Modeling Hybrid Systems with Petri Nets

Debjyoti Bera, Kees van Hee and Henk Nijmeijer

Abstract The behavior of a hybrid system is a mixture of continuous behavior and discrete event behavior. The Simulink/Stateflow toolset is a widely used industrial tool to design and validate hybrid control systems using numerical simulation methods for the continuous parts and an executable Stateflow (combination of Statecharts and Flowcharts) for the discrete event parts. On the other hand, Colored Petri Nets (CPN) is a well-known formalism for modeling behavior of discrete event systems. In this paper, we show how the CPN formalism can be used to model a hybrid system. Then we consider the special case of Simulink/Stateflow models and show how they can be expressed in CPN.

Keywords Petri nets · Simulink · Colored Petri nets · CPN tools · Discrete event systems · Time-driven systems · Model checking · Performance analysis

1 Introduction

A hybrid system can be distinguished into two types of subsystems, a time-driven subsystem and a discrete event subsystem. The former defines one or more control algorithms together with their environment, while the latter defines discrete event logic in terms of an executable state machine.

A *time-driven subsystem* is usually defined as a set of mathematical equations, evaluated at discrete points in time. In general, such a subsystem models a set of controllers and its environment (also referred to as a plant). The goal is to define a

D. Bera (✉) · K. van Hee · H. Nijmeijer
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: d.bera@tue.nl

K. van Hee
e-mail: k.m.v.hee@tue.nl

H. Nijmeijer
e-mail: h.nijmeijer@tue.nl

M.S. Obaidat et al. (eds.), *Simulation and Modeling Methodologies, Technologies and Applications*, Advances in Intelligent Systems and Computing 402, DOI 10.1007/978-3-319-26470-7_2

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). A *discrete event subsystem* is usually expressed as a finite state machine, whose state evolution depends entirely on the occurrence of discrete events (instantaneous) over time, like "button press," "threshold exceeded," etc. So the underlying time-driven dynamics (expressed as difference and/or differential equations) of a hybrid system are captured by the notion of time progression in a state (i.e., the time elapsed between consecutive event occurrences).

The Simulink/Stateflow toolset is an industrial tool that is widely used to design and validate hybrid control systems using numerical simulation methods for the time-driven parts and an executable Stateflow specification (combination of Statecharts [1] and Flowcharts) for simulating the discrete event parts. The Simulink toolset has only an informal semantics (c.f. [2, 3]) and of course an operational semantics in the form of an implementation in software. Since these models are eventually implemented as software and often integrated into larger systems, it becomes necessary to understand their logical structure and verify them in an exhaustive manner. The problem is even more serious for Stateflow because it has numerous complex semantics explained in a very verbose manner [3]. Furthermore, the semantics of Stateflow deviates from that of Statecharts, as a result, the many existing results on the formalization of Statecharts (c.f. [4–6]) are not directly applicable. As a result, runtime failures of Stateflow models are quite common in practice and these problems are usually hard to understand and analyze by designers. As a consequence several guidelines have been proposed by the industry restricting the language to safe subset and prescribing design patterns. However, these guidelines have no formal basis and rely solely on lessons learned from experience. So it becomes necessary to understand these models in a formal manner.

Many attempts have been made to address this shortcoming by proposing translations of Simulink models into existing formal frameworks such as automata and its various extensions [7–11]. It is only in [12] 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 described here captures the behavior of the simulation engine itself, describing the outcome of a numerical simulation. Similar attempts have been made for Stateflow as well [7, 13–15].

In this paper, we present a method to model hybrid systems with the formalism and tools of Colored Petri Nets (CPN) [16]. Although there are several extensions of a Petri net for modeling hybrid systems such as hybrid Petri nets [17], differential Petri nets [18], etc., and their applications (for instance in bio-chemical networks [19]), there are several good reasons to apply the CPN formalism in this context such as (a) CPN has a simple and well-defined semantics, and (b) CPN as an extension of classical Petri nets has many analysis methods [20, 21] for verification of models. There are also specific analysis methods for Colored Petri Nets. Nonexperts often say that Petri nets are a very primitive modeling framework. This might be true for classical Petri nets although they generalize finite state machine models with concurrency. Classical Petri nets treat *states* and *events* in a symmetrical way, which is a big advantage. On the other hand, CPNs extend tokens with arbitrary complex data types and transitions have arbitrary complex functions to compute values of output tokens using input tokens. So they have a strong modeling power (of course Turing complete) and are supported by good tools for modeling and analysis (c.f. [22]). We also consider the special case of the Simulink/Stateflow toolset and express them as a CPN. The main difference between Simulink/Stateflow and CPN is that the former is developed as a tool and only afterwards scientists have tried to define formal semantics for it, while CPN were developed as a theory for concurrent systems in the 1960s and much later, in the 1990s, its supporting tools were developed. Although CPN was originally meant for modeling concurrent discrete event systems, they are very suitable for modeling continuous systems and hybrid systems as well. The advantage of Simulink/Stateflow is that they provide handy notations for often recurring modeling issues. Therefore we also introduce similar notations for CPN. Note that these notations are only syntactic sugar, i.e., their semantics are derived from the translation back to the basic syntax of CPN.

This paper is structured as follows: In Sect. 2, we introduce the Petri net and Simulink frameworks in an informal way. In Sect. 3, we give a compact notation for CPN and call it CPN*. In Sect 4, we show how models of hybrid systems can be expressed in CPN*. Here we also consider Simulink models as a special case. In Sect. 5, we study the relationship between Stateflow models and CPN* models and show how the latter can be derived from the former. In Sect. 6, we present our conclusions.

2 Overview of the Frameworks

In this section, we introduce the four modeling frameworks of interest in this paper, namely Petri nets, its colored and timed extension colored Petri net (CPN), the Simulink framework for modeling continuous and discrete timed behavior, and its Stateflow component for modeling discrete event systems.

2.1 Petri Nets

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 Fig. 1. The nodes labeled $P1$ and $P2$ are called places and

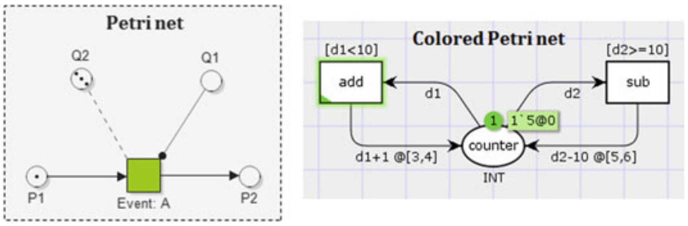

Fig. 1 An example of PN and CPN

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. A *preset* (*postset*) of a node in a Petri net is the set of all nodes having an incoming (outgoing) arc to (from) it. 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, or an object. The distribution of tokens in a Petri net is called its *marking* or *state*. We call the initial distribution of the tokens in a net as its *initial marking*. A Petri net having an initial marking is called a *net system*. 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. We lift the notion of transition enabling and firing to sequence of transitions. A given marking is said to be *reachable* by a net system if there exists an executable sequence of transitions leading to it. The set of all markings that are reachable by a net system is called the *set of reachable markings*. 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 *Q*1) can fire only if the place associated with the inhibitor arc does not contain consumable tokens, i.e., place *Q*1. A transition connected by a reset arc (represented as a dotted edge from transition *A* to place *Q*2) to a place, removes all tokens residing in that place (i.e., *Q*2) when it fires.

