R. Xu, S. Zhang, D. Liu and D. A. Zaitsev, "Sleptsov net based reliable embedded system design on microcontrollers and FPGAs," 2024 IEEE International Conference on Embedded Software and Systems (ICESS), Wuhan, China, 2024, pp. 1-8, doi: 10.1109/ICESS64277.2024.00011.

Sleptsov net based reliable embedded system design on microcontrollers and FPGAs

Ruiyao Xu
School of Mechano-Electronic Engineering
XIDIAN University
Xi'an 71071, China
0009-0006-0781-7189

Si Zhang
School of Mechano-Electronic Engineering
XIDIAN University
Xi'an 71071, China
0009-0005-7446-6561

Ding Liu, *Member, IEEE*
School of Mechano-Electronic Engineering
XIDIAN University
Xi'an 71071, China
0000-0003-0159-8545, dliu@xidian.edu.cn

Dmitry A. Zaitsev, *Senior Member, IEEE*
School of Computing
The University of Derby
Derby, United Kingdom
0000-0001-5698-7324

***Abstract*—We present a novel methodology of embedded systems design based on Sleptsov net (SN) formalism. We use an SN as a graphical language of concurrent programming and as a modeling language for plant specification and integrated model composition. Among the advantages, we mention the applicability of formal verification techniques for reliable embedded system design, vivid graphical language, and compatibility of the toolchain with different classes of hardware. Composed toolchains are supplied with our ad-hoc tools for embedded systems design on both microcontrollers and FPGAs. We develop a software SN machine for microcontrollers and a generator of Verilog programs for FPGAs. Compared to known SN machine implementations on desktop computers and GPUs, we developed indexed sparse matrix data structure to optimize both memory usage and performance. Benchmarks on real-life Sleptsov net programs show the robustness of the approach with considerably higher performance on FPGA.**

***Index Terms*—Sleptsov net, embedded system, design, verification, microcontroller, FPGA.**

I. Introduction

A Sleptsov net (SN) [1] represents a generalization of a Petri net [2], widely applied for decades to model concurrent systems in manifold application domains [3]. However, applying SNs to embedded system design presents unique challenges, particularly in optimizing resource usage on platforms with constrained memory and processing power, such as microcontrollers and FPGAs. This paper aims to address these challenges, developing SN-based methods that balance computational efficiency with system reliability. We managed to prove the Turing-completeness of an SN [4], [5]; the corresponding universal net [6] has been constructed as a prototype for an SN processor.

This work was supported by the National Natural Science Foundation of China under Grants Nos. 62076189, 62102285, and 62302364, the Complex Systems International Joint Research Center of Shaanxi Province, and Xi'an Theory and Applications of Discrete Event Dynamic Systems International Science and Technology Cooperation Center, China.

There were many attempts to use a Petri net as a concurrent programming language, especially for programmable controllers [7], though because of incremental character of computations, Petri nets are slow, especially in arithmetic operations. Due to the rather attractive vivid graphical representation, numerous efforts were made to load a Petri net graph with constructs of some programming language [8], as the most successful system of such kind, we mention CPN Tools [9].

While general SN computing addresses challenges in high-performance computing (HPC) and artificial intelligence (AI) [10]–[13], embedded system design requires a distinct approach due to its constrained hardware resources and real-time processing requirements. This paper adapts SN principles to these specific needs, that makes SN-based approach feasible and efficient for low-power, resource-limited embedded platforms. Applying formal methods, we prove correctness of a system, that results in reliable ESs, especially required for life-critical applications.

This paper introduces a novel SN-based methodology for embedded system (ES) design, which leverages the unique advantages of SNs, such as graphical representation for concurrent programming and compatibility with various hardware platforms. By incorporating formal verification techniques, this approach ensures system reliability, making it particularly suitable for life-critical applications. Unlike previous work heavily reliant on [14], this study develops dedicated SN implementations for both FPGAs and microcontrollers, emphasizing memory efficiency and computational performance.

In Section II, we introduce general principles of SN based ES design, including concurrent graphical programming, modeling plant, integrated debugging, and verification. In Sections III and IV, we develop specific tool-chains for ES design on MC and FPGA, respectively. Finally, in Section V, we implement benchmarks of our tool-chains on MC and FPGA to acknowledge robustness of our approach.

