MODELING PETRI NETS AS LOCAL CONSTRAINT EQUATIONS FOR HYBRID SYSTEMS USING MODELICA™

Pieter J. Mosterman[†*] **Martin Otter**[†] **Hilding Elmqvist**[‡]

[†]DLR Oberpfaffenhofen, D-82230 Wessling, Germany, Pieter.J.Mosterman@dlr.de/Martin.Otter@dlr.de

[‡]Dynasim AB, Research Park Ideon, SE–223 70 Lund, Sweden, Elmqvist@Dynasim.se

KEYWORDS

Modelica, hybrid systems, embedded control systems, physical system modeling, object-oriented modeling.

ABSTRACT

To model complex mixed continuous/discrete, *hybrid*, systems, the continuous part can be described by differential and algebraic equations using an object-oriented modeling language such as Modelica. It is shown how a discrete formalism such as Petri nets can be incorporated by describing all components by strictly *local* equations. This allows a unifying treatment since both the continuous and the discrete part of a system are described by equations. The resulting Petri net library is applied to model the redundancy management control of the elevator subsystem of an aircraft.

INTRODUCTION

Complex dynamic systems often consist of piecewise continuous physical processes controlled by discrete computing hardware (Mosterman, Biswas, & Sztipanovits 1997). To model such systems requires a formalism that captures both complex continuous behavior specified by a large number of equations, as well as intricate process control schemes described by a discrete event formalism. Models that facilitate these combined continuous/discrete characteristics are called *hybrid systems*.[1]

There are two main approaches to model hybrid system behavior. Discrete modeling formalisms, such as finite state machines, Petri nets, state charts or CSP are extended with continuous dynamics, e.g., hybrid automata (Alur *et al.* 1993; Kowalewski *et al.* 1998; Murata 1989). Alternatively, the continuous formalism of differential equations can be augmented with discrete aspects. In both cases, formalisms usually lack strength in the added area.

In this paper, *Modelica*^{TM} (Modelica 1997), a unified object-oriented language for physical system modeling, is used. Modelica[2] components are mathematically described by differential and by algebraic equations with some discrete event features to handle discontinuities and sampled data systems. Modelica is developed in an international effort by more than 10 institutions, and is a small non-causal language intended for modeling across many application domains, such as electrical circuits, multi-body systems, drive trains, hydraulics, thermodynamical, and chemical systems. Furthermore, it is designed for model exchange between different modeling and simulation environments.

Although Modelica is primarily designed to handle continuous systems, it is possible to capture discrete phenomena using the present language elements. This paper shows how Petri nets can be described by strictly local sets of Boolean equations for place and transition components, leading to a compact realization which can be incorporated into an object-oriented modeling system such as Modelica. The visualization of Petri nets as object diagrams in Modelica is close to the common visualization of Petri nets.

PETRI NETS

Petri nets are an intuitive visual formalism to model discrete event behavior. It is especially well suited to deal with parallelism and the corresponding issues of synchronization and nondeterminism.

Petri Net Semantics

The basic elements of a Petri net are *places* and *transitions* with directed connections between them. Places and transitions appear alternatingly, and, therefore, a Petri net is a directed bipartite graph. Places can contain *tokens* that are moved around by transitions that

*Pieter J. Mosterman is supported by a grant from the DFG Schwerpunktprogramm KONDISK.

[1]See also the IEEE Control Systems Society CACSD Working Group on Hybrid Dynamic Systems web site at http://www-er.df.op.dlr.de/cacsd/hds.

[2]Modelica is a trademark of the Modelica Design Group.

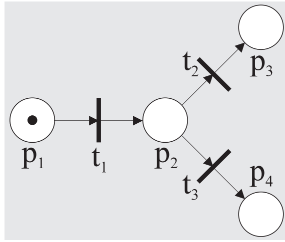

Figure 1: A Petri net.

fire according to the following rules: If (i) all of the transition's input places have sufficient tokens to *enable* it, (ii) tokens from the *input places* are removed and tokens are added to the *output places.* For example, in Fig. 1, since a token is present in p_1, t_1 is enabled. When it fires, the token is removed from p_1 and placed in p_2. In general, a place can hold multiple tokens.

