Combining Data-Flows and Petri Nets for Cyber-Physical Systems Specification

Fernando Pereira, Luis Gomes

Combining Data-Flows and Petri Nets for Cyber-Physical Systems Specification

Fernando Pereira[1,2,3], Luís Gomes [1,3]

[1] Universidade Nova de Lisboa - Faculdade de Ciências e Tecnologia – Portugal
[2] ISEL, Instituto Superior de Engenharia de Lisboa- Portugal
[3] UNINOVA - CTS – Portugal
fjp@deea.isel.ipl.pt, lugo@fct.unl.pt

Abstract. This paper proposes a new modeling formalism for the specification of cyber-physical systems, combining the functionality offered by Petri nets and synchronous data flows. Petri nets have been traditionally used to model the behavior of reactive systems, whose state evolves depending on the interaction with external events. On the opposite, data-flow formalisms have been used predominantly to describe data-driven systems that produce output data through mathematical transformations applied to input signals. The proposed formalism covers both kinds of problems, offering support for the design of mixed systems containing linear control and signal processing operations along with event driven elements. Model composition using multiple components communicating through input and output signals and events, enable the implementation of distributed cyber-physical systems. The new formalism and the respective execution semantics are presented, with special attention to the bidirectional interaction between Petri net elements and data-flow nodes.

Keywords: Cyber-physical systems, Embedded systems, Petri nets, Data-flow.

1. Introduction

Cyber-physical systems assume a growing importance in all fields of the modern world, with many applications that include industrial machines, home appliances, entertainment systems and gadgets. The fast dissemination of the Internet and the wide availability of inexpensive networking technology, brought Internet connectivity to the recent generations of embedded devices, contributing to the birth of the Internet of Things. This evolution enabled the development of new applications and services, including access to automatic payment systems, connection to social media platforms and solutions based on distributed networks of remote devices, like smart grids, city traffic control systems, in-vehicle systems and wireless sensor networks.

These advances opened a gap for novel development solutions adapted to the new design challenges. Model based development formalisms, from which Petri nets can be highlighted, promise to answer these questions: offering high level design concepts that hide the low level platform details, contribute to accelerate development time, minimize the probability of coding errors and reduce time-to-market.

From the existing Petri net [1] based tools, the IOPT tools framework [2] and the underlying Petri net class [3] have been designed for embedded system controller

development. However, the IOPT tools currently do not offer component based model composition and lack support for complex data manipulation operations. The formalism proposed in this paper was specified to address both problems: provide support for model composition and introduce a complementary formalism to deal with data driven problems. These extensions simplify the modeling of mixed linear/event-driven controllers and systems that make extensive use of mathematical data operations. Model composition bring advantages to the modeling of complex systems, support component re-use and allow the creation of component libraries.

2. Relationship to Cyber-Physical Systems

This paper proposes a new development formalism designed to support the development of embedded systems and cyber-physical systems. The proposed formalism supports model composition using components. The external interface of the components is defined by input and output signals and events.

The formalism supports both centralized and distributed implementations, where each component can be located on remote Internet locations. A complete Cyber-physical system may be specified as a single model composed by multiple components and the components may be executed at different locations. A communication protocol for remote control, monitoring and debug of embedded controllers [12][13] satisfies the requirements to support the component interactions.

3. Related Work

The formalism proposed in this paper is the result of previous work around the IOPT tools framework [2] and the IOPT Petri net class [3]. Other Petri net based tools that support the modeling of embedded systems have been proposed by different authors, from which the NCES [4], SNS [5], SIPN [6] and CPN [7] should be mentioned.

However, the traditional Petri net based formalisms have primarily focused on the reactive part of the controllers and do not offer good support to solve data-driven problems. Petri net classes have usually relied on text based mathematical expressions that are enabled when specific places are marked or transitions fire. In some classes these expressions may call procedures written using standard programming languages, as Java and Standard-ML. In contrast, this paper proposes a hybrid formalism that employs Petri nets and data-flows. Petri nets provide good modeling capabilities to design reactive controllers. Data-flows offer advantages to solve data driven problems, including digital signal processing, linear control of systems and support systems that require many data transformation operations.