II. SN BASED DESIGN OF ES

In this section, we outline the SN-based embedded system (ES) design methodology. Starting from defining the fundamental SN structure, we describe its application to embedded control and plant models. Subsequent sections expand on implementing this framework for specific hardware platforms, detailing the tool-chains and resource usage optimization necessary for microcontrollers and FPGAs.

For various kinds of target hardware, for instance: microcontrollers (MCs) or FPGAs, we use a unified graphical language of SNs for ES design following basic principles of SN computing [1]. An SN is well applicable for specifying both ES control and plant since in majority of applications, without loss of generality, we either consider a discrete system or we can use some kind of sampling. We apply a wide spectrum of formal analysis techniques [3], [15] to models of ES control and plant, and then, to integrated models. In particular, we find whether a system is unbounded or contains deadlocks and apply techniques for enforcing liveness and boundedness [16] required for an ideal ES. Finally, a specification of a reliable ES is obtained. Then we use additional tools, described in Sections III and IV, to generate executable code to upload into a target device.

As compared to state-of-the-art works on reliable ES design, static and dynamic techniques of software verification [26] are applied with respect to microcontrollers for pragmatic, programming language centered approaches. We would like to mention the approach benefits for small and non-life-critical applications, where programs are composed via adjusting prototypes with consequent application of software verification tools. Thorough verification inevitably leads to formal model application in an explicit or implicit form that is complicated with the necessity for parsing code with the purposes of building a formal model. In majority of cases the code itself does not contain complete information for successful verification. To mend this deficiency, additional language of annotations is introduced, often in semi-formal syntactic form of special kind of comments or compiler directives. Having a model, hidden inside a verification system, does not allow us to take, to the full extend, advantage of it.

We consider SN based reliable ES design as one close, in its concept, to model-driven development (MDD) paradigm [27]. Though, MDD is heterogeneous and rather sophisticated, using up to dozen different types of diagrams, and resulting in automatic code generation. We use a uniform place-transition net family of models with well-developed formal methods for their verification and finally just enhance the specification with SN extension. Instead of producing code for a certain MC, we implement, on a certain MC, an SN virtual machine optimizing its performance and memory consumption. Thus we avoid both code generation and code parsing that gives us more opportunities to focus on the correctness proof of ESs to ensure their reliability. Benchmarks in Section V testify that our approach provides rather good performance and memory consumption to prove its robustness.

A. SN Definition

Following notation of [4], we introduce an SN as $N = (P, T, A, \mu_0)$, where P (places) and T (transitions) are two parts of nodes, mapping A specifies (multiple) arcs connecting them, and nonnegative numbers μ_0 represent places' marking. The transition firing multiplicity, based on the arc firing multiplicity, are evaluated as follows, respectively

$$c(t) = min_{A(p,t)!=0}(c(p,t)),\ c(p,t) = \mu(p)/A(p,t). \quad (1)$$

Any transition $t \in T$ with $c(t) > 0$ fires at a step, changing the place marking in the following way

$$\mu(p) = \mu(p) - c(t) \cdot A(p,t) + c(t) \cdot A(t,p),\ p \in P. \quad (2)$$

In the sequel, composing SN machines, we implement the above formula in the corresponding software for MC and FPGA in an optimal way based on ad-hoc data structures. Both time and space complexities of the resulting code are taken into consideration and thoroughly evaluated formally and with benchmarks.

Besides regular arcs, we use inhibitor arcs, which enable firing at zero marking, and priority arcs between transitions [1] for brevity of graphical representation of sophisticated algorithms. An inhibitor arc is directed from a place with a transition only and contains a little hole circle on its end; the transition is fireable only in case the place marking equals to zero. A priority arc directed from transition t to transition t' means that priority of t is higher than priority of t'; thus t' never fires if transituon t is fireable.

B. Vending Machine Design

For prototyping ESs, we are using a vending machine and a lift control applications. Let us consider a simple vending machine which schematic image is shown in Fig. 1. It accepts coins of 1 type and sells chocolate bars of 2 types: milk for 2 coins and dark for 3 coins. Pressing the coin return button returns 1 coin. The machine does not keep more than 3 coins.