A place can be input and output to the same transition, a *self-loop*. When a place is connected to two transitions and it is the only input place to these transitions, then it is undetermined which transition fires. In Fig. 1, this occurs when p_2 has a token. Both t_2 and t_3 are enabled but only one fires. Which one depends on the execution semantics. If no priorities are assigned, the token should be distributed *fairly* (i.e., if the situation occurs a number of times, p_3 and p_4 should receive the token equally often). This nondeterminism is inherent in Petri nets. In case priorities are assigned, t_2 may always be selected to fire over t_3 or the other way around. Now, the system has a deterministic execution.

Based on the particular problem, different semantics of a number of types of Petri nets (Murata 1989; Starke 1990) may be applied:

- *Bounded Petri nets* enforce an upper bound on the number of tokens each place can hold (i.e., their *capacity*). This requires a modification of the firing rule, since a transition can only fire when the new number of tokens in each of its output places would not exceed their respective capacities.
- *Normal Petri nets* are bounded Petri nets with places of capacity 1.
- *Inhibitor Petri nets* contain place to transition connections that prevent the transition from firing.
- *Priority Petri nets* allow for assigning priority to transitions. When two transitions are enabled, the one with the highest priority fires. This can be applied to eliminate nondeterminism.
- *Maximum firing Petri nets* require transitions that are enabled to fire immediately (possibly taking priorities into account). This is a *must* fire semantics as opposed to the *may* fire semantics of Petri nets in general.
- *Colored Petri nets* assign attributes to tokens and transitions can fire based on the presence of tokens of a specific type in its input places.
- *Time Petri nets* associate a lower and upper temporal bound with transitions. A transition may fire as soon as time exceeds its lower bound and it has to fire before its upper bound is reached, unless disabled before then.
- *Timed Petri nets* associate a duration with transitions. When a transition fires, tokens in the input places are removed and after the duration has passed, tokens are moved into the output places. Timed Petri nets can be expressed in terms of time Petri nets.

Note, that the *finite state machine* formalism (Kohavi 1978) is a subset of Petri nets where only transitions with one input and one output are allowed. In addition, there is a global constraint that only one initial token is present.

Petri Nets for Control Specification

Complexity of systems often is caused by the interaction of a discrete controller with a physical plant. The physics of the plant can be well modeled by differential equations, and the controller behavior can be specified by a discrete formalism such as Petri nets. Before implementation, it is important to choose the most appropriate semantics as discussed before. Because process control is deterministic where nondeterminism is typically solved by priorities, we choose priority Petri nets as our implementation. Moreover, because typically control actions that are enabled are executed as well, we implement the maximum firing semantics for normal priority Petri nets.

Petri nets are discrete and unless the semantics are augmented, no time is associated with their behavior. Only the ordering of state changes is given. To interact with the continuous time formalism that models plant behavior, an external enable signal is associated with transitions. This enables or disables transitions based on continuous plant conditions. In turn, the presence or absence of a token in a place is communicated back to the continuous model part.

For example, the Petri net in Fig. 2 specifies the state change behavior of an electrical diode. When a diode is *on*, state p_{on}, it enforces a small voltage drop irrespective of the current that flows through it, as long as it is positive. If the current would become negative, the diode immediately switches *off*, state p_{off}. Therefore, the Petri net has maximum firing semantics associated

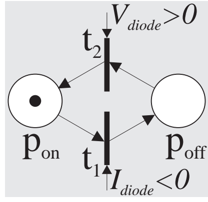

Figure 2: A Petri net specifying a diode.

with it. When *off*, irrespective of the negative value of the voltage, the diode does not conduct.

IMPLEMENTATION

The goal is to describe the components of a Petri net by *local equations* and to generate executable code for a complete Petri net by the equation sorting algorithm of *object-oriented modeling* (Elmqvist 1978): (1) *Collect* the equations of all components of the system. (2) Add trivial equations of the form $a = b$ for every component *connection*. (3) *Sort* all equations into an explicit forward sequence.

