Relationship between Simulink and Petri Nets

Debjyoti Bera[1], Kees van Hee[2] and Henk Nijmeijer[1]

[1]*Dynamics and Control, Department of Mechanical Engineering, TU Eindhoven, Eindhoven, The Netherlands*

[2]*Information Systems, Department of Mathematics and Computer Science, TU Eindhoven, Eindhoven, The Netherlands*

Keywords: Petri Nets, Simulink, Colored Petri Nets, CPN Tools, Discrete Event Systems, Time Driven Systems, Model Checking, Performance Analysis.

Abstract: Matlab/Simulink is an industrial tool that is widely used to design and validate control algorithms for embedded control systems using numerical simulation. A Simulink model of a control system typically defines one or more control algorithms together with their environment. Such models exhibit both discrete and continuous dynamics, simulated by discretizing time. On the other hand, a colored Petri net (CPN) is a well known formalism for modeling behavior of discrete event systems. In this paper, we give a formal semantics to Simulink using the CPN formalism, by describing how Simulink models can be expressed as a CPN. We also show how Petri nets can be simulated in Simulink. Finally, we show how a CPN model can be used for performance analysis of a Simulink model.

1 INTRODUCTION

The use of Matlab/Simulink is one of the de facto standards in the design of embedded control systems. A Simulink model describes a *time driven dynamic system* (Cassandras and Lafortune, 2006) as a set of mathematical equations, evaluated at discrete points in time. In general, a model of a control system consists of a set of controllers and its environment (also referred to as a plant). The goal is to define a mathematical model of a controller, given a model of its environment and a set of requirements that a controller must satisfy. Such models may contain both discrete (difference equations) and continuous parts (differential equations). An exact solution of such a model is in many cases not computable, therefore it is *approximated* by numerical simulation methods, i.e. by discretizing time into time steps, whose minimum is bounded by the resolution of the system. The results obtained from numerical simulation are used as a reference to validate the behavior of both the model and the implemented system (described in lower level languages like C).

An embedded control system is a *discrete event system* (DES) (Cassandras and Lafortune, 2006) whose state evolution depends entirely on the occurrence of discrete events (instantaneous) over time, like ”button press”, ”threshold exceeded” etc. The underlying time-driven dynamics (expressed as difference and/or differential equations) of an embedded control system are captured by the notion of time progression in a state (i.e. the time elapsed between consecutive event occurrences).

On the one hand, the formal specification and verification of a DES is well studied in the context of automata theory (Hopcroft and Ullman, 1979) and Petri nets (Reisig, 1985; Peterson, 1981). Both these formalisms have been extended with the notion of time (Alur and Dill, 1994; Bowden, 2000) (e.g. Timed Automata, time Petri nets) and support model checking of timed properties expressed in temporal logics. On the other hand, Simulink has only informal semantics and of course an operational semantics in the form of an implementation in software. So it is not clear how a Simulink model can be incorporated into a formal framework. Many attempts have been made to address this shortcoming by proposing translations of Simulink models into existing formal frameworks (Agrawal et al., 2004; Denckla and Mosterman, 2005; Tripakis et al., 2005; Tiwari, 2002; Zhou and Kumar, 2012). However, most of these proposals restrict themselves to a subset of Simulink features and do not formally capture the behavior of a simulation run in Simulink. It is only in (Bouissou and Chapoutot, 2012), that a formal operational semantics of Simulink is defined. Unlike other approaches that focus on formalizing the solution method of equations encoded by a Simulink model, the semantics de-

Bera D., van Hee K. and Nijmeijer H..
Relationship between Simulink and Petri Nets.
DOI: 10.5220/0005012000120023
In *Proceedings of the 4th International Conference on Simulation and Modeling Methodologies, Technologies and Applications* (SIMULTECH-2014), pages 12-23
ISBN: 978-989-758-038-3

scribed here captures the behavior of the simulation engine itself, describing the outcome of a numerical simulation.

In this paper, we give a formal semantics to Simulink models using the Petri net formalism. The class of Petri nets present some inherent advantages over automata: (a) Petri nets are graphically intuitive and capture the structural information of a system, (b) A finite automaton can be represented as a Petri net but not vice versa; as a result Petri nets represent a larger class of languages than the class of regular languages, and (c) Structural analysis techniques of Petri nets overcome the drawbacks of state space exploration techniques for analysis of behavior. Furthermore, there are many extensions of time in Petri nets (van Hee and Sidorova, 2013) and their relationship to Timed Automata is well studied (Bera et al., 2013; Cassez and Roux, 2006). We focus on one such extension, namely *Colored Petri nets* (CPN) (Jensen et al., 2007). A CPN is an extension of a Petri net with time and data, and therefore an ideal choice for modeling behavior of Simulink models. The state space reduction technique presented in (Bera et al., 2013; van Hee and Sidorova, 2013) can be used to model check a CPN (after discarding data). Furthermore, the modeling and analysis of a CPN is well supported by the popular CPN Tools. We also show how a Petri net can be expressed as a Simulink model. Furthermore, using a CPN model expressing a Simulink model, we show how existing model checking techniques can be used for performance analysis.

This paper is structured as follows: In the Sec. 2, we present an informal description of Petri nets and CPN. In the Sec. 3, we discuss the underlying concepts of Simulink and show how a Simulink model can be expressed as a CPN model. In the Sec. 4, we show how a Petri net can be expressed as a Simulink model. In the Sec. 5, we show how performance properties of a Simulink model can be verified. In the Sec. 6, we present our conclusions.

2 CONCEPTS

A Petri net (PN) is a bipartite graph consisting of two types of nodes, namely places (represented by a circle) and transitions (represented by a rectangle). We give an example of a Petri net in the Fig. 1. The nodes labeled $P1$ and $P2$ are called places and the node labeled A is called a transition. A place can be connected to a transition and vice-versa by means of directed edges called arcs. The notion of a token gives a Petri net its behavior. Tokens reside in places and often represent either a status, an activity, a resource

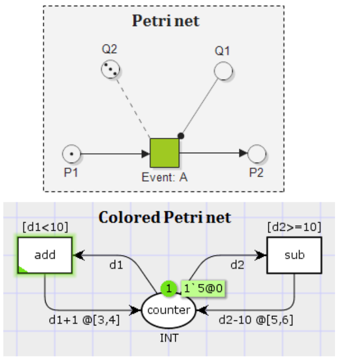

Figure 1: An example of PN and CPN.

or an object. The distribution of tokens in a Petri net is called its *marking* or *state*. Transitions represent events of a system. The occurrence of an event is defined by the notion of *transition enabling and firing*. A transition is enabled if all its input places have at least one token each. When an enabled transition fires it consumes one token from each input place and produces one token in each output place, i.e. changes the state. Apart from arcs between places and transitions there are two other types of arcs, namely inhibitor and reset arcs. An enabled transition having an inhibitor arc (represented as an edge having a rounded tip from transition A to place $Q1$) can fire only if the place associated with the inhibitor arc does not contain consumable tokens, i.e. place $Q1$. A transition connected by a reset arc (represented as a dotted edge from transition A to place $Q2$) to a place, removes all tokens residing in that place (i.e. $Q2$) when it fires.