Some Petri net based dialects, as the NCES[4] and SNS[5], support model composition based on components communicating through signals and events. One of the main applications of this capability is the study of the interaction between controllers and controlled systems (called plants). However, the plant models often require intensive data processing and are not adequately modeled using Petri nets. The

new formalism offers both model composition and data processing capabilities and is well adapted to model the plants and the controllers.

Other formalisms, as the synchronous data flows [8][9], IEC61499 [10] and Matlab Simulink [11] offer support for data driven problems, but are not centered on Petri nets and do not benefit from all the available Petri net model-checking tools.

4. Graphical Representation

A DSPnet model (Data-flow, Signals and Petri nets), is a directed graph composed by five types of nodes: Petri net places, Petri net transitions, input/output signals, inputs/output events and data-flow operations. The nodes may connected using two types of arcs: normal-arcs and read-arcs. Normal arcs correspond to the traditional Petri net arcs. Read arcs are used to transmit data between graph nodes and may be used to connect signals, data-flow operations and Petri net nodes.

Figure 1 presents an example model. Petri net places as drawn as yellow circles and transitions as cyan rectangles. Input signals are presented as green circles and input events as green diamonds. In the same way, output signals and events are drawn as red circles and red diamonds. The data-flow nodes that perform mathematical operations, are represented by gray trapezoids, with green anchors to attach input arcs and red anchors to attach output arcs. Normal Petri net arcs are drawn as solid black and read arcs are drawn as dashed blue.

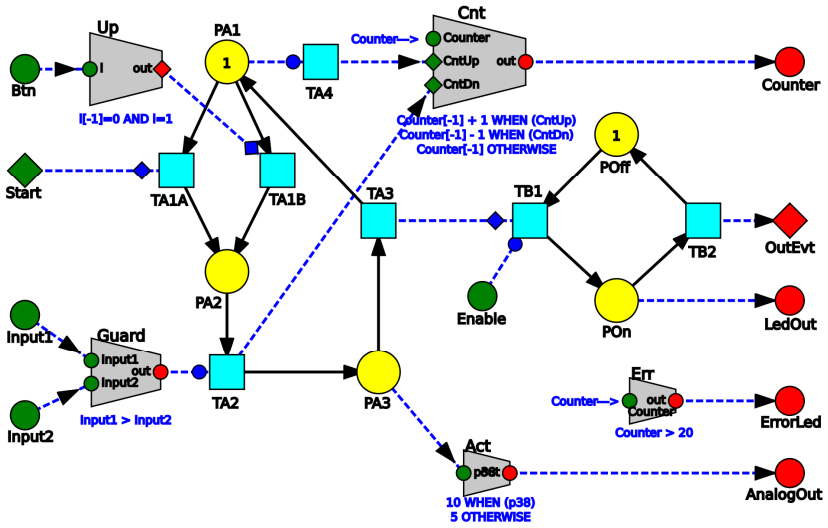

Fig. 1: Example model

Data-flow nodes, called operations, employ mathematical expressions to calculate new values. Each operation has a set of input and output anchors that may be connected to other nodes using read arcs. Each output anchor holds one expression,

used to compute the corresponding value. Input anchors are used as operands in the mathematical expressions. Each anchor has an associated name and data-type. Computation time is considered instantaneous, with no propagation delays, implementing a synchronous data-flow behavior. As a consequence, signal loops, where the output of an operation is directly or indirectly feedback to an input, are forbidden. When a signal loop is desired, expressions may employ a delay operator «[-n]», to refer the value of the feedback input from previous execution steps. In the example on figure 1, the *Cnt* operation employs the delay operator to read the previous value of the *Counter* output signal («Counter[-1]»).

Input and output signals and events are used to define the external interface of the model and establish the communication with the external world. Input signals may be connected to sensors or user interface items like buttons and switches. Output signals may be connected to LEDs, power-electronics, relays or mechanical actuators. When used as components in distributed cyber-physical systems, the input and output signals and events may be used to communicate with other sub-systems. Each signal has an associated data-type. Available data-types are Boolean, integer ranges and fixed-point ranges.