This equation based implementation has the advantage that a discrete formalism can be easily combined with a formalism based on continuous differential-algebraic equations. All parts of a system are described by equations and during sorting no distinction is made between the two types of equations which automatically leads to proper synchronization between discrete and continuous parts of a system. The only difference occurs during continuous time integration because boolean equations are evaluated at event instants only.

LICS (Elmqvist 1985) and Omola (Andersson 1994) use local models with equations for Grafcet elements, a discrete modeling formalism related to Petri nets. However, LICS did not handle interaction with continuous models and Omola relied on special language constructs for the interaction with the continuous part.

Basic Equations

The basics of the implementation are explained by a special place and transition component. On the left-hand side in Fig. 3 a *place* with two input and two output transitions is shown. The state of the place is described by the boolean variable *state* which is *true* if the place is *active* (a token is present) and which is *false* otherwise (no token is present).

With every *connection* two boolean variables s and f are associated. Variable s is used to report the state of the place to all transition components that are directly connected to this place. Based on this information, a transition component decides whether it fires or not. This firing information is reported back to the corresponding places via variables *f*. For example, the place

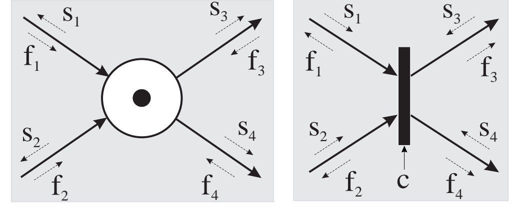

Figure 3: Special place and transition.

of Fig. 3 is described by the following boolean equations:

$$\begin{aligned} s_1 &= state^i \\ s_2 &= state^i \textbf{ or } f_1 \\ s_3 &= state^i \\ s_4 &= state^i \textbf{ and not } f_3 \\ state^{i+1} &= (state^i \textbf{ and not } (f_3 \textbf{ or } f_4)) \textbf{ or } f_1 \textbf{ or } f_2 \end{aligned}$$

Since a place can hold at most one token, it is necessary to synchronize the transition connections to ensure that at most one token is moved to the place and at most one token is removed from it. This is achieved by completely evaluating one transition before the next one is investigated. The state $state^i$ of the place at the current event i is first reported to the first input transition via s_1. Only after the firing condition f_1 is reported back, the state of the place is reported to s_2 taking f_1 into consideration. If, e.g., $state^i$ = false and f_1 = true then s_2 = true which prevents the second transition from firing. Otherwise, the place may receive tokens from both transitions, violating its capacity 1 constraint. In a similar way the output transitions 3 and 4 are handled to ensure that only one of the two fires by giving higher priority to transition 3. Finally, the new state $state^{i+1}$ is computed given the four firing conditions f_1, f_2, f_3, f_4. Note, that during sorting of the equations, the new state $state^{i+1}$ and the actual state $state^i$ are treated as two different variables to avoid algebraic loops.

For the *transition* with two input and two output connections shown on the right-hand side in Fig. 3 the same two variables are associated with every connection. Based on the reported states of the connected places (s_1, s_2, s_3, s_4) and the external boolean condition c, the transition component is able to determine the value of the firing variables (f_1, f_2, f_3, f_4):

$$\begin{aligned} &fire = c \textbf{ and } s_1 \textbf{ and } s_2 \textbf{ and not } (s_3 \textbf{ or } s_4) \\ &f_1 = f_2 = f_3 = f_4 = fire \end{aligned}$$

The transition fires if the external condition is true and all input places and none of the output places have a token.

An example is given in Fig. 4, where the state of a place i is defined by boolean variable p_i. The new

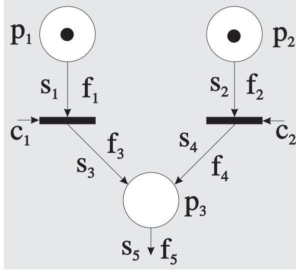

Figure 4: Example of a Petri net.