A colored Petri net (CPN) is an extension of a Petri net with data and time. We give an example of a counter modeled as a CPN in the Fig. 1. Each place has an inscription which determines the set of token colours (data values) that the tokens residing in that place are allowed to have. The set of possible token colours is specified by means of a type called the *colour set* of the place (for eg. the place $counter$ has a color set of type integer denoted by INT). A token in a place is denoted by an inscription of the form $x`y$ (see token 1`5), interpreted as x tokens having token color y, i.e. as a multiset of colored tokens. Like in a standard Petri net, when a transition fires, it removes one token from each of its input places and adds one token to each of its output places. However, the colors of tokens that are removed from input places and added to output places are determined by *arc expressions* (inscriptions next to arcs).

The arc expressions of a CPN are written in the ML programming language and are built from typed variables, constants, operators, and functions. An arc expression evaluates to a token color, which means exactly one token is removed from an input place or added to an output place. The arc expressions on input arcs of a transition together with the tokens residing in the input places determine whether the transition is *color enabled*. For a transition to be color enabled it must be possible to find a binding of the variables appearing on each input arc of the transition such that it evaluates to at least one token color present in the corresponding place. When the transition fires with a given binding, it removes from each input place a colored token to which the corresponding input arc expression evaluates. Firing a transition adds to each output place a token whose color is obtained by evaluating the expression (defined over variables of input arcs) on the corresponding output arc. In our example, the variables $d1$ and $d2$ are bound to the value of the token (value 5) in place *counter*. When one of the transitions, say *add* fires, it produces a token with value $d1+1=6$ in the place *counter*. Furthermore, transitions are also allowed to have a *guard*, which is a Boolean expression. When a guard is present it serves as an additional constraint that must evaluate to true for the transition to be color enabled. In our example, the guard $d1 < 10$ ensures transition *add* can fire if the value of the token in place *counter* is less than 10.

In addition, tokens also have a timestamp that specifies the earliest possible consumption time, i.e. tokens in a place are available or unavailable. The CPN model has a *global clock* representing the current model time. The distribution of tokens over places together with their time stamps is called a *timed marking*. The execution of a timed CPN model is determined by the timed marking and controlled by the global clock, starting with an initial value of zero. In a timed CPN model, a transition is *enabled* if it is both color enabled and the tokens that determine this enabling are available for consumption. The time at which a transition becomes enabled is called its *enabling time*. The *firing time* of a timed marking is the earliest enabling time of all color enabled transitions. When the global clock is equal to the firing time of a timed marking, one of the transitions with an enabling time equal to the firing time of the timed marking is chosen non-deterministically for firing. The global clock is not advanced as long as transitions can fire (eager semantics). When there is no longer an enabled transition at the current model time, the clock is advanced to the earliest model time at which the next transition is enabled (if no such transition then it is a deadlock), i.e. the firing time of the current timed marking. The time stamp of tokens are written after the @ symbol in the following form $x`y@z$, interpreted as x tokens having token color y carry a timestamp z. When an enabled transition fires, the produced tokens are assigned a color and a timestamp by evaluating the time delay interval, inscribed on outgoing arcs of a transition (see time interval $@[a,b]$). The time stamp given to each newly produced token along an output arc is the sum of the value of the current global clock and a value chosen from the delay interval associated with that arc. Note that the firing of a transition is instantaneous, i.e., takes no time.

3 EXPRESSING SIMULINK MODELS USING PETRI NETS

In this section, we describe the basic concepts underlying a Simulink model, namely signals, blocks and system. For these concepts, we give a formal semantics using the CPN formalism. First, we show how a Simulink block can be modeled as a CPN, which we will call a *P-block*. A few simple rules describe how a set of P-blocks can be connected to construct a system, which we will call a *P-system*. The blocks of a Simulink system are executed in a certain order called the *block execution order*. We show how the block execution order of a Simulink system can be modeled as a CPN, on top of an existing P-system. The resulting system describes fully the behavior of a simulation in Simulink and we call it a *C-system*.

3.1 Signals, Blocks and System

Signals and Blocks are the two basic concepts of a Simulink model. A *signal* is a step function in time, representing the behavior of a variable. So only at discrete points in time, the value of a variable may change. Furthermore, a signal in Simulink can have one of the following types: integer, real, complex or its multi-dimensional variants.

A *block* defines a relationship between a set of signals and possibly a variable representing its state (state variable). If a block has a state variable then it is a *stateful block*, otherwise it is a *stateless block*. The set of signals belonging to a block are either of type: *input* or *output*. A block can have zero or more inputs and at most one output and one state variable.

A *stateless block* defines an *output function* $f: I^n \rightarrow O$, where I is the set of inputs, O is the set of outputs and n is the number of inputs of the block. A *stateful block* defines an *output function* $f: I^n \times S \rightarrow O$ and an *update function* $g: I^n \times S \rightarrow S$, where I is the set of inputs, O is the set of outputs, S is the set of state variables and n is the number of inputs of the block. The output function of a stateful block must be evaluated before its update function is evaluated. The result of evaluating the update function of a stateful block (say unit delay block as in the Fig. 2), updates its state

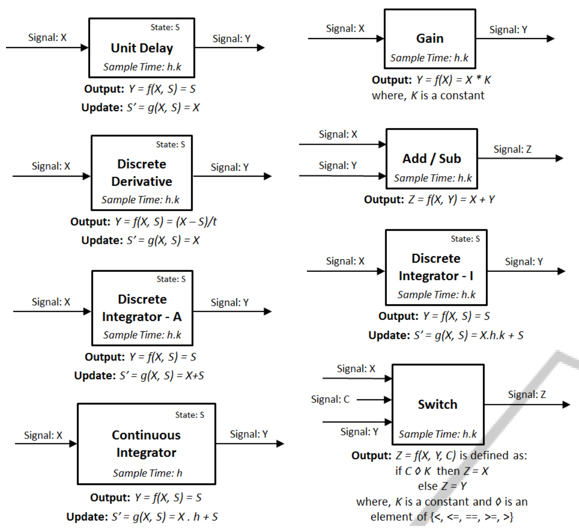

Figure 2: Commonly Used blocks in Simulink.

variable S which is expressed by S', so $S' = g(X,S)$. Furthermore, the *initial condition* of a stateful block is specified by the initial value of its state variable.

A block also has an associated *sample time* that states how often a block should repeat its evaluation procedure. The sample time of a block must be a multiple of the time resolution of a system called the *ground sample time*.