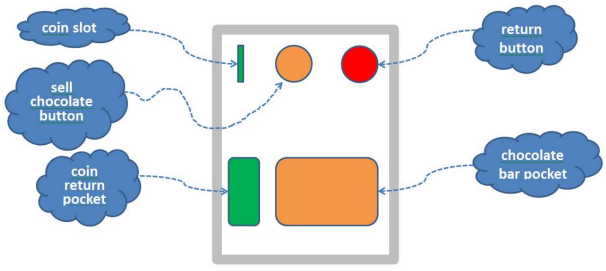

Fig. 1: Vending machine schematic image.

We compose the ES control program in the form of SN represented in Fig. 2. We use a toolset Tina [17] for graphical design an verification of ESs. Places, depicted as circles, and transitions, depicted as squares, are connected via arcs. Places specify conditions and variables, while transitions specify events or actions. To the left, input places, which are mapped

into sensors, are situated. To the right, output places, which are mapped into actuators, are situated. Textual labels specify details of the contact places (input and output) mapping onto pins of certain hardware.

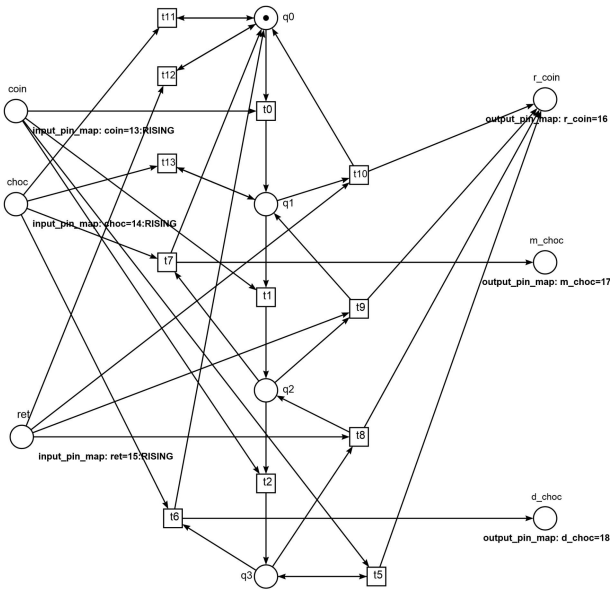

Fig. 2: SN of vending machine control.

The longest simple loop $q_0t_0q_1t_1q_2t_2t_6q_0$ corresponds to a regular routine of selling the dark chocolate bar: accept 3 coins, accept sell chocolate button, output dark chocolate bar, and return to the initial state. Index of the place in this loop corresponds to the number of received coins.

Using incoming arcs of a transition, we synchronize events and processes. For instance, on insertion of a coin, a token appears within the input place *coin* that cause transition t_0 to fire and reset the input place *coin* and move the state token from place q_0 into place q_1. In case the chocolate button is pressed in this state, a token appears within the input place *choc*, which is just reset by transition t_13 without the state change. When the next coin is inserted, the state token is moved to place q_2 by transition t_1. In this state, pressing the chocolate button leads to firing transition t_7 causing the state token to return into place q_0 and produce a token within the output place *m_choc* that, because of mapping onto the device pin, starts actual placing of the corresponding chocolate bar into the chocolate bar pocket.

During the design process, we can simulate the net behavior using graphical stepper simulator of Tina, adding tokens to the input places and observing the output place state. For hierarchical design of models, we use compiler-linker of SNs described in [14].

C. Supplying ES with Plant Model

We can use SN to design a model of plant as well, either of a specific or general form. Because the general plant model for vending machine is rather cumbersome, we show in Fig. 3 a specific model that describes a fun of milk chocolate behavior.