state is characterized by **new**(p_i). A translator of an object-oriented modeling system collects the described equations of all components, sorts the overall system of equations, removes trivial equations (e.g., $s_1 = p_1$ and $f_3 = f_1$) and generates the following code:

loop
f_1 := c_1 **and** p_1 **and not** p_3
s_4 := p_3 **or** f_1
f_2 := c_2 **and** p_2 **and not** s_4
new(p_1) := p_1 **and not** f_1
new(p_2) := p_2 **and not** f_2
new(p_3) := f_1 **or** f_2 **or** p_3 **and not** f_6
...
break if (**new**(p_1)==p_1 **and** **new**(p_2)==p_2 **and** **new**(p_3)==p_3)
p_1 := **new**(p_1)
p_2 := **new**(p_2)
p_3 := **new**(p_3)
end loop

After every iteration of the loop, the actual states of the places are updated with the new states. The iteration finishes if no further state changes occur.

Algebraic Loops

Nondeterminism of transitions is resolved by explicitly ordering places that make their state available to transitions on outgoing connections. However, there is also an explicit order in which places make their state available on incoming connections, and, therefore, in pathological cases the overall system of equations cannot be sorted into a strict forward sequence, i.e., a system of algebraic equations is present. Because each place breaks algebraic loops by their **new** state variable, problems only occur if at least two input connections of one place are connected through a transition to at least two output connections of one, possibly the same, place, e.g., if two self-loops are present. A Modelica translator reports an error because an algebraic loop is present containing unknown *Boolean* variables and the translator is not able so solve such equations.

Realization with Modelica

For the implementation of the Petri net formalism with the Modelica language, first the underlying classes have to be identified along with the object interfaces. Two basic model classes are used: (i) model *Place* to describe places and (ii) model *Transition* to describe transitions. Places and transitions may have any number of input and output connections. Therefore, the connection point of these components has to be described by a vector of connectors containing variables *s* and *f* defined in the previous section. This would lead to the situation that connectors are completely equivalent, and, therefore, that places can be connected to places and transitions to transitions. To prevent this, two types of connectors are used which have different variable names. Because Modelica requires names and types of connected connector components to be identical, this feature ensures that only places can be connected to transitions and vice versa.

The Modelica Petri net library is structured as:

```
package NormalPetriNet
  connector FireCut
    Boolean state "State of connected place";
    Boolean fire  "True, if transition fires";
  end FireCut;

  connector SetCut
    Boolean state "State of connected place";
    Boolean set   "True, if transition fires";
  end SetCut;

  model Place
    ...
  model Transition
    ...
end NormalPetriNet;
```

The *FireCut* connectors are used to connect places to transitions, whereas the *SetCut* connectors are used to connect transitions to places. The *state* variable of the connector classes corresponds to variable *s*, whereas *fire* and *set* correspond to variable *f*. The Place and Transition model classes contain a vector of *SetCut* connectors and a vector of *FireCut* connectors. The Place and Transition models can now be described as:

```
model Place
    constant Integer nSet  = 1;
    constant Integer nFire = 1;
    SetCut           in [nSet];
    FireCut          out[nFire];
    Boolean          state(start=false);
    Boolean          dummy1, dummy2;
  equation
    // Set new state for next iteration
       new(state) = AnyTrue(in.set) or state
```

```
                    and not AnyTrue(out.fire);

      // Report state to input transitions
         [in.state; dummy1] =
            [state; in.state or in.set];

      // Report state to output transitions
         [out.state; dummy2] =
            [state; out.state and not out.fire];
  end Place;

  model Transition
      input Boolean condition;
      constant Integer  nSet  = 1;
      constant Integer  nFire = 1;
      FireCut            in [nFire];
      SetCut             out[nSet];
    equation
      fire = condition and AllTrue(in.state)
                 and not AnyTrue(out.state);
      in.fire = fire;
      out.set = fire;
  end Transition;
```

These models use the following utility functions:

```
  function AnyTrue "Forms logical OR of vector"
      output Boolean result;
      input  Boolean in[];
    algorithm
      result := false;
      for i in 1:size(in,1) loop
        result := result or in[i];
      end for;
  end;

  function AllTrue "Forms logical AND of vector"
      output Boolean result;
      input  Boolean in[];
    algorithm
      result := true;
      for i in 1:size(in,1) loop
        result := result and in[i];
      end for;
  end;
```

The Place and Transition models are formulated using matrix equations. These equations are equivalent to the special cases discussed previously. The used utility functions *AnyTrue* and *AllTrue* form the logical *or* and the logical *and* of all the elements of a Boolean vector, respectively. If a function is "small", which is the case for the two auxiliary functions, a Modelica translator inlines the function in order to avoid function call overhead.