A Simulink model is a directed graph where a node corresponds to a block represented as either a triangle, rectangle or circle and a directed edge between blocks corresponds to a signal. A signal models the data dependency between the output of one block and the input of another block. An output from a block can be connected to input of one or more other blocks. Note that the output of a block can be split into two or more copies which serves as an input to multiple blocks. However, the output signals of two or more blocks cannot join to become one input for another block. We call a network of blocks connected over signals in this manner as a *system*. If all blocks of a system have the same sample time then we call it an *atomic system*. A subset of blocks belonging to a system and having the same sample time is called an *atomic subsystem*.

A mathematical model of a dynamic system is a set of difference and differential equations. Such equations can be modeled as a system in Simulink. A simulation of such a system solves this set of equations using numerical methods. The *state* of a system is defined as the valuation of all its output variables (signals) and state variables. So from an initial state of the system the behavior in the state space is computed in an iterative manner over discrete time steps bounded by the ground sample time. A block having a sample time equal to the ground sample time is called a *continuous block*. All other blocks are *discrete blocks* and have a sample time that is equal to an integer multiple of the ground sample time. For continuous integration, the whole simulation is repeated several times within one ground sample time (depending on the numerical integration method, i.e. solver type) to get a better approximation and detect zero crossing events (Bouissou and Chapoutot, 2012). Note that discrete blocks change their outputs only at integer multiples of the ground sample time which implies that the input to continuous integration blocks remains a constant. However, we neglect the numerical refinement of the ground sample time and consider only one integration step per ground sample time.

The output and update functions of a block can be programmed in Simulink using a low level programming language such as *C*. However, some commonly used constructs are available as predefined configurable blocks in Simulink. In the Fig. 2, we give a few examples of commonly used blocks in Simulink consisting of five stateful blocks (unit delay, discrete derivative, discrete integrator-A, discrete integrator-I, continuous integrator) and three stateless blocks (gain, add/sub and switch). We assume a ground sample time of h time units and $k \in \mathbb{N}$.

The *unit delay* block is a stateful block having one input signal X, a state variable S and one output signal Y. A unit delay block serves as a unit buffer by delaying the current value of its input signal by one sample time, i.e. $h.k$ time units in the following way: The output function f assigns the current value of its state variable to its output Y. The update function g copies the current value of its input signal to its state variable. After every $h.k$ time units, the output function is evaluated and then its update function.

The *gain* block is a stateless block that produces as its output the current value of its input signal multiplied by some constant (specified as a block parameter). The *add/sub* block is a stateless block that produces as its output the sum/difference of the current value of its input signals. The *switch* block is a stateless block that produces as its output either the current value of signal X or signal Y depending on the valuation of the boolean expression defined over signal C.

The *continuous integrator* block is a stateful block that receives as its input (signal X) the derivative of the state variable S (i.e. the rate of change of valuation of the state variable). The output function f assigns the current value of its state variable S to its output Y. The update function g, updates the value of its state variable by integrating the product of the derivative and the ground sample time. The *discrete integrator-I* block is similar to the continuous integrator block. The only difference is that the sample time of the block is an integer multiple of the ground sample time. The *discrete integrator-A* block accumulates the sum of values of its input signal and updates its state variable with this value. The output function of this block copies the current value of its state variable to its output. Note that Simulink blocks such as *transfer function* and *state space* are similar to continuous integrator blocks.

Modeling Simulink Blocks. In the Fig. 3, we show with an example (add block) how a stateless block can be modeled as a P-block. A signal is modeled as a place having one colored token (see places X, Y and Z). The valuation of this colored token is a step function over time, i.e. its value maybe updated by a transition at discrete points in time. A stateless block has two transitions *compute* and *done* connected over a shared place *busy*. The former transition has an incoming arc from each of its places modeling its inputs and the latter is connected with bi-directional arcs to the place modeling its output. The availability of the token in place *enable* concerns the sampling rate. The incoming arc to the place enable, specifies the sample time of the block. In our example, the add block has a sample time of 3, since the token in the place *enable* is delayed by 3 time units, each time the compute transition fires. When a *compute* transition fires, it consumes one token each from its inputs (X and Y) and produces one token in the place *busy* having a color value determined by evaluating its *output function*. For an add block, the output function $f(X,Y) = X + Y$. The execution time of the compute transition (called the *block execution time*) is modeled as a delay along the outgoing arc from the *compute* transition (see delay of 1 time unit along the incoming arc to the place *busy*). Note that it is also possible to specify the block execution time as a time interval. When the token in the place *busy* becomes available, transition *done* consumes this token and produces back one token in each of the inputs of the block (places X and Y with color value unchanged) and updates the color value of the token in its output with the computed result from the output function. This way of modeling a P-block is good for understanding its behavior. However, for a compact representation, the model of a P-block can be simplified by merging together the transitions *compute* and *done* into one transition. In the Fig. 4, we show a simplification of the add block. However, such a model can only be used if the block execution time is specified as a fixed point delay. This is because a time interval specifying a block execution time allows different choices of timestamps for tokens produced in the input and output places of the block. For the remainder of this section, we will consider block execution times as a fixed point delay and use the simplified model of a P-block to present our concepts.

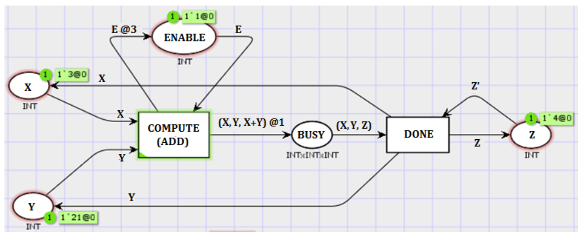

Figure 3: An example of a Stateless Block (P-Block).

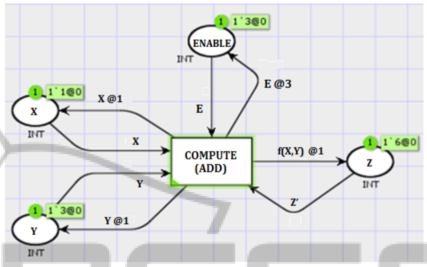

Figure 4: Simplification of the P-Block (Stateless).

The P-block of a *stateful block* has all the concepts of a stateless block, and in addition is extended in the following way: (a) add a special place *state* containing one colored token representing the state of the block and connect it with bi-directional arcs to the block's transition specifying the *output function*, (b) define the *update function* along the outgoing arc to place *state*. In the Fig. 5, we give an example of an unit delay block modeled as a stateful P-block. The update function $g(X,S) = X$ is updating the state variable and the output function $f(X,S) = S$ is updating the value of the token residing in the output place of the block.