A Petri net is called a *state machine net* if its transitions have a preset and postset of size equal to one. A *subnet* of a given Petri net is a subset of its places and transitions and the connections between them. A place of a Petri net is *safe* if in all reachable markings of its net system, the place contains at most one token. A net system with safe places is called a *safe net*.

2.2 Colored Petri Nets

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 Fig. 1. Each place has an inscription which determines the set of token colors (data values) that the tokens residing in that place are allowed to have. The set of possible token colors is specified by means of a type called the *color set* of the place (for e.g., 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 functional 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 timestamps 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 nondeterministically 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 timestamp 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. If the place is safe, then we do not mention the number of tokens. 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 $@[a, b]$ or just $[a, b]$). The timestamp 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. Since the time domain is infinite and there is an infinite number of choices that can be made in a time interval, such a CPN model has an infinite state space. As a result, it is not possible to model check them directly. To be able to analyze such models, we need reduction methods as proposed in [23, 24].

2.3 Simulink

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 data types: integer, real, complex, or its multidimensional variants.

A *block* defines a functional 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 *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* $f : I_1 \times \ldots \times I_n \rightarrow O$, where $n \in \mathbb{N}$ is the number of inputs of the block, I_j is the type of input signal, for $j \in \{1, \ldots, n\}$, and O is the type of the output signal. A *stateful block* defines an *output function* $f : I_1 \times \ldots \times I_n \times S \rightarrow O$ and an *update function* $g : I_1 \times \ldots \times I_n \times S \rightarrow S$, where $n \in \mathbb{N}$ is the number of inputs of the block, I_j is the type of input signal, for $j \in \{1, \ldots, n\}$, S is the type of the state variable and O is the type of the output signal. 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 (see unit delay block as in Fig. 2) updates its state 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 by splitting it into two or more copies, which serves as an input to multiple blocks. However, the output signals of two or

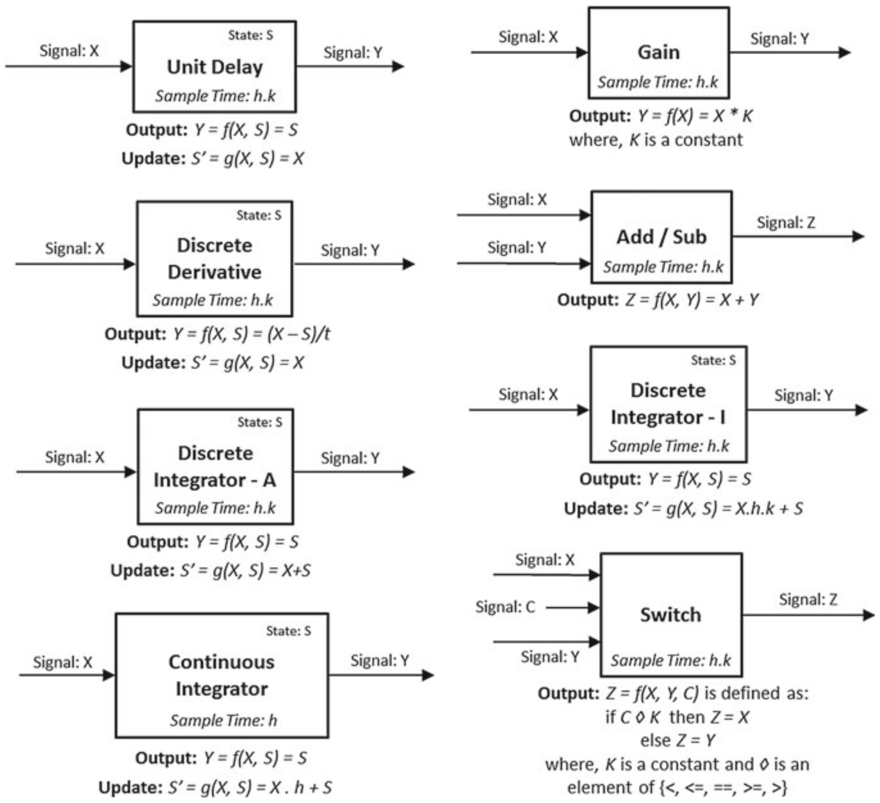

Fig. 2 Commonly used blocks in simulink

more blocks cannot join to become one input for another block. A network of blocks connected over signals in this manner is called a *system*. If all blocks of a system have the same sample time then the system is called 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 [12]. 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, without loss of generality, 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 Fig. 2, we give a few examples of commonly used blocks in Simulink along with their function definitions. We consider 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 will discuss two of them.

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 *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. Note that Simulink blocks such as *transfer function* and *state space* are similar to continuous integrator blocks.
Simulink System. A *simulink system* is a set of Simulink blocks connected over shared input and output signals. In Fig. 3, we present an example of a cruise control system modeled in Simulink. The model is an adaptation of the example in the paper [12]. We consider this example because it is simple and covers all relevant modeling aspects of Simulink. For a detailed description of mathematical equations underlying this model, please refer to [12].

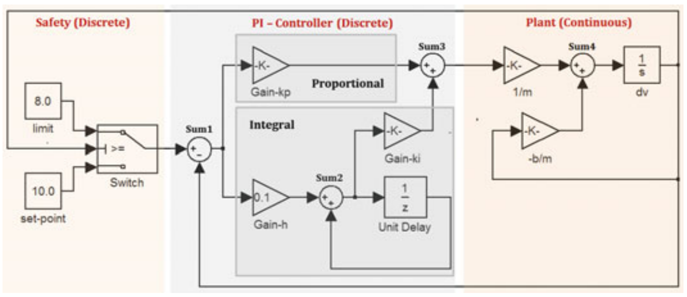

Fig. 3 Simulink model: Cruise controller

Simulation of Simulink System. The blocks of a Simulink system 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, for instance the unit delay block), 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 system 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 system 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 the *dependency graph* of a Simulink system. For the example presented in Fig. 3, the dependency graph between blocks is shown in Fig. 4. 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.