A HYBRID SYSTEM EXAMPLE

This section presents an example based on (Seebeck 1998), with a less complex discrete part and a more involved continuous part of the overall system. Modeling

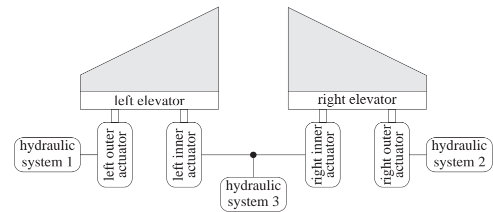

Figure 5: Elevator system.

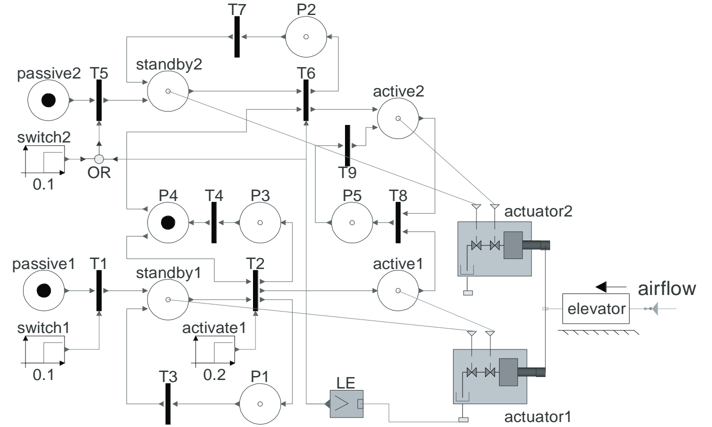

Figure 6: Top-level diagram of elevator control.

and simulation are performed with Dymola (Elmqvist, Brück, & Otter 1996) using Dymola's hydraulic library (Beater 1997) and the Petri net library described in the previous sections.

Given the critical nature of their operation, aircraft are equipped with redundant components to perform control tasks. For example, the elevator system, located at the trailing edge of the horizontal stabilizer, may consist of four actuators connected to three hydraulic subsystems, Fig. 5. Each mechanical elevator is positioned by two hydraulical actuators. During normal operation, each of these elevators is controlled by their outer actuator. In case of failure, this actuator is deactivated and a redundant actuator takes over.

To study the dynamics of attitude control in case of actuator failure, the continuous dynamic behavior of the actuators and elevator are modeled. The elevator is represented as a mass with stick-slip friction and constant force, e.g., because of interaction with airflow (see Fig. 6). The cylinder that drives the elevator (see Fig. 7) is controlled by opening a servo valve, modeled by second order dynamics. A spool valve allows the pressure of the servo valve to be passed to the cylinder, depending on whether the actuator is active or not.

The redundancy management of the three actuators is modeled by a Petri net. Each of the actuators can be in one of three possible states, (i) *active*, the control

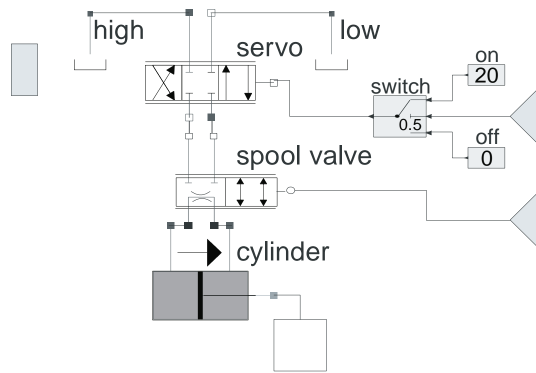

Figure 7: The hydraulics of an actuator.

