IsDeterministic, Iteration, and StateSpace are meant to act either as outputs or storage for plugins. ColorSet, on the other hand, defines a set of colors which may be used by Petri Objects.

Petri Objects all have one attribute, Tokens, but its meaning changes with the type of Petri Object. For a Place, the Tokens attribute defines how many tokens are currently present at that Place. For a Transition, the Tokens attribute defines how many tokens of each color are required for it to be enabled. For a Place to Transition Arc, the Tokens attribute defines the capacity of the Arc, i.e. how many tokens of a particular color may be contributed by this Place and how many tokens should be removed from the Place. For the Transition to Place Arc, the Tokens attribute defines the weight of the Arc, i.e. how many tokens of a particular color should be added to the Place.

If a Petri Object uses a color that is not in the color set, the TokenSet constraint will flag a violation. Similarly, the NoRepeats constraint will flag a violation if a Petri Object utilizes the same color multiple times.

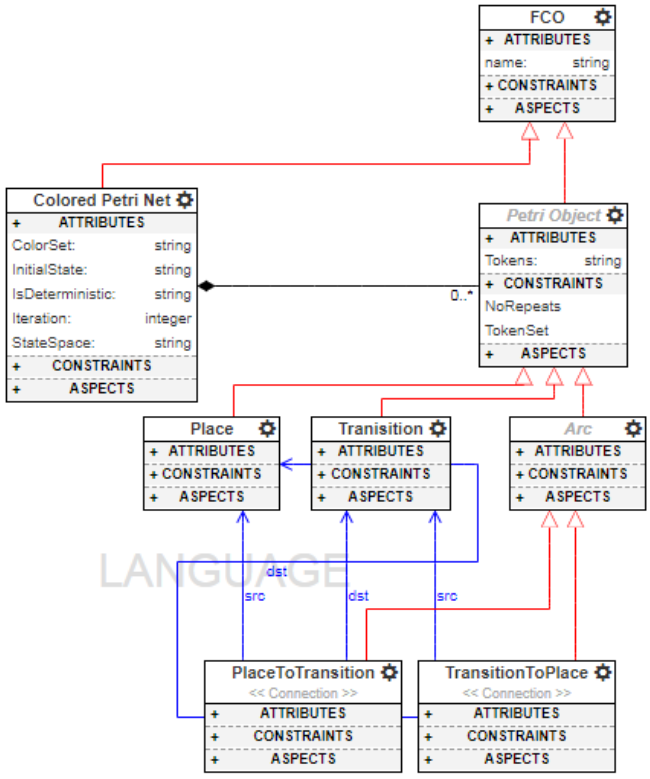

Figure 2: Colored Petri Net Meta-Model

V. Colored Petri Net DOMAIN SPECIFIC MODEL

A. *Domain Specific Model*

Domain specific models (DSMs) of Colored Petri Nets may be designed using the instances of meta-model components. The composition tab within the WebGME framework provides a visual workspace for the creation of such a model. Refer to Figure 3 for a DSM of a Colored Petri Net that was assembled utilizing the composition. As illustrated, colored tokens, thresholds, and weights have been applied to the model. In the current state the top left place has one blue token, the bottom left place has one red token, and each of the other places have no tokens. The top left place has a transition leading out and in. The threshold of the transition leading out has one blue token and the threshold of the transition leading in has 1 blue token and -1 red tokens. The -1 red tokens, means that the transition will be disabled if there is a red token in the Resource place. A similar set of transitions are related to the bottom place. The transition between the resource and deadlock state have a threshold of 1 red token, and one blue token. This demonstrates that the transition to a deadlock place is enabled if both process A & process B access the Resource at the same time.

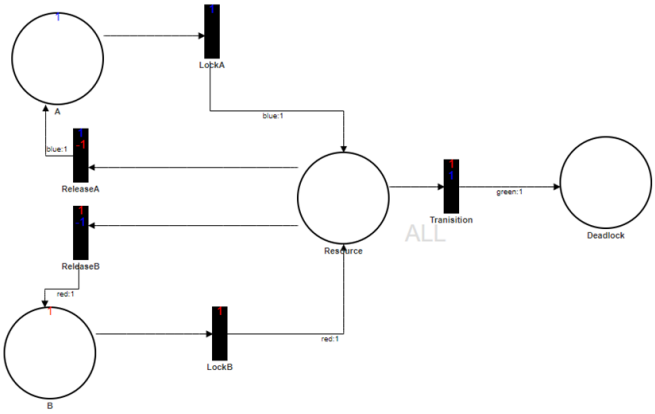

Figure 3: Colored Petri Net DSM

B. *Python Simulation Plugins*

We provide several plugins by which we may simulate a Colored Petri Net or which simplify certain usage aspects.