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 must be a multiple of the ground sampling time called the *step size*. The value of the step size for a given Simulink model can be either specified explicitly or is determined by the simulation engine such that the bounds on approximation errors of integrator blocks are within some specified threshold. So there are two different simulation modes, namely *fixed step solver* and *variable step solver*. In a fixed step solver, a simulation step occurs every *step size* time units. In a variable step solver, 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. For a description of the algorithm underlying the two solver modes, please refer to [25].

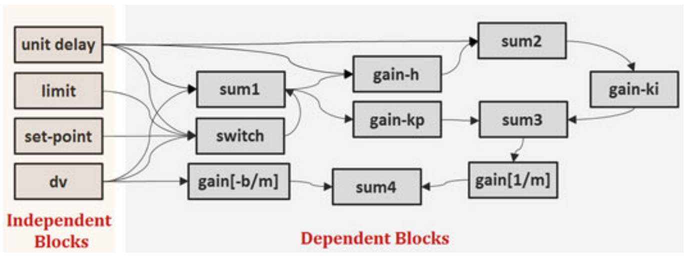

Fig. 4 Dependency graph: Cruise controller

2.4 Stateflow

Stateflow is almost a Simulink block except that it can have more than one output signal. A Stateflow block models a combination of communicating hierarchial and parallel state machines (resembling Statecharts) and flowcharts. The three basic concepts of Stateflow are states, transitions, and junctions.

A *state* in a Stateflow model has a hierarchial parent–child structure representing its decomposition, i.e., into substates. A state can be decomposed into either exclusive OR substates connected with transitions (directed edges) or parallel AND substates that are disconnected. The former is represented by substates with solid borders and the latter by substates with dashed borders. The parent of these substates are referred to as a superstate. If a superstate has an OR decomposition then it is active if exactly one of its substate is active. If a superstate has an AND decomposition then it is active if all its substates are active. In Fig. 5, we give an example of AND/OR state decomposition in Stateflow and represent the parent–child relationship between a superstate and its substates as a tree, starting with a root node (not explicitly represented in Statecharts). To distinguish between superstates that have an AND/OR decomposition, we represent outgoing edges of an AND superstate by dashed edges. All leaf nodes represent atomic substates, i.e., they have no children. Note that an AND superstate having a child substate with an AND decomposition is a redundant specification because this decomposition can be represented by the parent AND superstate itself. Note that if a nonatomic substate is active then its superstate is active as well.

Furthermore, a state in Stateflow has a set of associated actions. The action language consists of variable assignments or emission of events.

- Entry action: This action is executed when the state is entered.
- During action: This action is executed when no valid outgoing transitions exist in an active state.
- Exit action: This action is executed when a valid outgoing transition has been found.
- On event E action: This action is executed when an event E occurs in an active state.

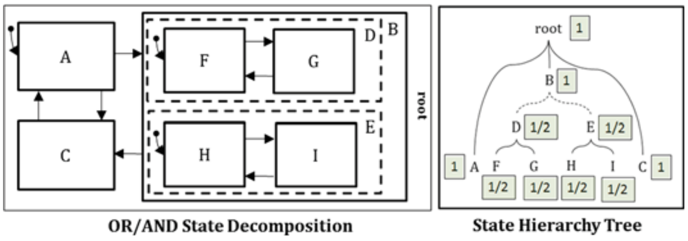

Fig. 5 State hierarchy in stateflow

In addition, Stateflow defines a special type of state called a *junction*. In Stateflow models a junction is represented as a circle and is used to model logical decision patterns using a flowchart type notation. However, unlike a state, a junction does not have any hierarchy, nor any associated actions and must be exited instantaneously, when entered.

A *transition* is represented by a directed edge and models the passage of a system from one state to another. A transition may also have a junction in its source or(and) destination (also referred to as transition segment). The semantics of a transition is given in terms of an expression (specified as a label) having the form $E[C]\{Ac\}/At$, where E is the event, C is the guard condition, Ac is the condition action, At is the transition action, which means that a transition can occur only if an event E has occurred and the guard condition C is true. If the guard condition is true then the condition action Ac is executed. The transition action At is executed when a destination state has been found. So in case of a transition from one state to another state, transition action coincides with a condition action. However, in case of transitions connecting junctions, transition actions are executed (in the order they were visited) only when a valid path has been found to a state. Furthermore, all parts of a transition's expression are optional. Note that we will not consider the nested event broadcast feature in actions of Stateflow transitions because its preemption semantics (see [3]) is a bad practice and often leads to unpredictable results. So we will consider this feature without preemption.

Furthermore, transitions are allowed to cross superstate boundaries to a substate destination, referred to as *inter level transitions*. In Stateflow semantics, an initial state of a set of substates belonging to a parent superstate having an OR decomposition is specified using a special type of transition called a *default transition*. It is represented as a directed edge having no source to a single destination state. Furthermore, for a superstate, Stateflow has the notion of a *history junction* denoted as the symbol H with a circle around it. Such a junction remembers the last active substate of the superstate when it is exited and assigns this substate as the initial state. We give an example of a Stateflow in Fig. 6.

3 CPN*: A Compact Notation for CPN

Although the CPN formalism is expressive enough for most practical applications (as they are Turing complete), the modeling of large systems often lead to large models that are cumbersome and difficult to read. In this paper, we focus on modeling hybrid systems expressed partly as a set of difference and differential equations and partly by some kind of state transition diagram. In this context, we will identify few recurring CPN patterns and show how they can be graphically represented in a more compact and handy manner without changing the underlying semantics. We call the resulting CPN model as a CPN* model.

Given an arbitrary CPN model (see the example in Fig. 7.), we distinguish between two types of its places, namely Signal Places and State Places.

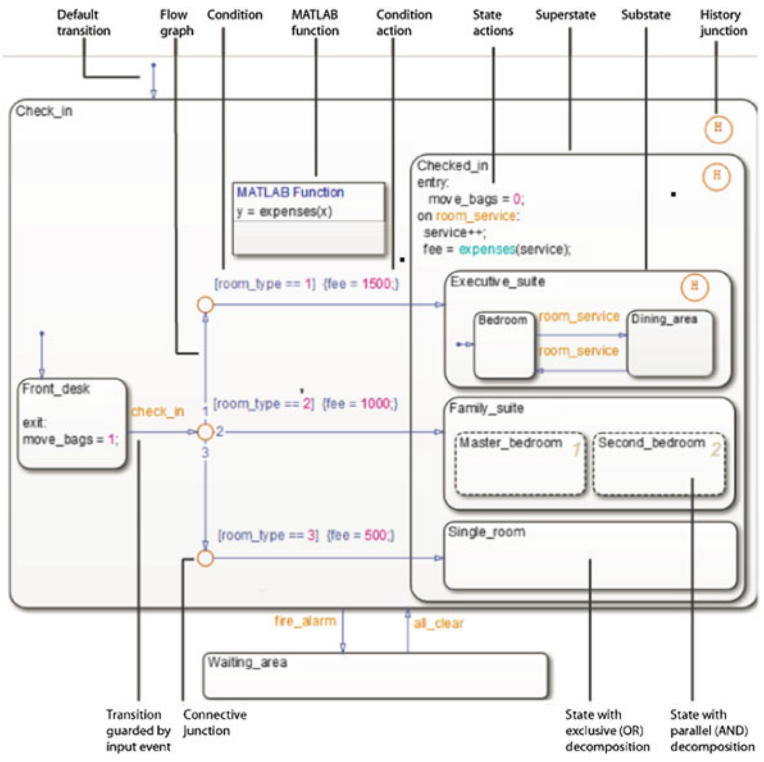

Fig. 6 An example of stateflow showing all its features [3]