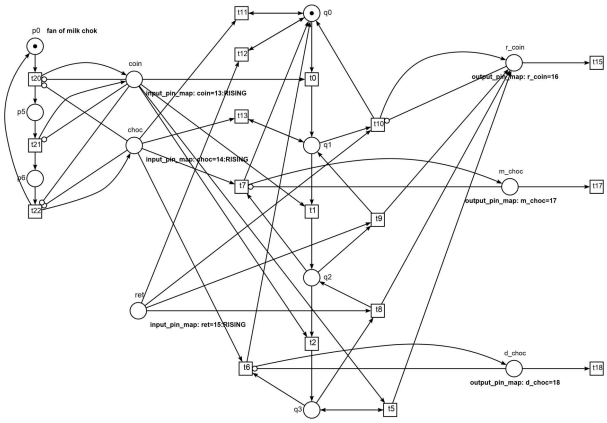

Fig. 3: ES with plant model of a milk chocolate fun.

The subnet to the left imitates insertion, one by one, of two coins and then pressing the sell chocolate button. The subnet to the right, is represented by hanging transitions which imitate removing the pockets' content. A loop $p_0t_20p_5p_21p_6t_22$ specifies a simplified human being behavior; in particular firing transitions t_20 and t_21 inserts a token into place *coin* while firing transition t_{22} inserts a token into place *choc* that is recognized as pressing the chocolate button. Inhibitor arcs, connected with the corresponding transitions, check that a token has been processed and consequently removed from an input place, before inserting a new token.

D. ES Verification

The spectrum of formal verification techniques starts with state space construction and analysis, state space of integrated model shown in Fig. 3 is represented in Fig. 4. A node of state-space corresponds to an SN marking. Markings are inscribed by the nodes as sets of places having nonzero marking. Usually the place symbol is preceded by its marking, traditionally, markings values equal to unit are omitted. Thus, marking specification $\{coin, p_6, q_1\}$ means that places $coin$, p_6, and q_1 contain a token each while other places do not contain tokens. Arcs connecting states are inscribed with the symbol of transition which fires casing the marking change.

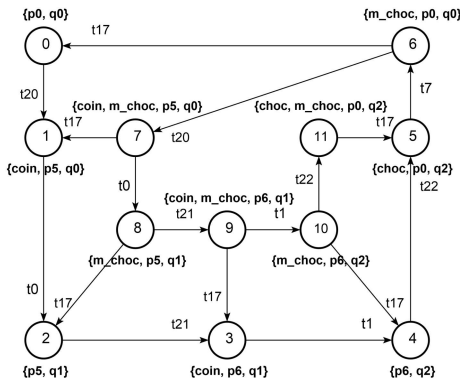

Fig. 4: State space of integrated model shown in Fig. 3.

Also, Tina provides linear algebra methods of analysis with linear invariants of places and transitions. Basic techniques for finding deadlocks and liveness enforcing [15], [16], as well as other methods of formal analysis [3], are applicable and can be implemented as Tina plug-ins.

III. ES Design on Microcontrollers

For MC, we employ a preliminary developed SN machine called *sna.ino*. This SN machine uses such parameters as a given SN declarations and pin map in the form of C language *.h* files produced by our plug-in *SNtoMC* for Tina. The general toolchain scheme is shown in Fig. 5. Using Arduino IDE [18] as a central tool allows us to employ a wide range of MCs supported by the Arduino open hardware concept; for practical implementations we were using MC Raspberry Pi Pico [19]. Developed tools and example SN programs are uploaded on GitHub [28] for public use.

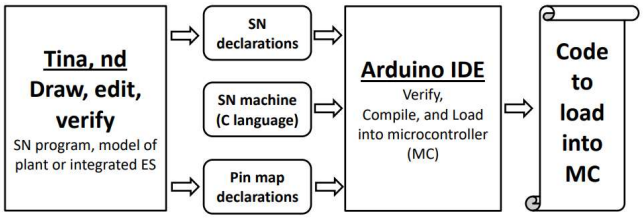

Fig. 5: ES design for MCs, general toolchain.

Our SN machine for MCs represents further development of SN VMs described in [14], in particular, we use a variant of SN VM for GPU, replacing certain multidimensional grid structures by the corresponding (nested) loops. To address the memory constraints of microcontrollers and FPGAs, we introduce a specialized data structure, we called Indexed Column-Wise Sparse Matrix (ICWSM), which minimizes memory usage by storing only non-zero elements, significantly optimizing both storage and access time compared to conventional matrix representations [14]. This structure is particularly advantageous for SN applications, where sparsity is common, allowing our SN machine to achieve up to 3-100 times speed-up.