Events represent instantaneous actions that may (or not) happen on any execution step. Input events are triggered by external sources, but internal or output events may be triggered by the firing of transitions or as the result of data-flow computations. For example, a data-flow operation may detect the crossing of a predefined threshold on an input signal and produce an event. In this case, the data-type associated with the operation output must be defined as an «event». Events may also be used as input for data-flow operations, treated as Boolean values inside expressions. In figure 1, the *Up* operation produces an event by detecting an up edge on the input signal *Btn,* and the *Cnt* operation interprets two events to implement and up-down counter.

The Petri net places and transitions are used to specify the control logic of the cyber-physical systems, whose state evolves reacting to external events and changes in signals and data-flow elements. A maximal step execution semantics is employed, where all enabled transitions must always fire in the next execution step. Conflicts between transitions, when multiple enabled transitions compete for the same place tokens, are resolved using priorities.

Transition firing is inhibited using guards and events. A guard condition is defined by a read arc starting on a Boolean signal or data-flow operation, and the transition can only fire when the corresponding value holds true. Read arcs starting on events also prevent transition firing. In this case, the arcs may originate on external events, data-flow nodes producing events or another transition. When the arc originates on a transition, it is equivalent to a synchronous channel. In the same way, when a read arc starts on a place, it creates a guard condition that is equivalent to a test arc: transition firing depends of the place marking but no tokens are consumed and no conflicts with other transitions are raised.

Figure 1 presents several examples of guard conditions and transition input events. Arcs terminating on a transition, with a solid dot near the end, denote a guard condition and arcs ending with a diamond denote an event. Transitions *TA1*, *TA2* and *TB1* are conditioned by input events, while transitions *TA2*, *TA4* and *TB1* have guard conditions. In particular, transition *TA4* has a test arc from place *PA1* and transitions *TA3* and *TB1* are connected using a synchronous channel.

In the opposite direction, place marking and events generated by transition firing, may be used to influence the data-flow calculations. Read arcs starting on places read the number of place tokens. Read arcs starting on a transitions transmit events. In figure 1 the value of *AnalogOut* is calculated from place PA3, with value 10 when the place is marked and 5 otherwise. The value of the *Counter* output is defined by the *Cnt* operation. This operation implements an up/down counter controlled by events triggered by the TA2 and TA4 transitions.

Arcs may be visualized using two formats: in addition to the usual graphical representation, a symbolic format contributes to minimize clutter and improve readability. When this format is chosen, instead of drawing an arrow, the identifier of the source node is presented near the target node. Figure 1 includes two symbolic arcs. Both arcs start at the *Counter* output, ending at the *Cnt* and *Err* operations.

Input and output signals and events may be connected directly to Petri net nodes. In figure 1, transition *TB2* generates an output event and the marking on place *POn* defines the value of output *LedOut*.

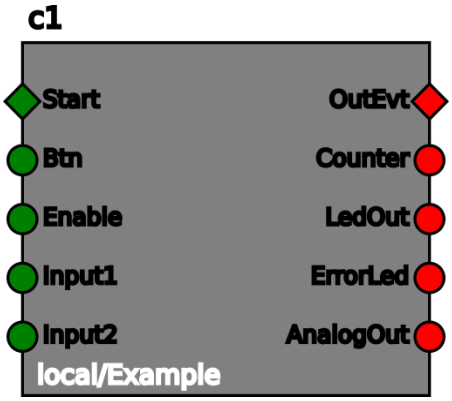

Fig. 2: External interface

A typical approach to the design of complex systems, is the assembly of multiple components that implement individual sub-systems. This way, the same component sub-systems may be re-used on multiple projects. Existing models might be used as components to create higher level solutions. Figure 2 displays the external interface of the model presented on Figure 1. The interface has a set of anchors corresponding to the input and output signals and events defined in the original model, which may be directly used in other models using read arcs. Distributed implementations may place the components on different hardware platforms, or even on remote network locations. Centralized implementations may start with the assembly of a flat model containing the contents of all components.

5. Formal Definition

A DSPnet is a directed graph composed by three parts: an external interface composed by input and output signals and events, a state control part consisting of a low-level Petri net and a data-processing part composed by data-flow operations and internal signals.