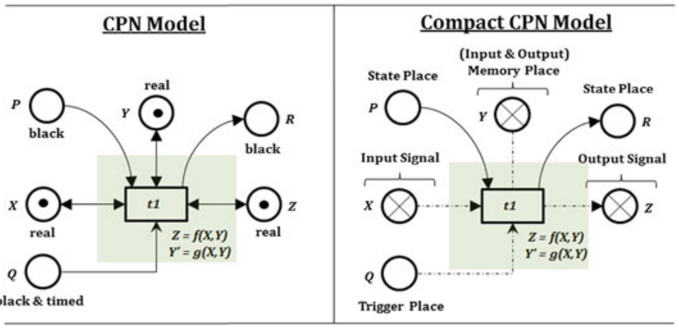

Fig. 7 A CPN and its CPN*

- **Signal Places**. A CPN place is called a *signal place* (a) if it is untimed and has a color type *real* having exactly one token in the initial marking of the CPN model, and (b) it has a preset that is equal to its postset, i.e., transitions that consume a token from it also produce back a token into it. As a consequence, a signal place has exactly one token in any reachable marking of the net system, i.e., it is always marked and safe. In models of hybrid systems, a marked signal place can be seen as modeling a *variable*, whose valuation is given by the color value of the token in that place. As a compact notation, we represent a marked signal place by a crossed circle.

- **Trigger Places**. A CPN place is called a *trigger place* if (a) it is timed and has a color type with only one value, which we call *black* (i.e., it can contain only *black* tokens like in a classical Petri net), and (b) it has a postset of size equal to one. In models of hybrid systems, a trigger place is used to model an event of a system. To make the distinction from other places of a CPN more graphically explicit, we will distinguish it by representing its incoming (outgoing) arcs to (from) this place by a dashed directed edge.

A transition of a CPN model is called a *function transition* if (a) it has associated with it signal places and exactly one trigger place, and (b) it defines one or more functions over its connected signal places (i.e., modeling variables). The function definition of such a transition induces a distinction between signal places as either an input, an output, or both. We call a signal place of a transition that is both an input and an output as a *memory signal place*.

In the CPN model of Fig. 7, we see the annotations of the transition $t1$ as $Z = f(X, Y)$ and $Y' = g(X, Y)$. The meaning of these annotations is that Z, X, Y are signal places and Y' is the same signal place as Y but Y' is the updated value after evaluation of function g. From this annotation, we can derive that X is an input and Z is an output for transition $t1$, which is indicated by the arrow on the arc between the signal place and the transition. For a memory signal place like Y, we have no direction on the arc. Note that these arcs and the use of directions are different from the meaning in the CPN model itself.

For the purpose of modeling Stateflow, we identify state places and super places of a given CPN model.

- **State Place**. A CPN place is called a *state place* if (a) it is untimed and has a color type *black*. We call a transition that has associated with it one or more state places as a *state transition*. So transition in Fig. 2 is also a state transition.
- **Super Place**. A *super place* is a subset of places of a CPN denoted by a solid box around its places. The set of all state places of a CPN is also a super place. We require that two super places of a CPN are either disjoint or one is contained in the other. A super place q is a child of super place p and p is a parent of super place q if and only if q is contained in p and no other super place r satisfies $q \subseteq r \subseteq p$. Due to this property, it is easy to see that the set of all super places forms a tree since if q is a child of p and q is a child of r then p and r are not disjoint. So either $p \subseteq r$ which means q is not a child of r, or $r \subseteq p$ which means q is not a child of p.
- **Super Place Net**. The subnet of a CPN model containing exactly the state places of a super place and all transitions that have their preset and postset contained in q is called the *super place net*. We require that super place nets are connected.

4 Modeling Hybrid Systems

In this section, we show how a hybrid system can be expressed as a CPN* model. We start with modeling arbitrary functions that do not have a notion of time and then show how these models can be extended to model functions having time as a parameter. Finally, we show how functions can be combined to model a system, both in a general way and in the Simulink framework.

4.1 Modeling Untimed Functions

From a modeling perspective, we distinguish between two types of functions, namely memoryless functions and memory functions.

The output of a *memoryless function* is a function of only its input variables. As an example, consider the memoryless function $Z = f(X, Y)$, with input variables X and Y and output variable Z. The memoryless function f is modeled in CPN* by assigning it to a function transition having two input signal places X and Y and an output signal place Z. We show the corresponding CPN* model in Fig. 8.

On the other hand, a *memory function* has also a variable modeling its memory (data store). The output of such a function depends not only on its input variables but also on its memory variable. Such a function may also define an additional *update function* to overwrite the current value of its memory variable. This update function depends on input variables or(and) the memory variable itself. As an example, consider the memory function $Z = f(X, Y)$, where X is an input variable, Z is the output variable, and Y is the memory variable with an associated update function $Y' = g(X, Y)$. We model a memory function f along with its update function g in CPN*, by assigning the two functions to a function transition having one input signal place X, one output signal place Z, and one memory place Y. We show the corresponding CPN* model in Fig. 8. Note that two or more transitions may share the same memory place.

So far we have seen how a function can be modeled using a function transition, signal places, and memory places. Using these concepts it is straightforward to model a set of functions related to each other by input/output variables and mem-

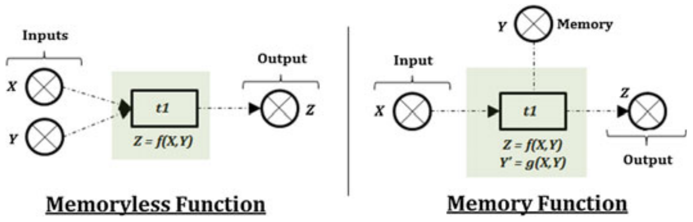

Fig. 8 Memory function and memoryless function

ory variables (by sharing common signal/memory places) as a network of function transitions. Such a network of function transitions defines a predecessor/successor relationship between them. Given a function transition, its predecessor (successor) transition is a transition whose output (input) signal place is the input (output) signal place of the given transition.