3.2 Modeling in Simulink

In the Fig. 6, we present an example of a cruise control system modeled in Simulink. The model is an adaptation of the example in the paper (Bouissou and

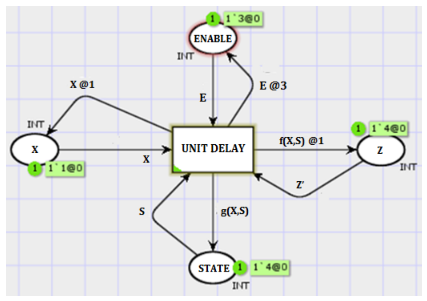

Figure 5: An example of a Stateful Block (P-Block).

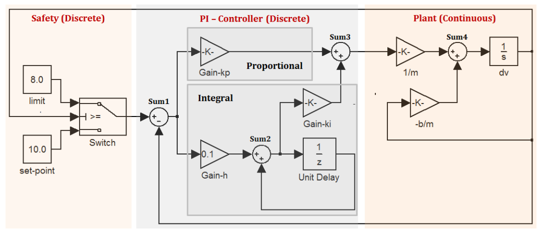

Figure 6: Simulink Model: Cruise Controller.

Chapoutot, 2012). We consider this example because it is simple and covers all relevant modeling aspects of Simulink. The system consists of three main parts: (a) the plant (the device being controlled), (b) the PI (proportional-integral) controller, and (c) the safety mechanism. The plant approximates continuous behavior whereas the other parts are discrete. The safety mechanism ensures that the velocity of the plant does not exceed a specified limit.

The *plant* models continuous behavior of a vehicle whose speed v and position x is given by the equation: $m\dot{v} = -b.v(t) + u(t)$ and $\dot{x} = v(t)$, where m is the mass, b is the friction coefficient, $\dot{x}$ and $\dot{v}$ are the derivatives of x and v w.r.t time t and $u(t)$ is the power of the engine over time. The above equation is implemented in the Simulink model (see Fig. 6) region highlighted as *plant (continuous)* consisting of one sum block, two gain blocks that multiply their input by a factor $1/m$ and $-b/m$ and one continuous integrator block labeled dv. All blocks of the plant have a sample time equal to the ground sample time h of the system.

A standard *PI controller* is modeled with its two parts labeled *proportional* and *integral*. The goal of the PI controller is to produce as output, the power demand $u(t)$ such that the the velocity of the plant $v(t)$ converges to the desired velocity, produced as output of the switch block, as quickly as possible. Note that the *unit delay block* acts as a unit buffer and serves for discrete integration (Euler method). The remaining blocks define the safety mechanism. The switch block monitors the current velocity of the plant. If it exceeds some specified threshold, then it produces as its output, the output of the *limit block*, otherwise the output of the *set-point block* is chosen. All blocks of the PI controller and Safety are discrete and have a sample time of $h.k$ time units, where k is an integer.

Modeling Simulink System. Given a set of P-blocks, we model a P-system by fusing the input places of blocks with the output place of blocks, representing the same signal (i.e. having the same place label). Note that two or more blocks are not allowed to have the same output place because in Simulink, blocks have only one output and this is modeled in P-blocks with a unique output place. To keep the figure readable, we consider only the *integral part* of the PI controller (with $h = 1$ and $k = 10$) and model it as a P-system in the Fig 9.

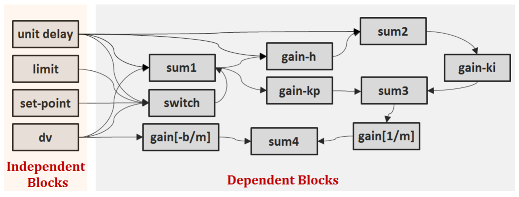

Figure 7: Dependency Graph: Cruise Controller.

3.3 Execution of Simulink Models

The blocks of a Simulink model are executed in a sorted order. To determine this sorted order, we distinguish between two types of blocks, namely independent blocks and dependent blocks. If the output function of a block is defined over only its state variable (i.e. does not depend on its inputs), then we call it an *independent block*, otherwise we call it a *dependent block*. The order in which independent blocks are evaluated is not important. However, the order in which the output of dependent blocks are evaluated is important because the output of a dependent block must be computed only after all other blocks that produce an input to this block have been evaluated. This kind of dependency induces a natural ordering between blocks of a Simulink model which can be represented as a directed graph (blocks as nodes, dependency between blocks as directed edges) representing the order in which blocks of a Simulink model must be evaluated, i.e. a directed edge from block A to block B indicates that block A must be evaluated before block B. We call this graph as the *dependency graph* of a Simulink system. For the example presented in the Fig. 6, the dependency graph between blocks is shown in the Fig. 7. The *block sorted order* is a sorted sequence of blocks whose ordering satisfies the dependency graph of a Simulink system. The sorted order of a Simulink system is determined by the simulation engine before the start of a simulation.

The simulation engine of Simulink executes the contents of a Simulink model (blocks) according to the block sorted order. Each execution of the block sorted order is called a *simulation step*. In each simulation step, every block occurring in the model is visited according to the block sorted order and the following condition is checked for each block: if the time elapsed since this block's last execution equals its sample time, then the block produces its output, otherwise the block execution is skipped, i.e. the block's functions are not reevaluated. The time that is allowed to elapse between each simulation step is a multiple of the ground sampling time called the

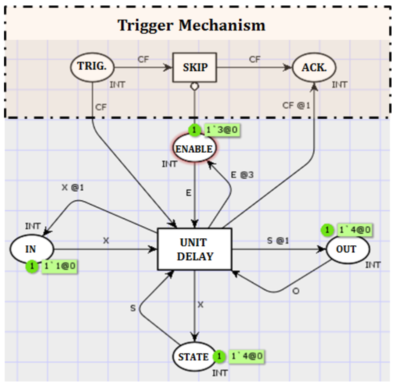

Figure 8: Triggering Mechanism in P-blocks.

step size. The value of the step size for a given Simulink model can be either specified explicitly or it is determined by the simulation engine such that the bounds on approximation errors of integrator blocks are within some specified threshold.

The simulation engine of Simulink is able to execute a model under two different simulation modes, namely *fixed step solver* and *variable step solver*. If a fixed step solver is chosen then a simulation step occurs every *step size* time units. If a variable step solver is chosen then a simulation step occurs at the earliest time instant when at least one block in the model exists such that the time elapsed since its last execution is equal to its sample time. The logic underlying the *variable step solver* mode follows:

Given the *initial conditions* of a Simulink system: initial condition of stateful blocks, current simulation time: $t = 0$, end time: t_f, ground sample time: h, initial step size: $l = h.k$, where k is an integer, block sorted sequence: $\sigma = \langle b_1; \ldots; b_k \rangle$, where $\{b_i \mid i \in 1 \ldots k\}$ is the set of $k \in \mathbb{N}$ blocks, the simulation loop of a variable step is as follows:

- Loop until $t \leq t_f$
  - Evaluate output function of all blocks in the order specified by σ;
  - Evaluate update function of all stateful blocks;
  - Detect zero crossing events (see (Bouissou and Chapoutot, 2012));
  - Update $t = t + l$; Determine the next simulation step size l (an integer multiple of h);

In the fixed step solver mode, the simulation step size is a constant and zero crossing cannot be detected.