Definition 1: A system specified by a DSPnet is described as a tupple $DSPnet = (P, T, S, E, O, A, R, m_0, s_0, w, pt, ex, st, ot)$ satisfying the following requirements:

1) P is a finite set of places
2) T is a finite set of transitions
3) S is a finite set of signals
4) E is a finite set of events
5) O is a finite set of data-flow nodes, called operations
6) $P \cup T \cup S \cup E \cup O = \phi$
7) A is a finite set of normal Petri net arcs with $A \subseteq (P \times T) \cup (T \times P)$
8) R is finite set of read arcs with
   $R \subseteq (S \times S) \cup (S \times O) \cup (S \times T) \cup (O \times S) \cup (O \times O) \cup (O \times T) \cup (P \times T) \cup (O \times E) \cup (E \times O) \cup (E \times E) \cup (E \times T) \cup (T \times T)$
9) $\forall s \in S, \#\{(x \times s) | (x \times s) \in R\} \leq 1$ (signals have no more than one input arc)
10) $\forall e \in E, \#\{(x \times e) | (x \times e) \in R\} \leq 1$ (events have no more than one input arc)
11) m_0 is the initial place-marking function with mapping $m_0: P \rightarrow N_0$
12) s_0 is the initial signal values partial function with mapping $s_0: S \rightarrow \{N_0, -\}$
13) w is the normal-arcs weight function with mapping $w: A \rightarrow N_0$
14) pt is the transition priority function with mapping $pt: T \rightarrow N_0$
15) ex is a function applying operations to mathematical expressions (where all expression non-literal operands are the source of the operation input arcs)
   $ex : O \rightarrow exp$, where $\forall nlop \in exp(O), nlop \in \{x | (x,O) \in R\}$
16) st is a signal type function with mapping $st: S \rightarrow t, t \in \{Boolean, Range\}$
17) ot is an operation result type function with mapping ot: $O \rightarrow t, t \in \{Boolean, Range, Event\}$

The external interface of a system defined by a DSPnet is composed by a set of input signals, output signals, input events and output events that is a subset of the system signals and events.

Definition 2: The external interface of system specified by a DSPnet is a tuple $EIF = (IE, IS, OE, OS)$ satisfying the following requirement:

1) $IE \subseteq E$
2) $IS \subseteq S$
3) $OE \subseteq E$
4) $OS \subseteq S$
5) $IE \cap IS \cap OE \cap OS = \phi$
6) $\forall s \in IS, \{(x \times s) | (x \times s) \in R\} = \phi$ (input signals have no input driver arcs)
7) $\forall e \in IE, \{(x \times e) | (x \times e) \in R\} = \phi$ (input signals have no input driver arcs)

A system defined by a DSPnet can be decomposed in two parts, the state control logic and the data-processing part:

Definition 3: The state control logic part of a *DSPnet* is a low level Petri net defined by a tuple $PN = (P, T, A, m_0, w, tp, R^-)$ where:

1) P is the DSPnet set of places
2) T is the DSPnet set of transitions
3) A is the DSPnet set of normal arcs
4) m_0 is the DSPnet initial marking mapping $m_0: P \rightarrow N_0$
5) w is the DSPnet arc weight mapping w: $A \rightarrow N_0$
6) tp is the DSPnet transition priority mapping: $T \rightarrow N_0$
7) R_P is a subset of the DSPNet set of read arcs such as $R_P \subseteq R \wedge R^- \square (P \times T) \cup (T \times T)$
   (test arcs and synchronous channels)

Definition 4: The data processing part of a *DSPnet* is a synchronous data-flow $SDF = (O, S, E, R^+, s_0, ex, st, ot)$ where:

1) O is the DSPnet set of data-flow operation nodes
2) S is the DSPnet set of signals
2) E is the DSPnet set of events
4) R_D is a subset of the DSPnet read arcs $R_D = R - R_P$
5) s_0 is the DSPnet initial signal values partial function $s_0: S \rightarrow \{N_0, -\}$
6) *ex* is the DSPnet operation expressions function
7) st is the DSPnet signal types function
8) *ot* is the DSPnet operation results type function

The *ex* mathematical expressions described in Definition 1, produce integer range values, Boolean values, or events and can include the following items:

- Literal operands: decimal values or hexadecimal values starting with the «0x» prefix
- Variable operands corresponding to the graph nodes directly connected through input arcs
- The arithmetic operators +, -, *, / and *MOD,* plus the unary operator -
- The comparison operators =, <>, <, <=, > and >=
- The logical operators AND, OR, XOR and the unary operator NOT
- The «bit» operators and (&), or (|) and not (*!*) in addition to shift left («) and shift right (»), - Sub-expressions inside parentheses *(* and *)*
- The delay operator ([-n]) in association with variable operands, to refer past values from previous execution steps
- The array index operator ([+i]) associated with tables of constant values stored in operation nodes, to implement mathematical functions based on tables of values
- The conditional operators *WHEN* and *OTHERWISE* to build if/case constructs

In Figure 1, the *Cnt* up-down counter was implemented using the expression:

```
Counter =     Counter[-1] +1 WHEN CntUp,
         Counter[-1] -1 WHEN CntDn,
         Counter[-1] OTHERWISE
```

Meaning that the new value of the *Counter* output signal is calculated as:

a) The previous value of *Counter* plus 1 if the *CntUp* event happens

b) The previous value of *Counter* minus 1 if *CntDn* happens
c) The value remains unchanged when none of these events occur

6. Execution Semantics

The execution semantics of the proposed modeling formalism inherits principles from both synchronous data flows [9] and low level Petri nets [1], in particular from the IOPT Petri net class [3].

The evolution of the system state is performed in quantum steps, called execution steps that typically occur at a certain frequency, with variable of fixed time intervals. Each execution step is considered instantaneous, but may be divided in a finite number of micro-steps.

As presented in definition 3, the state control logic part of a model consists on a low level Petri net. As a consequence, the main aspect defining the evolution of a Petri net are the firing rules associated with transitions: in order to fire, a transition must be simultaneously enabled and ready.

Definition 5: A transition is enabled when all input places, connected through normal Petri net arcs, hold a number of tokens that is equal or more than the respective arc weights. For a transition t: $\forall(p \in P \mid (p,t) \in A),\ m(p) > w(p,t)$

Definition 6: A transition guard condition is defined by a read arc ending in the transition, originating on a node containing a Boolean value or a range value, evaluated as true when different from zero.

Definition 7: A transition input event is defined by a read arc ending in the transition, originating on an event, a transition or an operation producing a result of type event.

Definition 8: A transition is ready when all guard conditions and input events are true

Definition 9: Maximal step execution semantics - all transitions simultaneously enabled and ready are forced to fire on the next execution step.

Definition 10: Conflict – Two or more transitions are in conflict when all of them are simultaneously enabled, but the number of tokens on the shared input places is not enough to fire all of them.

Definition 11: Conflict resolution – Conflicts between transitions are solved using priorities. Priority criteria is composed by: 1) execution micro-step, 2) transition priority and 3) transition unique identifier.

A read arc starting on a transition propagates events produced when the transition fires, called transition output events. These events can be forwarded to other transitions or used by data-flow operations. For example, a read arc directly connecting two transitions creates a synchronous channel, where the source transition can be viewed as a master and the target as a slave. Synchronous channels are typically used to synchronize transitions located inside different components. Systems with multiple components may contain large chains of master-slave transitions.

However, as both the master and slave transitions must fire on the same execution step (when enabled and ready), the execution semantics rules must ensure that the firing of the master transitions is evaluated before evaluating the slave transitions. Any data-flow operations that depend on events produced by the masters must also be calculated before evaluating the slave transitions. This problem lead to the concept of

micro-steps, that define a precise sequence of evaluation. All DSP-net nodes are associated with a micro-step number, including transitions and data-flow operations.

Definition 12: Micro-step assignment:

1) Nodes with no input read arcs are assigned to micro-step 1

2) Nodes with input read arcs are assigned a micro-step number corresponding to the maximum micro-step associated with these input read arcs, according to the following rules:

a) Read arcs used in inside mathematical expressions in association with the delay operator «[-n]», are assigned to micro-step 1, as the expression uses values stored from previous executions steps.

b) Read arcs starting on a transition, propagating transition output events, are assigned a micro-step number equal to the transition micro-step plus 1.

c) Read arcs starting on non-transition nodes, are assigned the same micro-step number as the source node.

In the same way as transition dependencies lead to the concept of micro-steps, the existence of dependencies between data-flow operations evaluated in the same micro-step, require the definition of another sequencing mechanism to fine-tune the order of operations evaluation, called nano-steps.