Although a network of function transitions is syntactically correct, it is semantically not well defined because all its transitions are always enabled. So it is not clear what the output of such a system really means, even if it is an acyclic network! For a cyclic net as shown in Fig. 9, the process (i.e., the successive values of the signals) is completely nondeterministic. Therefore, we use a trigger place as shown on the right-hand side of Fig. 9. Now the values of the signals X and Y can be computed as the sequence $(X_0, Y_0), \dots, (X_n, Y_n)$, where for some $i \in \{1 \dots n\}$, (X_i, Y_i) represents the value of the signals X and Y after firing transitions $t1$ and $t2$, i times, i.e., $(g \circ f)^i(X_0) = X_i$ and $f(g \circ f)^{i-1}(X_0) = Y_i$.

Next we show two ways of using a trigger place for a transition modeling an arbitrary function. A transition is *predecessor triggered* if every predecessor transition is connected only by an outgoing arc to the trigger place of this transition. A transition is *self-triggered* if it is connected to its own trigger place and there are no other transitions in either the postset or preset of this trigger place. Note that a transition with no predecessor transition must be self-triggered. Furthermore, the trigger place of a self-triggered transition has exactly one token in the initial marking.

In Fig. 10, we give an example of a CPN* with self-triggered and predecessor triggered transitions. The function h (modeled by transition $t2$) is a memory function having no predecessor. So it is self-triggered. This is also the case with function g (modeled by transition $t3$) except that it is memoryless. Furthermore, function g produces an output Y that is an input to the functions f (modeled by transition $t1$) and r (modeled by transition $t4$). As a result, transition $t3$ is connected to the trigger place of transitions $t1$ and $t4$. Since function f receives an input X from function h, its trigger place is connected to it.

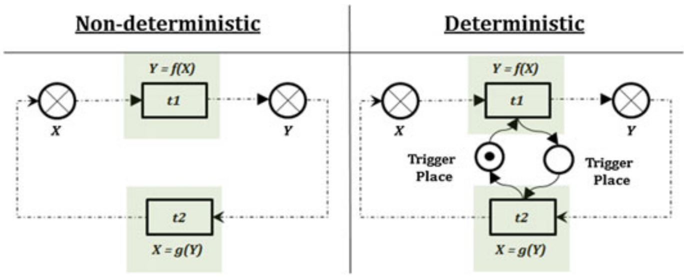

Fig. 9 Modeling a network of function transitions

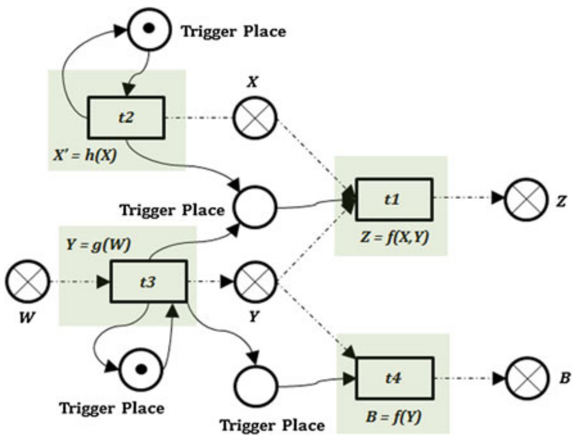

Fig. 10 CPN* with self-triggered and predecessor triggered transitions

4.2 Modeling Timed Functions

In a hybrid system, the notion of state progression, i.e., the successive valuation of its variables, are a consequence of the progression of time. So time is an important aspect of modeling such systems. Here functions have time as a parameter, for instance in ordinary differential equations having time as a parameter. To model such functions in CPN*, we must address the following two questions:

1. When Should the Functions of the System Be Evaluated? We capture this aspect by making a distinction between functions, namely *timer functions* and *reactive functions*. Note that both timer functions and reactive functions can be either memoryless or memory functions.

A *timer function* has an associated time attribute called the *sample time*, which specifies how often its output must be computed. A *reactive function* has no obligation as to how often its output must be computed. So it only computes an output whenever there is a change in the values of any of its input variables (indicated by a token in its trigger place).

Modeling. A reactive function is modeled as a predecessor triggered transition. A timer function is modeled as a self-triggered transition having a delay (corresponding the sample time of the function) on the incoming arc to the trigger place. We give an example of each type in Fig. 11.

2. How Can Time Be used as a Parameter in a Modeled Function? To model a hybrid system, the function definitions must also allow the inclusion of time as a parameter. However, the time parameter may either be relative to the model called the *absolute model time* or relative to a function called the *relative function time*. The former represents the time that has elapsed since the model began its execution, while the latter represents the time that has elapsed since the last execution of a given function.

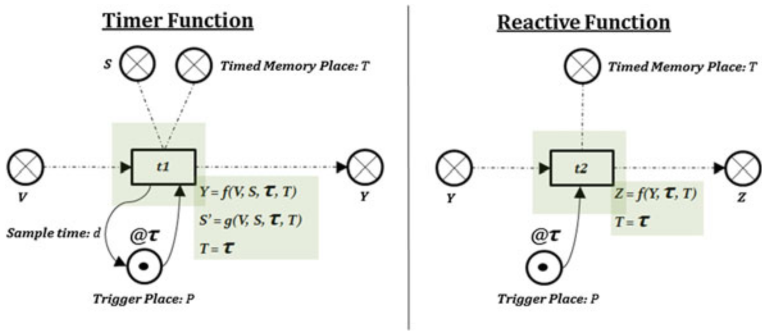

Fig. 11 Timed functions

Modeling. A transition modeling an arbitrary function may access the absolute model time by looking up the timestamp of the enabling token in its trigger place, denoted by τ. Note that by the eagerness property of CPN, this is the earliest token timestamp in the trigger place.

The relative function time of a given transition can be computed by first adding a memory place to it without any delays on any of its arcs. We call this memory place as a *timed memory place*. Its role is to store the absolute model time at which its connected function transition last fired. We model this by specifying an update function that produces as its output, the timestamp of the enabling token in the trigger place. Now the the relative function time can be computed by subtracting the color value of the token in the timed memory place from the timestamp of enabling token in the trigger place. See Fig. 11.

4.3 Modeling Simulink System and its Simulation Using CPN*

In this section, we give a formal semantics to Simulink using the CPN* formalism. First, we show how a Simulink block can be modeled in CPN* as a transition function having at most one output, which we will call a Petri net block (*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*. Note that the execution of the block execution order is carried out by the underlying simulation engine, so it is an implementation detail and not a modeling concern for designers.