Modeling Simulink Control Flow. In the previous section, we have seen how an arbitrary Simulink system can be expressed as a P-system. The enabling of P-blocks in a P-system is determined by their sample time (i.e. by the availability of token in the place *enable*). This means that in each simulation step of the model, multiple P-blocks with the same sample time can become enabled. In CPN semantics, the choice of executing a P-block is done in a non-deterministic manner. However, in a Simulink simulation, the blocks of a Simulink model that can produce their output at a simulation step (determined by their sample times), are evaluated according to their block sorted order.

Consider the P-system of the integral part of the PI controller in the Fig 9. This system has four blocks, namely one sum block (sum2), two gain blocks (gain-h and gain-ki) and one unit delay block. For this system, one of the block sorted order satisfying the dependency graph of the Fig. 7, is the sequence: $\langle$*unit delay; gain-h; sum2; gain-ki*$\rangle$. The block execution order of P-blocks of a P-system is modeled by first adding a *trigger mechanism* to each P-block and then connecting the trigger mechanism of blocks according to the block sorted order. We call the resulting system a *C-system*.

The trigger mechanism is modeled on top of a P-block by adding an input place *trigger* (labeled *trig*), an output place *acknowledge* (labeled *ack*) and a timed inhibitor transition (inhibitor arc that accounts for the availability of token in place *enable*) called *skip*. If the token in place *trigger* is available and the token in place *enable* is unavailable, then the *skip* transition is enabled due to the timed inhibitor arc. Firing the skip transition does not progress time and the execution of the block is skipped in the current time step. If the *skip* transition is disabled (due to an available token in place *enable*), then the block transition (unit delay) fires at its enabling time and produces a token in the place *acknowledge*, delayed by the execution time of the block (see inscription $CF@1$). In both cases, one token is produced in the place *acknowledge*. In the Fig. 8, we give an example of a P-block, extended with a trigger mechanism.

Next, we model the block execution order of a system by introducing a new transition called the *glue transition* between the acknowledge and trigger places of successive blocks as they occur in the sorted order. The construction is carried out in the following way: For each block occurring in the block execution order, we add one glue transition that consumes a token from the place *acknowledge* of the preceding block and produces a token in the place *trigger* of this block. In this way, the C-System describes a simulation run of the system. In order to allow for more than one run of a simulation at a rate specified by the step size: (a) we add a transition labeled *closure* that consumes a token from the place *acknowledge* belonging to the last block in the sorted order and produces a token in the place *trigger* corresponding the first block

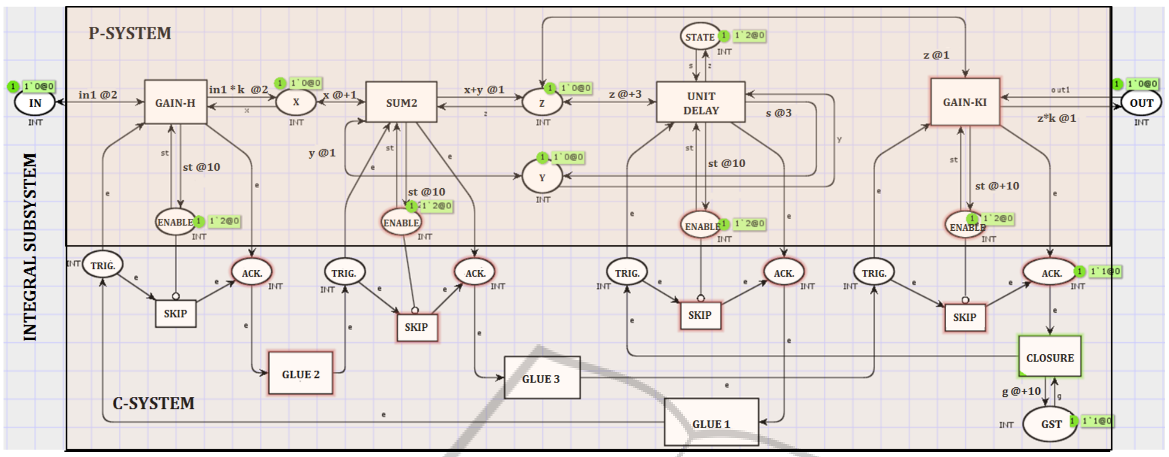

Figure 9: C-System: Integral Part (PI Controller).

in the sorted order, and (b) we initialize the place *acknowledge* of the last block occurring in the block sorted order with a token having a timestamp zero. Furthermore, to this closure transition, we connect a place called *GST* with bidirectional arcs and having one token. On the incoming arc to the place *GST*, we associate a delay corresponding the *step size* of the simulation (multiple of ground sample time). As a result, a simulation run can only occur once every *step size* time units. If the model has continuous blocks then the step size must equal the ground sample time. To simulate a variable step solver, the step size must be equal to the least sample time of all blocks in the system. In the Fig. 9, we describe the C-system of the integral part of the PI controller having a step size of 10 time units.

4 EXPRESSING PETRI NETS USING SIMULINK

In this section, we will express the semantics of a Petri net as a Simulink system by modeling a place and a transition as an atomic subsystem. The underlying strategy is simple: The place subsystem keeps track of the number of tokens and informs all successor transitions about this value. When a transition subsystem either consumes/produces a token, it indicates the predecessor/successor place subsystem about the occurrence of this event.

In the Fig. 10, we give an example of a Petri net consisting of one place P, and two transitions $t1$ (pre-transition) and $t2$ (post-transition), and show how the place P of a Petri net can be modeled as an atomic subsystem (place subsystem). The place subsystem has two input signals (signals: a and b) and one output signal (b'). The signals a and b are assumed to be updated by the pre and post transitions $t1$ and $t2$, respectively and the signals can either have a value of 0 or 1. A change in the value of either of these signals indicates that a transition has altered the number of tokens in the place. The output signal b' is an input signal to transition $t2$ and has a data type *integer*, whose value represents the number of tokens in a place subsystem. We call signals a and b as *indication signals* and signal b' as a *token signal*.