NextStep: This is a python plugin which, based on the current state of the Net, picks an enabled transition at random from the set of all enabled transitions and fires it. The model is then updated to reflect this.

TotalRun: Given the current state of the model, this plugin randomly picks an enabled transition to fire and records the new state. From the new state, it then repeats this process. This continues until there are no transitions to execute, or until a maximum number of iterations are reached. The maximum can be changed in the model. The final state is saved as the current state of the model. Provided the Naked python module is installed, then the plugin will also create an html file for a visual trace of states.

AllPossible: Given the current state of the model, this plugin fires every enabled transition, and saves the resulting states. For each unique resulting state, this process is repeated. The process continues until no new unique states are generated, there are no transitions to fire, or a maximum number of iterations are reached. The model is not changed, but the trace of all states is recorded. Provided the Naked python module is installed, then the plugin will also create an html file for a visual trace of states.

IsDeterministic: Given the current state of the model, this plugin acts as AllPossible. However, if at any step there is more than one enabled transition, then the process ends early, as the current state is not deterministic. If the process ends as a result of number of iterations, then the model will be updated with the information that it is deterministic so far. If it terminates because there are no additional unique states to consider and no new transitions which can fire, then the model is updated with the information that it is deterministic.

SetInitialState: The model records the current state for future use.

Reset: The model sets the current state to the initial state, which would have been set by SetInitialState.

C. *Visualization*

For visualizing and constructing CPNs, we utilize WebGME's built in composition tool along with its customizable SVG Decorator. As WebGME can use ejs, Embedded JavaScript, we were able to relatively easily define dynamic elements which change their appearance based on certain attributes. In this case, the change is reflected by additional colored text appearing with the relevant object.

In addition to this, as was mentioned above, two of our plugins provide 'trace' artifacts. In order for this functionality to fully work, the Naked module for Python is required. If it is present, the plugin will execute a node script and compile some ejs into an HTML file. If the module is not available, the plugin will instead return an ejs file which can be compiled into the HTML file. The trace provides a minimalistic representation of states, which hopefully allows for a fast visual comparison of consecutive states.

D. *Example*

Figure three show an illustration of a Colored Petri Net DSM designed within the Colored Petri Net Design Studio. This model consists of two processes; A and B. Both process A and process B need access to a shared resource during execution. If during and execution of the Petri net, both process A and process B access the shared resource at the same time the program will enter a deadlock state.

In order to reason about this concurrent system we utilize the Python plugins describe above. We first run the SetInitialState plugin so a record of the token values for the current state are stored in InitialState attribute of the network node. Next we test if the current state is deterministic by running the IsDeterministic plugin. Refer to Figure 4 for the result. As illustrated the results populates the IsDeterministic attribute field with a value of False, because in the current sate both process A and process B are enabled.

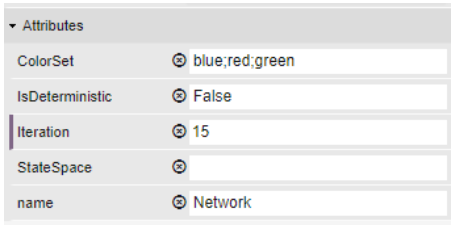

Figure 4: Result of IsDeterminstic Plugin

We then run the NextStep plugin to see which process ends up firing in this non-deterministic state. Refer to Figure 5 for the result. As illustrated process B fired causing one red token to be removed from process B and one red token to be placed in the Resource.

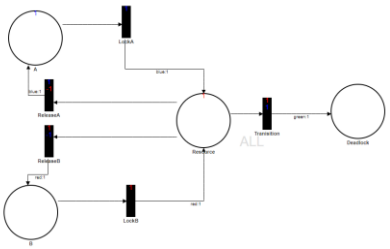

Figure 5: Result of NextStep plugin.

We run IsDeterministic again to test whether the system is deterministic from this point. Refer to figure 5 for the result. As illustrated the results populates the IsDeterministic attribute field with a value of False, because in the current sate both process A and the Resource are enabled.

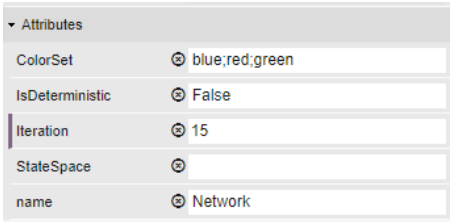

Figure 5: Result of Second IsDeterministic Plugin

We then run the NextStep plugin to see which process ends up firing in this non-deterministic state. Refer to figure 5 for the result. As illustrated the Resource fired, causing one red token to be removed from the Resource, and one red token to be placed in process B. In this case because process A did not fire, the simulation returned to the intial state and no dealock will occur.