Modeling Simulink Blocks. It is straightforward to model the two basic types of Simulink blocks, namely stateless blocks and stateful blocks in the CPN* notation as memoryless functions and memory functions, respectively. Furthermore, since most of these blocks model functions having time as a parameter and have an associated sample time, we extend them with the notion of time as explained in Sect. 4.2. We call a CPN* model of a Simulink block a P-block. In Fig. 12, we give an example of modeling a discrete derivative block as a P-block having a sample time 3. Note that the initial condition of a stateful P-block is specified by the initial color value of its memory place(s).

Modeling Simulink System. Given a set of P-blocks, we model a P-system by fusing the shared input signal places of blocks with the output signal place of blocks. Note that two or more blocks are not allowed to have the same output signal place because in Simulink, blocks have only one output and this is modeled in P-blocks with a unique output signal 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 Fig. 13.

Modeling Simulation of Simulink System as Control Flow. We have seen how an arbitrary Simulink system can be expressed as a P-system. The enabling of P-blocks

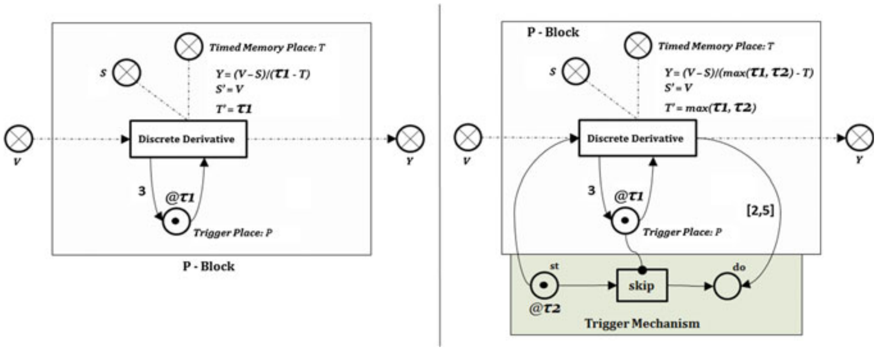

Fig. 12 P-block of the discrete derivative block

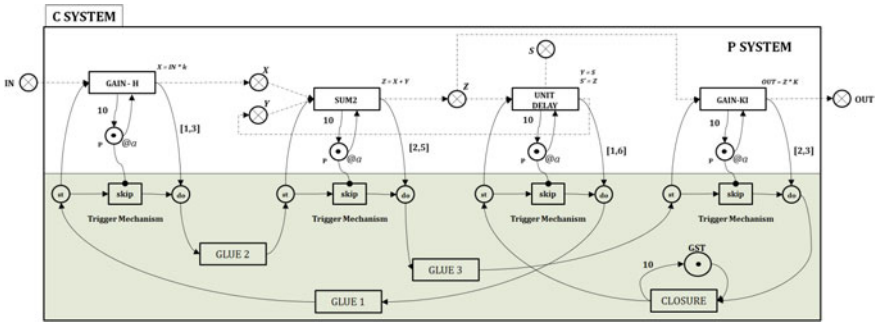

Fig. 13 C-system: integral part (PI controller)

in a P-system is determined by the timed availability of a token in the trigger place of function transitions. 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 nondeterministic 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 Fig. 13. 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 Fig. 4, is the sequence: ⟨*unit delay; gain-h; sum2; gain-ki*⟩. 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. Furthermore, we also model that Simulink blocks have a nondeterministic *execution time* (i.e., the time it takes for a block to produce its output). In CPN*, such an execution time can be associated with a transition by specifying a time interval along an outgoing arc from it. We call the resulting system a *C-system*.

The *trigger mechanism* is modeled on top of a P-block by adding a timed input place *start* (labeled *st*), a timed output place *done* (labeled *do*), and a timed inhibitor transition (inhibitor arc that accounts for the availability of token in the trigger place of that block) called *skip*. If the token in place *start* is available and the token in the trigger place 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 the trigger place), then the block transition (discrete derivative) fires at its enabling time and produces a token in the place *done*. Note that the functions requiring an absolute model time may compute it by taking the maximum of the timestamp of enabling tokens in the trigger place (called τ_1) and start place (τ_2). Furthermore, along the arc to the place *done* it is possible to specify the *execution time* of the block as a time interval. In both cases, one timed token is produced in the place *done*. In Fig. 12, we give an example of a P-block, extended with a trigger mechanism and modeling an execution time in the interval [2, 5].

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 *done* of the preceding block and produces a token in the place *start* 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 *done* belonging to the last block in the sorted order and produces a token in the place *start* corresponding the first block in the sorted order, and (b) we initialize the place *done* 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 Fig. 13, we describe the C-system of the integral part of the PI controller having a step size of 10 time units. In [25, 26], we show some interesting properties of C-systems that can be verified by model checking.

5 Modeling Stateflow Using CPN*

Stateflow is a Simulink block except that (a) it can have more than one output signal, (b) its internal structure is defined as a hierarchial and concurrent state machine, and (c) its execution is either triggered by the simulation engine (in a simulation step), or triggered by events generated by Simulink/Stateflow blocks.

In this section, we will consider the relationship between Stateflow models and CPN*. For this we address the following three questions; First, we show how CPN* models can be constructed in *Stateflow style* by stepwise refinements using two refinement operations. Then we consider how an arbitrary CPN* model can be checked for conformance to the Stateflow style. Finally, we show how a CPN* model can be derived from a given Stateflow model.

5.1 Constructing CPN Models in Stateflow Style*

In this section, we define a Stateflow style construction method for CPN*. It is based on stepwise refinement with the two refinement techniques, namely place refinement and place duplication corresponding to an OR decomposition and an AND decomposition, respectively. For the construction, we will consider state machine nets that have only state places (We will add the signal and trigger places on top of the CPN* model, derived using the construction method).

Place Refinement. Given a place p in a CPN* model and a state machine net N, the *place refinement* operation replaces place p with the net N in the following way: Remove place p from the CPN* model and perform a net union of N (i.e., pairwise union of its places, transitions and relationships between them) with the given CPN* model. Next, connect all input arcs of p to places of net N and all output arcs of p to places of net N. Note that the modeler has the freedom to choose arbitrary places to connect these arcs.

Note that place p in the original CPN* model becomes a parent super place containing the set of state places belonging to the state machine net N in the resulting

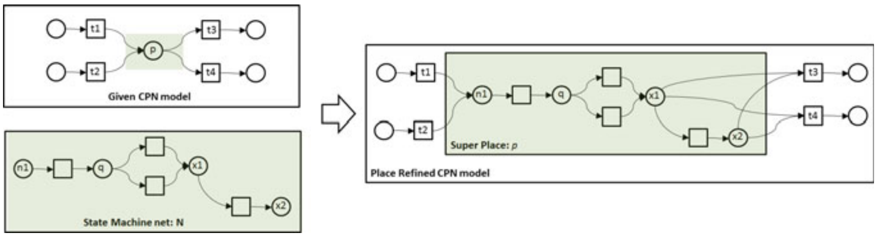

Fig. 14 Place refinement

CPN* model. We indicate this relationship by drawing a solid box around the nodes of the net N and labeling it with the name of the super place p. Furthermore, observe that the state machine net N is a super place net of the resulting CPN. See Fig. 14. Note that this operation corresponds to an OR operation of a superstate into its mutually exclusive substates in Stateflow.