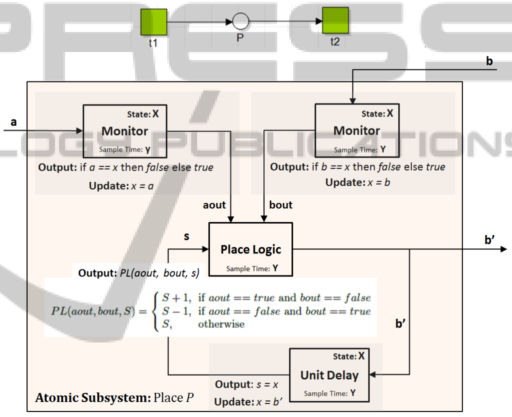
t1
P
t2
a
b
State: X
Monitor
Sample Time: Y
Output: if a == x then false else true
Update: x = a
Output: if b == x then false else true
Update: x = b
aout
bout
s
Place Logic
Output: PL(aout, bout, s)
b'
$PL(aout, bout, S) = \begin{cases} S+1, & \text{if } aout == true \text{ and } bout == false \\ S-1, & \text{if } aout == false \text{ and } bout == true \\ S, & \text{otherwise} \end{cases}$
Unit Delay
Output: s = x
Update: x = b'
Atomic Subsystem: Place P

Figure 10: Modeling a Place in Simulink.

The place subsystem has three stateful blocks (two monitor blocks and a unit delay block) and one stateless block (place logic). All the four blocks have the same sample time equal to y time units. The input monitor block receives an indication signal from a pre-transition and the output monitor block receives an indication signal from a post-transition. The two *monitor* blocks compare the current value of their input signal with the value recorded (i.e. state: x) from the previous time step. If they are equal then the block produces an output: *false*, otherwise it produces an output *true*. The state of an *unit delay* block corre-

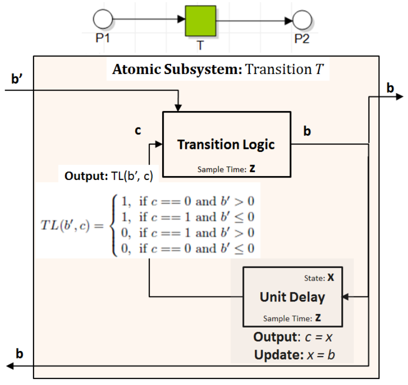
P1
T
P2
Atomic Subsystem: Transition T
b′
b
c
Transition Logic
Sample Time: Z
Output: TL(b′, c)
$TL(b',c) = \begin{cases} 1, & \text{if } c == 0 \text{ and } b' > 0 \\ 1, & \text{if } c == 1 \text{ and } b' \le 0 \\ 0, & \text{if } c == 1 \text{ and } b' > 0 \\ 0, & \text{if } c == 0 \text{ and } b' \le 0 \end{cases}$
State: X
Unit Delay
Sample Time: Z
Output: c = x
Update: x = b
b

Figure 11: Modeling a Transition in Simulink.

sponds the number of tokens in the place in the current time step. At every time step, the unit delay block produces as its output, the current value of its state variable. The initial state of the unit delay block corresponds the initial number of tokens in the place. The *place logic* block defines a function to compute the number of tokens in the current time step depending on the output of monitor blocks *aout* and *bout* and the output of the unit delay block S. If either *aout* or *bout* is true, then one of the transitions must have changed the number of tokens in the place and the value of signal b' must be updated (i.e. either increased or decreased by one), otherwise b' remains unchanged.

If a place subsystem has more than one pre-transition, then we add for each pre-transition, one indication signal (input) and a corresponding input monitor block whose output is connected to the place logic block. The logic underlying the place logic block is extended to handle an additional input condition. If a place subsystem has more than one post-transition, then we add for each post-transition, one indication signal (input) and a corresponding output monitor block whose output is connected to the place logic block. Additionally, the token signal (output) is split as an input for each post transition.

In the Fig. 11, we give an example of a Petri net consisting of one transition T and two places $p1$ (pre-place) and $p2$ (post-place), and show how transition T of a Petri net can be modeled as an atomic subsystem (transition subsystem). The transition subsystem has one input signal b' (token signal) as an output from pre-place $p1$ and one output signal b (indication signal) that is split as an input to both its pre-place $p1$ and post-place $p2$. The current value of signal b' is representing the number of tokens in the transition's pre-place $p1$. The signal b has either value 0 or 1.

The transition subsystem has two blocks having the same sample time of z time units. The unit delay block has a state either 0 or 1 and in each time step, produces as its output the state of the block in the previous time step. The output of the unit delay block is connected to the transition logic block. The *transition logic* block defines a function that produces as its output a value of 0 or 1 depending on the value of the token signal b' (from a pre-place) and the output signal of the unit delay block. If the value of signal b' is greater than zero, then it means there are enough tokens and transition T can fire. The firing of a transition is indicated by a change in the value of the indication signal b such that it has a value that is not equal to the output of the unit delay block.

If a transition subsystem has more than one pre-place, then we add for each pre-place: one token signal as an input to the transition logic block, and split the indication signal (output) as an input to the pre-place subsystem. The logic of the transition logic block is extended to change its value if and only if all of its inputs have a value greater than zero. If a transition subsystem has more than one post-place, then we split for each post-place, the indication signal (output) as an input to the post-place subsystem.

An equivalent model for a network of places and transitions can be obtained by connecting place subsystems to transition subsystems over their shared signals. In such a network, the place subsystems must produce their outputs faster than transition subsystems. If more than one transition shares the same pre-place then they must have different sample times.

5 ANALYSIS

In this section, we discuss how timed properties of a C-system can be verified by model checking techniques using a simple example. In the Fig. 12, we present an atomic subsystem modeling Euler integration with two blocks: unit delay and sum, both having a sample time equal to 10 time units and a simulation step size of 15 time units. The execution time of the unit delay block lies in the interval of $[1,3]$ time units and the execution time of the sum block is 2 time units. For analysis purposes, we modify the C-system with one additional transition *complete* and one place *init* (initialized with a token) with no delays between them. If we consider the notion of colored tokens in a state, then analysis is possible only for a finite future (partial state space). If we drop the notion of color from a CPN, then we obtain a subclass of *Discrete Timed Petri nets* (DTPN) (Bera et al., 2013; van Hee and Sidorova, 2013). So for analysis of a C-system, we will ignore color and discard places modeling signals and states. As the state space of a DTPN is infinite because (a) time intervals on outgoing arcs leads to an infinite choice of timestamps for

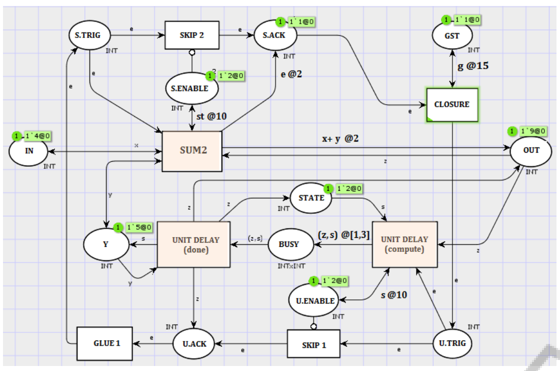

Figure 12: C-System: Euler Integration.