Definition 13: Nano-step assignment:

1) Nodes with no input read arcs are assigned to nano-step 1

2) Nodes with input read arcs are assigned a nano-step number corresponding to 1 the maximum nano-step associated with these input read arcs, according to the following rules:

a) Read arcs used in inside mathematical expressions in association with the delay operator «[-n]», are assigned to nano-step 1

b) Read arcs starting on nodes from past micro-steps, including all places and transitions, are assigned nano-step 1

c) Read arcs starting on nodes from the same micro-step, are assigned 1 plus the source node nano-step number

Lemma 1: Any data-flow operation nodes sharing the same micro-step and nano-step numbers can be evaluated by any execution order, or executed in parallel.

It is important to notice that micro-step and nano-step numbers are just used to sort the operation/transition nodes and define a deterministic sequence of evaluation, but does not imply any clocking scheme for hardware or software implementations.

In order to be syntactically correct, the number of micro-steps and nano-steps must be finite and a DSPnet cannot exhibit loops where nodes with higher micro/nano step numbers are used by nodes with lower numbers. This would contradict the rules on definitions 12 and 13, even if the loops occur across nodes located on different components. To avoid loops, the external interface of the components must include information about the dependencies between input and output signals and events. When models of the individual components are available, a flat model containing all the nodes from all components should be constructed and analyzed.

In conclusion, the following algorithm may be used to implement an execution step of a system described by a DSPnet:

```
read input-signals, input-events
```

```
for-each place do
   avail-marking[place] = marking[place]
   add_marking[place] = 0
done
for micro-step = 1 to n-micro-steps do
   for nano-step = 1 to n-nano-steps[micro-step] do
      execute data-flow-operations[micro-step][nano-step]
   done
   for-each transition[micro-step] (sort by priority,identifier)
   do
     if transition-is-enabled and transition-is-ready
     then
         for-each input-place[transition] do
             avail-marking[place] = marking[place] - arc-weight
          done
         for-each output-place[transition] do
             add-marking[place] = add-marking[place] + arc-weight
          done
    end if
done
for-each place do
   marking[place] = avail-marking[place] + add_marking[place]
done
for-each signal do
   if use-delay-operators(signal)    then shift-registered-
values(signal)
done
write output-signals, output-events
```

7. Conclusion and Future Work

This paper proposes a new modeling formalism for the design and specification of cyber-physical systems, addressing both reactive, data-driven and mixed systems, employing a hybrid language that joins Petri nets and data-flows. Model composition based on components that communicate using input and output signals and events, simplify the design of distributed solutions based on networks of remote components located on the internet.