Place Duplication. The place duplication operation provides a structured way to introduce concurrency into a CPN* model. The operation is defined in the following way:

Given a place p in a CPN* model and a set of places Q not belonging to this CPN* model, the *place duplication* operation adds the set Q to the set of places of this CPN* model in the following way: Remove place p from the CPN* model and add the set of places Q to the CPN* model such that for each place in the set Q, its preset (postset) is equal to the preset (postset) of p.

Note that place p in the original CPN* model becomes a parent super place containing the concurrent state places modeled by the set of places Q. We indicate this relationship by drawing a dashed box around the places of set Q and labeling it with the name of the super place p. We give an example in Fig. 15. Note that this operation corresponds an AND operation of a superstate into concurrent substates in Stateflow.
Construction Method. Start with a set of super places represented as a tree and associate each super place as having either an OR decomposition or an AND decomposition. Next we construct a CPN* model having no transitions and exactly one state place corresponding the root of the super place tree. Then set the *current node* to the root node of the tree and carry out the following steps in an iterative manner until all nodes have been *visited*. Note that after each iteration, we have a CPN* model.

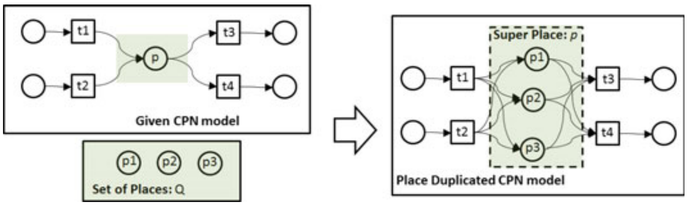

Fig. 15 Place duplication

- If the *current node* has an OR decomposition then
  - Construct a state machine net N containing a labeled place for each corresponding child of the current node.
  - For the place in the CPN* model corresponding the current node, carry out a place refinement with the state machine net N.
- If the *current node* has an AND decomposition then
  - Construct a set of places Q containing a labeled place for each corresponding child of the current node.
  - For the place in the CPN* model corresponding the current node, carry out a place duplication operation with the set of places Q.
- Mark the current node as *visited*
- If *there exits an unvisited child of a visited parent* then *set it as the current node*, otherwise *stop*.

We call the resulting CPN* model a *Stateflow Petri net* like in [4]. We also consider some interesting properties presented in [4]. To present them, we first give a few definitions.

Active Super Place and Correct Configuration. A super place is called an *active super place* if at least one of its state places contains a token. A marking of a Stateflow Petri net is said to be a *correct configuration* if and only if the following properties hold:

- The root super place is active.
- An active super place corresponding to an OR node has exactly one of its children active.
- An active super place corresponding to an AND node has all its children active.

The state places of a Stateflow Petri net satisfies an invariant property by construction. To see this, we first assign weights to each node of the state hierarchy tree according to the following rules:

- The root node of the tree has a weight equal to one.
- The children of an OR node in the tree inherit the weight of their parent.
- Each children of an AND node in the tree obtains a weight equal to *the weight of their parent* $\times$ (*number of children*)$^{-1}$.

Next to each node of the tree in Fig. 5, we indicate their node weights. For a Stateflow Petri net, it is easy to prove the following properties:

- A Stateflow Petri net is safe.
- In any reachable marking from an initial marking that is a correct configuration, the sum of the product of the number of tokens in a place (either zero or one) and its weight is a constant, i.e., the weight function is a place invariant (see [20, 21]).
- If a Stateflow Petri net is initially in a correct configuration then all its reachable markings are also correct configurations.

On top of a Stateflow Petri net, we may add signal places and trigger places.

5.2 Verifying Conformance of CPN* Models to Stateflow Style

Given a CPN* model, we verify that it conforms to the *Stateflow style* of construction by first deleting all its trigger places and signal places and then verifying if the resulting CPN* model satisfies the following property: *Is it possible to reduce the CPN* model to a single place by applying the refinement operations in their inverted form as reduction rules?* If this is true, then the CPN* model conforms to the Stateflow style, i.e., it is possible to derive the CPN* model by stepwise refinements with the two refinement operations. Note that in [4], a similar reduction is used without considering its inverted form as refinement rules.

5.3 Constructing CPN* Model from Stateflow Model

Now we present a *construction method* to derive a CPN* model from a given Stateflow model. However, before we use an arbitrary Stateflow model, we need to preprocess it due to unstructured and complicated semantics associated with flowchart notations (specified by junctions and transition segments modeling decisions and loop-based constructs) and redundant semantics associated with all kinds of actions in its states and transitions.

Flatten Stateflow. First, eliminate flowchart semantics in Stateflow by identifying a set of transition segments leading from a state to another state and defining their semantics (i.e., early return logic [3]) in terms of a recursive function. Then replace this set of transition segments by a single transition and associate it with the defined function. Note that in CPN, it is possible to express such a recursive function using the functional programming language ML. If a set of transition segments are such that they originate at a junction or terminate at a junction, then we treat the origin/termination junction as a state in Stateflow and refer to the transition replacing them as a junction transition. If both the origin and termination are a junction, then we do the same modification after adding a new transition (which we call closure transition) from the terminating junction to the origin and treating this connected pair of states as an AND decomposition of the root (see [3] for semantics of flowchart). Furthermore, to keep the distinction between transitions of Stateflow and CPN clear, we will refer to a transition in the former as an edge.

- Given a flattened Stateflow model, construct its state hierarchy tree.
- **Add Places and Super Place Notations**. Start with an empty CPN* model, (i.e., having no places and no transitions) and then add state places to it in the following way:
  - For each leaf node of the state hierarchy tree, add one corresponding state place to the CPN* model.

  - For each parent superstate in the state hierarchy tree, draw a solid box around the state places in the CPN* model corresponding its children.