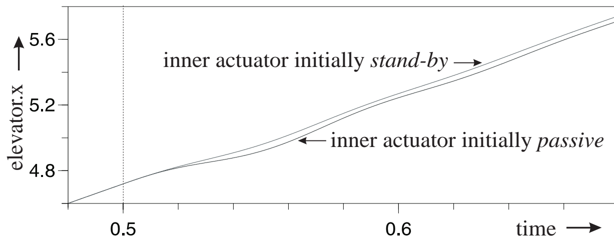

Figure 8: Two cases of abrupt pressure drop.

signal positions the servo valve and the corresponding pressure drives the cylinder piston, (ii) *stand-by*, the control signal positions the servo valve but the corresponding pressure is not passed to the cylinder, (iii) *passive*, the control signal does not position the servo valve and the pressure is not passed to the cylinder. This is shown in Fig. 6, which also shows that when the inner actuator (*actuator2*) is activated, the outer one (*actuator1*) is deactivated by transition T_8.

Fig. 8 shows a simulation where at time 0.1 s, the outer actuator is switched to its *active* state and forces a control pressure on the actuator cylinder. The elevator starts to move towards its setpoint. The inner redundant actuator is switched to its *stand-by* state to allow quick response in case of failure. At $t = 0.5$ s, the pressure in the outer actuator drops abruptly. This continuous time signal is coupled to the redundancy management and because too low, it causes transition T_6 to fire. As a result, the inner actuator is activated which, in turn, causes the outer actuator to be deactivated. If the inner actuator were not in stand-by but in its passive state, there is a delay in response when switched to active because of the response time of the servo valve. This is shown by the lower trace in Fig. 8.

CONCLUSIONS

This paper implements Petri net components, places and transitions, in terms of local continuous constraint equations. This allows for their use in the object-oriented, unifying, physical systems modeling language Modelica. The local equations that describe the discrete system behavior are compiled into a globel set of differential and algebraic equations. Therefore, this implementation seamlessly integrates discrete and continuous model behavior, e.g., as required for comprehensive modeling of embedded control systems.

REFERENCES

Alur, R.; Courcoubetis, C.; Henzinger, T. A.; and Ho, P.-H. 1993. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Lecture Notes in Computer Science*, volume 736. Springer-Verlag. 209–229.

Andersson, M. 1994. *Object-Oriented Modeling and Simulation of Hybrid Systems.* PhD dissertation, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Beater, P. 1997. *hylib — Library of Hydraulic Components for Use with Dymola.* Dynasim AB, Research Park Ideon, Lund.

Elmqvist, H.; Brück, D.; and Otter, M. 1996. *Dymola — User's Manual.* Dynasim AB, Research Park Ideon, Lund.

Elmqvist, H. 1978. *A Structured Model Language for Large Continuous Systems.* PhD dissertation, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Elmqvist, H. 1985. *LICS — Language for Implementation of Control Systems.* Technical Report CODEN:LUTFD2/(TFRT-3179)/1-130/(1985), Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Kohavi, Z. 1978. *Switching and Finite Automata Theory.* New York: McGraw-Hill, Inc.

Kowalewski, S.; Stursberg, O.; Fritz, M.; Graf, H.; Hoffmann, I.; Preußig, J.; Simon, S.; Remelhe, M.; and Treseler, H. 1998. A case study in tool-aided analysis of discretely controlled continuous systems: The two tanks problem. In *Hybrid Systems V.* Springer-Verlag, in review.

Modelica. 1997. A unified object-oriented language for physical systems modeling. Modelica homepage: http://www.Dynasim.se/Modelica/.

Mosterman, P. J.; Biswas, G.; and Sztipanovits, J. 1997. Hybrid modeling and verification of embedded control systems. In *Proceedings of the* 7th *IFAC CACSD '97 Symposium*, 21–26.

Murata, T. 1989. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* 77(4):541–580.

Seebeck, J. 1998. *Modellierung der Redundanzverwaltung von Flugzeugen am Beispiel des ATD durch Petrinetze und Umsetzung der Schaltlogik in C-Code zur Simulationssteuerung.* Diplomarbeit, Arbeitsbereich Flugzeugsystemtechnik, Technische Universität Hamburg-Harburg.

Starke, P. H. 1990. *Analyse von Petri-Netz-Modellen.* Stuttgart: B.G. Teubner. ISBN 3-519-02244-3.