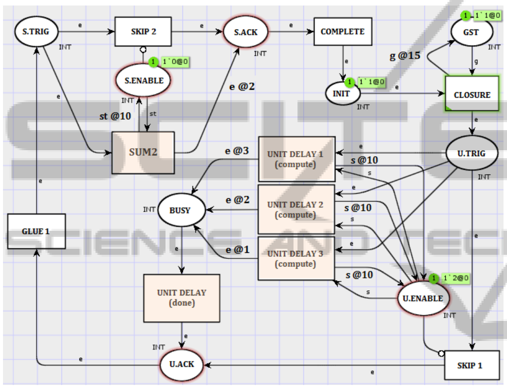

Figure 13: sDTPN of C-System.

newly produced tokens, and (b) time progression is non-decreasing; model checking is not directly possible. For this a *reduction method* is proposed that reduces the state space of a given DTPN into a finite one. The reduction method progresses in two stages: interval and time reduction.

The *interval reduction* step, replaces each time interval on outgoing arcs of a DTPN with a set of finite values in the following way: For each time interval specified along an outgoing arc, construct a set containing its lower bound, upper bound and a finite number of equidistant points between the two bounds such that the points are separated from each other by a so called *grid distance*. The grid distance is the least common multiple of the denominators of all non-zero elements of the set consisting of all token timestamps in the initial marking, and the lower and the upper bounds of all time intervals expressed as non-reducible elements of the set of rational numbers. The resulting net is called an fDTPN (preserves simulation equivalence). An fDTPN can be transformed into a DTPN with outgoing arcs having only singleton delays by making copies of transitions and associating each of their outgoing arcs with a unique element from the finite set of delays associated with that arc. The resulting net is called an sDTPN (preserves strong bisimulation).

In our example, the C-System (DTPN) has a grid distance equal to 1, so the time interval $[1,3]$ associated with the outgoing arc from the unit delay transition is replaced by the finite set $S = \{1,2,3\}$ and the delay of 2 time units on the outgoing arc from the sum transition is replaced by the singleton set $\{2\}$. The resulting system is an fDTPN. Next, the choice of delays represented by the set S on an outgoing arc from one unit delay transition is simulated by three copies of the unit delay transition (same preset and postset), each having a unique delay from the set S specified along its outgoing arc. The resulting sDTPN is shown in the Fig. 13. Note that transitions *unit delay 1*, *unit delay 2* and *unit delay 3* simulate the behavior of the transition *unit delay* (compute).

The *time reduction* step, transforms a given sDTPN into a finite transition system (rDTPN) by reducing each timed marking of an sDTPN (concrete marking) into a so called *abstract marking*. An abstract marking represents a class of concrete markings by aggregating them based on relative distances between timestamp of tokens. An abstract marking of a concrete marking is obtained by subtracting the timestamps of all tokens in the marking by its firing time. This means the *firing time* of an abstract marking is zero. It turns out that for any bounded sDTPN there exists a finite transition system (of abstract markings), called an rDTPN (preserves strong bisimulation).

As all places have at most one token in any reachable marking, we identify a token in a place by the name of that place and define a marking as a function that assigns to each token identity, a timestamp. A marking is represented as a vector of timestamps corresponding the ordered set of places: [*S.ENABLE; U.ENABLE; U.TRIG; BUSY; U.ACK; S.TRIG; S.ACK; INIT; GST*]. In the Fig. 14, we present the rDTPN of our sDTPN. Consider the reachable concrete markings $[11,10,-,-,-,-,-,3,15]$ and $[26,25,-,-,-,-,-,18,30]$ from the initial marking of the sDTPN in the Fig. 13. From both these markings, the earliest enabled transition is the *closure transition* at times 15 and 30, respectively. So their abstract marking is $L14 : [(-4),(-5),-,-,-,-,-,(-12),0]$.

Timed Analysis. To analyse a C-system, we must be able to compute the upper and lower bounds on the time required to reach a marking m from its initial marking m_0, i.e. timed reachability.

First, we show how to compute the time taken to execute a sequence of transitions $\sigma = \langle t_0;\dots;t_n\rangle$, starting from the initial marking m_0 and leading to marking m. Consider our sDTPN (see Fig. 13) and its rDTPN (see Fig. 14). As all token timestamps are

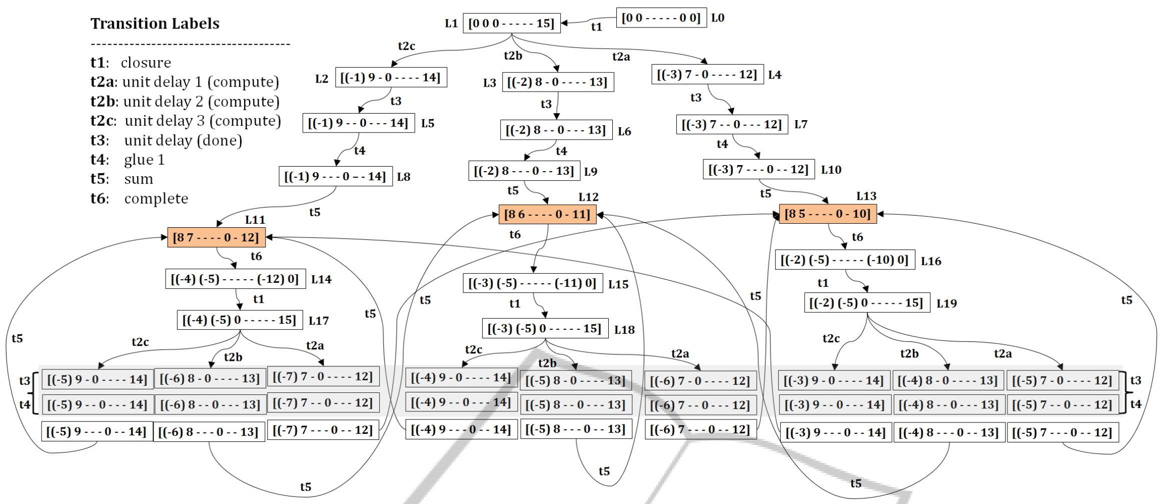

Figure 14: rDTPN.

zero in the initial marking m_0, the initial marking l_0 of the rDTPN is an abstract marking of m_0. Due to bisimulation equivalence, the same sequence σ exists from initial marking l_0 of the rDTPN, leading to a marking l such that it is an abstract marking of m.