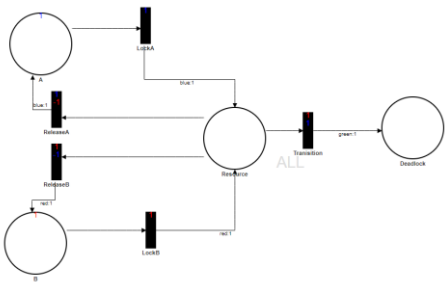

Figure 6: Result of NexStep plugin.

In order to demonstrated the deadlock state being reached we run the TotalRun plugin which runs for the 15 iterations that are specified in the attribute field. Refer to Figure 7 for the results. As illustrated, the Deadlock state was reached, which indicates that some time during the 15 iterations the Resource was accessed by both process A and process B during the same state.

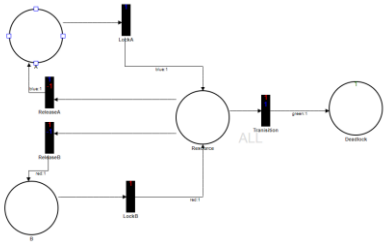

Figure 7: Reults of the TotalRun plugin.

From this state we run the IsDeterministic plugin. Refer to Figure 8 for the result. As illustrated, the results of the IsDeterministic attribute field is populated with a value of True, because in the deadlock state there is no enabled transition.

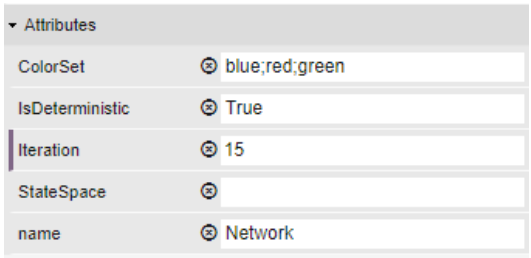

Figure 8: Result of the IsDeterministic Plugin

We can reason about the entire reachabled statespace at a given state by running the AllPossible plugin. First, we run the reset plugin to return to our initial state of as shown in Figure 3. Then we run the AllPossible plugin. Refer to Figure 9 for the results of the run. As illustrated, the results show all reachable states from the initial state.

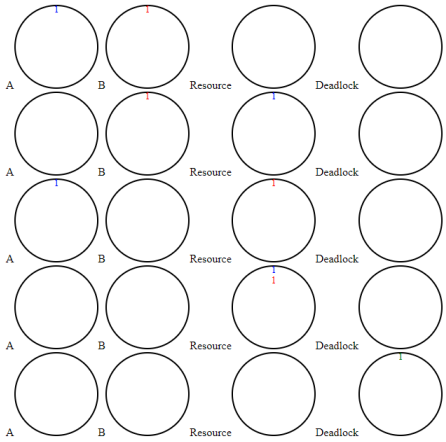

Figure 9: Results of the AllPossible Plugin

VI. Results

We presented a Design Studio for Colored Petri Nets build on the WebGME framework. We described the Meta-Model for Colored Petri Nets as well as a Domain Specific Model example built from the Meta-Model component instances. We demonstrated the Colored Petri Net plugins that are used to simulate, analyze, and visualize the Colored Petri Net models to reason about concurrent systems.

Acknowledgment

We would like to acknowledge T. Kecskes our project sponsor for his imparted knowledge, guidance, lectures on WebGME, and help with development problems. We would like to acknowledge J. Sztipanovits for his class lectures on Meta-Modeling and building sematic domains. We would like to acknowledge P. Meijer for his lectures on WebGME.

[1] techfak.uni, 'Petri Nets', 2018. [Online]. Available: https://www.techfak.uni-bielefeld.de/~mchen/BioPNML/Intro/pnfaq.html. [Accessed: 13-Dec-2018].

[2] en.wikipedia.org, 'Petri Net', 2018. [Online]. Available: https://en.wikipedia.org/wiki/Petri_net. [Accessed: 13-Dec- 2018].

[3] 30th International Conference, PETRI NETS 2009. Paris, France: Springer, 2009.

[4] brightspace.vanderbilt.edu, 'Design Studios', 2018. [Online]. Not Publically Available: https://brightspace.vanderbilt.edu/d2l/le/content/85901/viewContent/867810/View [Accessed: 13-Dec- 2018].