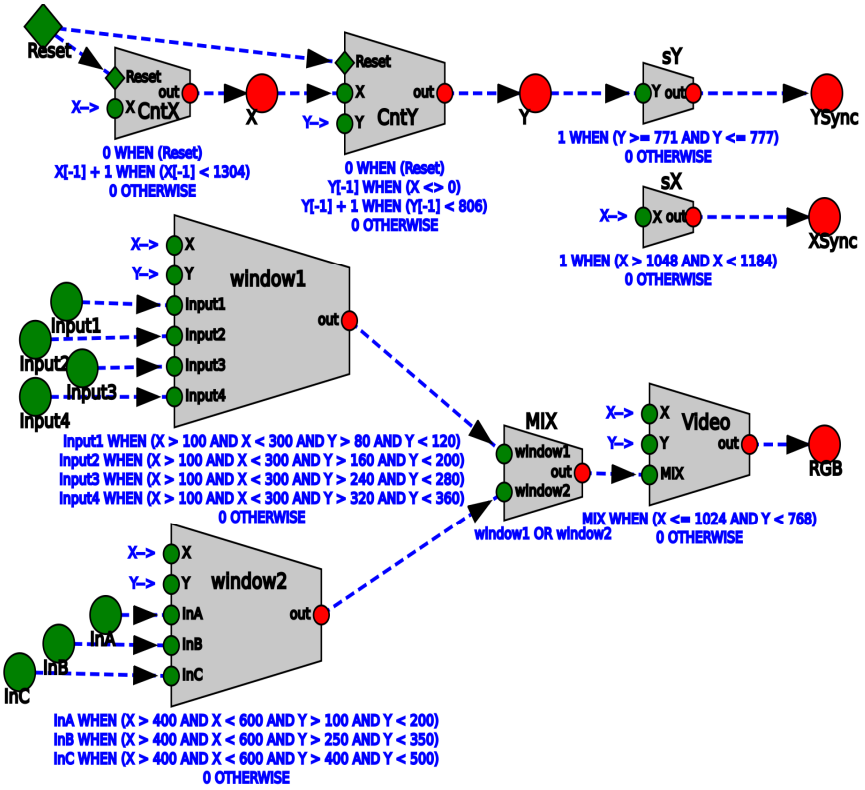

Fig. 3: Video image generator model

Relative to previous solutions, the new formalism offers many advantages to model data-driven systems that do not employ traditional state-machines, with advantages over purely Petri net centered based languages. As an example, fig. 3 presents a simple data flow model of a video image generator model. This model may be easily translated to VHDL and implemented on a FPGA device, to generate a video image that presents two windows with rectangular buttons that lighten whenever several input signals are active. A Web based graphical editor for the proposed formalism is currently under development, and all the examples presented in this paper were designed using this tool. The editor implements an algorithm to calculate the micro-step and nano-step number according to the definitions presented in this paper and displays the results. Future work includes the development of an entire tool framework, including a Web based simulator, model-checking tools, automatic code generation tools for software (C) and hardware (VHDL) based platforms. An existing protocol to support distributed implementations and remote debug and monitoring over the internet [12][13] will be ported to the new formalism.

References

[1] Reisig, W. (1985). Petri nets: an introduction; New York, USA: SpringerVerlag New York
[2] Pereira, F.; Moutinho, F; & Gomes, L. (2014). IOPT-Tools - Towards cloud design automation of digital controllers with Petri nets. ICMC'2014 International Conference on Mechatronics and Control. July 03-05 2014, Jinzhou, China
[3] Gomes, L., Moutinho, F., Pereira, F., Ribeiro, J., Costa, A., & Barros, J.-P. (2014) Extending Input-Output Place-Transition Petri nets for Distributed Controller Systems development. ICMC'2014 - International Conference on Mechatronics and Control; 3-5 July 2014, Jinzhou, China, pages 1099-1104
[4] Hanisch, H. M., & Lüder, A. (2000). A signal extension for Petri nets and its use in controller design. Fundamenta informaticae, 41(4), 415-431.
[5] Starke, P.; Roch, S. (2002). Analysing Signal-Net Systems. Humboldt-Universiät at zu Berlin, Institut fur Informatik, September 2002
[6] Frey, G. (2003). Hierarchical design of logic controllers using signal interpreted Petri nets. Proceedings of the IFAC AHDS 2003, Saint-Malo (France), 12, 401-406.
[7] Jensen, K., & Kristensen, L. M. (2015). Colored Petri nets: a graphical language for formal modeling and validation of concurrent systems. Communications of the ACM, 58(6), 61-70.
[8] Lee, E.A., & Messerschmitt, D.G. (1987). Synchronous data flow. in Proceedings of the IEEE, vol.75, no.9, pp.1235-1245, Sept. 1987, doi: 10.1109/PROC.1987.13876
[9] Colaço, J. L., Pagano, B., & Pouzet, M. (2005). A conservative extension of synchronous data-flow with state machines. In Proceedings of the 5th ACM international Conference on Embedded Software (pp. 173-182). ACM.
[10]Vyatkin, V., & Instrument Society of America. (2007). IEC 61499 function blocks for embedded and distributed control systems design (p. o3neida). ISA-Instrumentation, Systems, and Automation Society.
[11] Dabney, J. B., & Harman, T. L. (2004). Mastering Simulink. Pearson.
[12] Pereira, F; Gomes, L. (2013). Minimalist Architecture to Generate Embedded System Web User Interfaces. DoCEIS'13, Technological Innovation for the Internet of Things. Springer Berlin Heidelberg, 2013. 239-249.
[13] Pereira, F.; Melo, A.; Gomes, L. (2015). Remote operation of embedded controllers designed using IOPT Petri-nets. 13th IEEE International Conference on Industrial Informatics; 22-24 July 2015, Cambridge, UK