Then for each transition $t \in \sigma$, leading from an abstract marking l' to the abstract marking l'', compute the *relative delay* of marking l'' in the following way: Consider the marking l' as a concrete marking of the sDTPN. From this marking, fire the enabled transition t, leading to a marking, say $\tilde{m}$. The *firing time* of marking $\tilde{m}$ (i.e. the earliest enabled transition) is the *relative delay* of abstract marking l''. Note that the relative delay of initial marking l_0 is always zero.

As an example, consider the marking $L2$ reachable from marking $L1$ over transition $t2a$. Consider the marking $L1$ as a concrete marking of sDTPN. Firing transition $t2a$ from this marking leads to marking $[0, 10, -, 1, -, -, -, -, 10]$ (in the sDTPN) with a firing time 1. So the relative delay of $L2$ is 1.

The time taken to execute σ is the sum of all relative delays of all abstract markings visited by the transitions of this sequence. As an example, consider the marking $L12$ reachable from $L0$ by the sequence of transitions $\langle t1; t2b; t3; t4; t5 \rangle$. All the abstract markings visited by this sequence, except for $L3$ and $L12$ have a non-zero relative delay of 2 time units each. So the time taken to execute this sequence is 4 units.

Next, the lower and upper bounds on the time required to reach marking m from marking m_0 in the C-system corresponds the minimum and maximum of execution times of all possible transition sequences starting from marking l_0 and leading to marking l, respectively. A desirable property of a well-defined C-system is formulated as: *the upper bound of a run of the block execution order must never be greater than the simulation step size.* As an example, consider the set of shaded abstract markings called *home markings* in our rDTPN. These markings correspond the completion of a block execution order (token in the place acknowledge of the sum block). So every path starting from an initial marking or a home marking and leading back to a home marking represents a run of the block execution order. For this, the upper and lower bounds of the time can be computed by the techniques described so far. For our example, this turns out to be in the interval of $[3, 5]$ time units. Note that we have chosen a trivial example to present our concepts. For a multirate system, the analysis and the type of questions become more interesting.

An rDTPN can also be used to validate the timed specification of a Simulink model. Note that the unit delay and sum blocks have the same sample time. From its rDTPN, we verify that in every path between home markings, one of the unit delay and sum transition always occurs. If the simulation step size is 10 time units then there exist paths between home markings where the skip transition of the sum block occurs.

Other interesting features of Simulink like decisional constructs (if-then-else and switch) and loop constructs (for and while) can also be verified using the above techniques. These constructs can be modeled in a C-system as a non-deterministic choice between triggering mechanisms of blocks (see Fig. 16).

In many Simulink systems, messages may have to be exchanged over one or more communication channels (supported by UDP/TCP send-receive blocks). Such a message passing mechanism is easily modeled as a Petri net, which can in turn be expressed as a Simulink system as a network of place and tran-

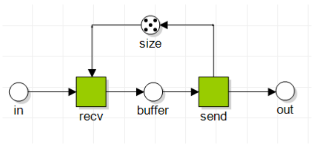

Figure 15: Petri net model of a Communication Channel.

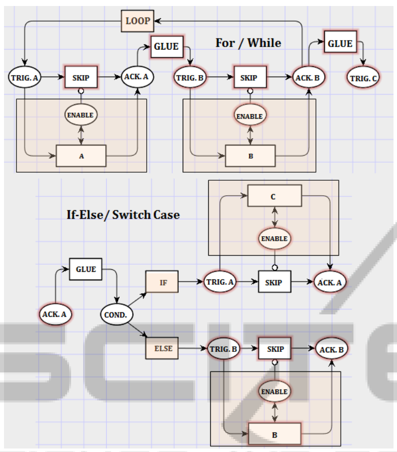

Figure 16: Modeling Choices in C-System.

sition subsystems. In the Fig. 15, we give an example of a Petri net modeling a communication channel with buffer size 5. For a C-system with a communication channel, interesting properties like buffer utilization and system performance in the presence of communication delays may be analyzed.

6 CONCLUSIONS

Simulink is a graphical way of modeling a set of difference and differential equations whose result is simulated by discretizing time. We have shown how the simulation behavior of Simulink models can be expressed using the formalism of CPN. As models of this formalism can be model checked, the safety and performance properties of a Simulink model can be guaranteed. We also showed how Petri nets can be simulated in Simulink. So the two formalisms are formally equivalent in their expressive power. However, the modeling comfort is different. Petri nets are a better choice for modeling discrete events like message passing, whereas Simulink has more built-in facilities for numerical approximation of differential equations, than existing tools for Petri nets. Furthermore, the formal study of DES is deeply rooted in the theory of Petri nets and automata theory. So the specification of both discrete and continuous parts of a Simulink model using the same formalism, gives a better understanding of their behavior within the context of DES.

REFERENCES

Agrawal, A., Simon, G., and Karsai, G. (2004). Semantic Translation of Simulink/Stateflow models to Hybrid Automata using Graph Transformations. In *International Workshop on Graph Transformation and Visual Modeling Techniques*, page 2004.

Alur, R. and Dill, D. L. (1994). A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235.

Bera, D., van Hee, K., and Sidorova, N. (2013). Discrete Timed Petri nets. Computer Science Report 13-03, Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands.

Bouissou, O. and Chapoutot, A. (2012). An Operational Semantics for Simulink's Simulation Engine. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 129–138, New York, NY, USA. ACM.

Bowden, F. D. (2000). A brief survey and synthesis of the roles of time in Petri nets. *Mathematical and Computer Modelling*, 31(10):55–68.

Cassandras, C. G. and Lafortune, S. (2006). *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Cassez, F. and Roux, O. H. (2006). Structural translation from Time Petri Nets to Timed Automata. *Journal of Systems and Software*, 79(10):1456–1468.

Denckla, B. and Mosterman, P. J. (2005). Formalizing Causal Block Diagrams for Modeling a Class of Hybrid Dynamic Systems. In *In IEEE CDC-ECC 05*.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company.

Jensen, K., Kristensen, L. M., and Wells, L. (2007). Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *Int. J. Softw. Tools Technol. Transf.*, 9(3):213–254.

Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Reisig, W. (1985). *Petri nets: An Introduction*. Springer-Verlag New York, Inc.

Tiwari, A. (2002). Formal Semantics and Analysis Methods for Simulink Stateflow Models. Technical Report, SRI International.

Tripakis, S., Sofronis, C., Caspi, P., and Curic, A. (2005). Translating Discrete-time Simulink to Lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4):779–818.

van Hee, K. and Sidorova, N. (2013). The Right Timing: Reflections on the Modeling and Analysis of Time. In *Proceedings of the 34th International Conference on Application and Theory of Petri Nets and Concurrency*, PETRI NETS'13, pages 1–20, Berlin, Heidelberg. Springer-Verlag.

Zhou, C. and Kumar, R. (2012). Semantic Translation of Simulink Diagrams to Input/Output Extended Finite Automata. *Disc. Event Dyn. Sys.*, 22(2):223–247.