- **Super Place Net Construction**. Next add the transitions to the CPN* model in the following way:
  - For each edge of the Stateflow model add a transition to the CPN* model.
  - **Extended Stateflow**. Next, extend the Stateflow model in the following way: For each interlevel edge having a destination as a superstate, *recursively* extend this edge to the nested initial atomic substate (indicated by the default transition) of this superstate. If the destination superstate has an AND decomposition, then create an additional edge, one for each substate. Note that in an extended Stateflow model, all interlevel edges have as their destination, an atomic substate (corresponding state places in our CPN* model).
  - For each transition in the CPN* model connect its postset to the state place corresponding the destination substate of the extended interlevel edge. If an extended edge has an atomic substate (leaf node) as its source, then add to the preset of the corresponding transition, the corresponding state place. If an extended edge has a super state as its source, then add reset arcs from the corresponding transition to every place in the corresponding super place.

In the resulting CPN* model, we associate signal places (modeling variables) and trigger places (modeling events) to transitions as they occur in the Stateflow model. A condition associated with an edge is expressed as a guard condition of corresponding transitions in the CPN* model. Since transition segments have been eliminated in the preprocessed Stateflow model, the definition of a condition action and transition action, coincide. These actions are specified as a function definition of a transition in CPN*. Furthermore, we eliminate the redundant definition of actions in states in the following way: An entry (exit) action of a state is modeled in CPN* as a function associated with all transitions in the preset (postset) of the corresponding state place. The during action of a state is modeled by adding a self-loop to a state place, i.e., a transition with preset and postset as the state place, and associating this transition to the function specified by the during action. An on event action of a state is modeled in a similar way but in addition, we associate a trigger place to the self-loop transition. It is a simple modeling exercise to express the semantics of a history junction in the CPN* model by duplicating each state places and using these copies to record the last active state place. Lastly, we model the triggering of a Stateflow block by the simulation engine in each simulation step. For this we add a memory place that is timed (i.e., can contain tokens with timestamps) and connect it to all transitions (except junction transitions) of the Stateflow model. Then by default, we have the *superstep semantics* (see [3]). For *step semantics*, we must add a delay equal to the ground sampling time of the system on each incoming arc to the newly added (timed) memory place.

6 Conclusions

We have shown how hybrid systems can be modeled in CPN*, a compact notation for CPN models. As a special case, we have considered the translation of Simulink/Stateflow models into the CPN* formalism. The result is that the models in both these formalism have the same size but the the CPN* formalism has many advantages: First, it has a well-defined semantics and it is simple, as compared to verbose semantic descriptions of the Simulink/Stateflow toolset. Second, it is a strong extension of the Simulink/Stateflow framework by not treating Stateflow elements as a block, i.e., it is possible to use elements of a Stateflow block on the same footing as Simulink blocks. Third, CPN* models exhibit true concurrency unlike the sequential semantics of Stateflow models. Lastly, there are numerous numerous Petri net analysis techniques and tools available to verify CPN* models by model checking,

References

1. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**, 231–274 (1987)
2. website, S.: http://nl.mathworks.com/help/simulink/
3. website, S.: http://nl.mathworks.com/help/stateflow/
4. Eshuis, R.: Translating safe petri nets to statecharts in a structure-preserving way. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 239–255. Springer, Heidelberg (2009)
5. Huszerl, G., Majzik, I., Pataricza, A., Kosmidis, K., Dal Cin, M.: Quantitative analysis of UML statechart models of dependable systems. Comput. J. **45**, 260–277 (2002)
6. Merseguer, J., Campos, J., Bernardi, S., Donatelli, S.: A compositional semantics for UML state machines aimed at performance evaluation. In: Proceedings. Sixth International Workshop on Discrete Event Systems, IEEE, pp. 295–302 (2002)
7. Agrawal, A., Simon, G., Karsai, G.: Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. Electron. Notes Theoret. Comput. Sci. **109**, 43–56 (2004)
8. Tripakis, S., Sofronis, C., Caspi, P., Curic, A.: Translating discrete-time simulink to Lustre. ACM Trans. Embed. Comput. Syst. **4**, 779–818 (2005)
9. Denckla, B., Mosterman, P.J.:. Formalizing causal block diagrams for modeling a class of hybrid dynamic systems. In: IEEE CDC-ECC, pp. 4193–4198 (2005)
10. Tiwari, A.: Formal semantics and analysis methods for simulink stateflow models. Technical Report, SRI International (2002)
11. Zhou, C., Kumar, R.: Semantic translation of simulink diagrams to input/output extended finite automata. Disc. Event Dyn. Sys. **22**, 223–247 (2012)
12. Bouissou, O., Chapoutot, A.: 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, ACM, pp. 129–138 (2012)
13. Hamon, G., Rushby, J.: An operational semantics for stateflow. Int. J. Softw. Tools Technol. Transf. **9**, 447–456 (2007)
14. Hamon, G.: A denotational semantics for stateflow. In: Proceedings of the 5th ACM International Conference on Embedded Software, ACM, pp. 164–172 (2005)
15. Scaife, N., Sofronis, C., Caspi, P., Tripakis, S., Maraninchi, F.: Defining and translating a safe subset of simulink/stateflow into Lustre. In: Proceedings of the 4th ACM International Conference on Embedded Software, ACM, pp. 259–268 (2004)

16. Jensen, K., Kristensen, L.M., Wells, L.: Coloured petri nets and CPN tools for modelling and validation of concurrent systems. Int. J. Softw. Tools Technol. Transf. **9**, 213–254 (2007)
17. David, R., Alla, H.: Discrete, Continuous, and Hybrid Petri nets. Springer, Berlin (2010)
18. Demongodin, I., Koussoulas, N.T.: Differential Petri nets: Representing continuous systems in a discrete-event world. IEEE Transactions on Automatic Control **43**, 573–579 (1998)
19. Gilbert, D., Heiner, M.: From petri nets to differential equations—an integrative approach for biochemical network analysis. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 181–200. Springer, Heidelberg (2006)
20. Reisig, W.: Petri Nets: An Introduction. Springer, New York (1985)
21. Peterson, J.L.: Petri net theory and the modeling of systems. Prentice Hall PTR, Upper Saddle River (1981)
22. CPN Website: www.cpntools.org/
23. Bera, D., van Hee, K., Sidorova, N.: Discrete timed petri nets. Computer Science Report 13–03, Technische Universiteit Eindhoven (2013)
24. van Hee, K., Sidorova, N.: The right timing: reflections on the modeling and analysis of time. In: Colom, J.-M., Desel, J. (eds.) Petri Nets 2013. LNCS, vol. 7927, pp. 1–20. Springer, Heidelberg (2013)
25. Bera, D., van Hee, K., Nijmeijer, H.: Relationship between simulink and petri nets. Computer Science Report 14–06, TU Eindhoven (2014)
26. Bera, D., van Hee, K., Nijmeijer, H.: Relationship between simulink and petri nets. In: Obaidat, M., Kacprzyk, J., Oren, T. (eds.) Proceedings of SIMULTECH 2014: Fourth International Conference on Simulation and Modeling Methodologies. SCITEPRESS, Technologies and Applications (2014)