To study snippets of basic algorithms and data structures, we will use a rather simple SN for addition of two nonnegative integer numbers shown in Fig. 6.

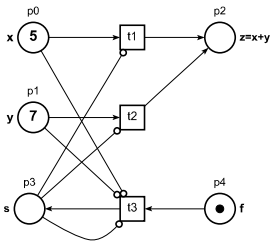

Fig. 6: An example of SN that computes a sum of two nonnegative numbers.

In the bottom of the figure, an inverse control flow is represented to start computations and recognize their completion. The complete state space for SN does not depend on the marking size and contains five states, including initial and final, graphically represented in [14]. Each of three transitions fires once in case of nonzero marking of data places x and y; transition t_3 finalizes the computation, removing a token from the final place f. An inverse control flow means that actually we advance zero marking similar to moving "holes" in electronics. We prefer the reverse control flow because zero check with an inhibitor arc does not restrict the firing multiplicity of a transition.

A code snippet of *sn-add.h* file for conventional matrices follows:

```
#define m 5
#define n 3
int b[m][n]={{1,0,-1},{0,1,-1},{0,0,0},
  {-1,-1,-1},{0,0,1}};
int d[m][n]={{0,0,0},{0,0,0},{1,1,0},
  {0,0,1},{0,0,0}};
int mu[m]={5,7,0,0,1};
```

Matrices b and d represent mapping A specifying the incoming and outgoing arcs of transitions, respectively; array *mu* represents the current marking μ; constants m and n equal to the number of places and transitions, respectively.

Enumerating each of matrices b and d, first by columns, then by rows, within nested loops, we produce ICWSM, which code snippet of *sn-add-icwsm.h* file follows:

```
#define kb 8
#define kd 3
int ibt[n+1]={0,2,4,8};
int b_icwsm[kb][2]={{0,1},{3,-1},{1,1},
  {3,-1},{0,-1},{1,-1},{3,-1},{4,1}};
int idt[n+1]={0,1,2,3};
int d_icwsm[kd][2]={{2,1},{2,1},{3,1}};
```

Let us consider *b_icwsm*, which contains all nonzero elements of matrix b preceded by the index of its row. An additional array *ibt* contains indexes of *b_icwsm* from where places of the corresponding transition start. We explain the ICWSM data structure with Fig. 7. Using sequential pairs of *ibt* elements, we arrange loops on input places of a transition. Thus, to calculate the fireability condition of transitions y, we use the following code snippet:

```
#define FMA(mui,w)
  (((w)==-1)?(((mui)==0)?MAX_INT:0):(mui)%(w))
for(j=0;j<n;j++){
  y[j]=MAX_INT;
  for(k=ibt[j];k<ibt[j+1];k++){
    p=b_icwsm[k][0];
    w=b_icwsm[k][1];
    y[j]=min(y[j],FCA(mu[p],w));
  }
}
```

Within macros FMA (Firing Multiplicity of an Arc), we take into consideration both regular and inhibitor arcs. An inhibitor arc produces an abstract infinite firing multiplicity in case the place marking *mui* equals zero, represented within C code by the maximal integer value *MAX_INT*. The regular arc firing

multiplicity *(mui)%(w)* complies with (1), minimal values are taken within the loop.

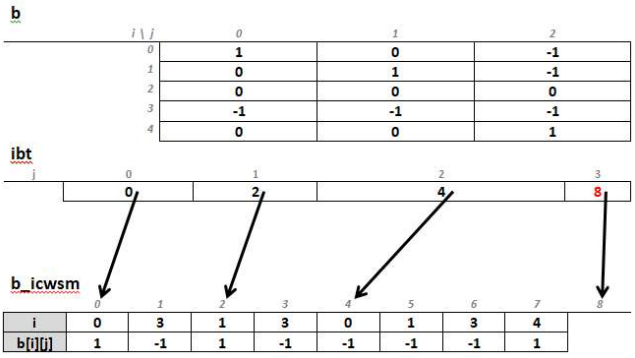

Fig. 7: Explanation of ICWSM data structure.

For the firing transition choice, a simulated nondeterministic choice procedure is applied after filtering lower priority fireable transitions based on the transitive closure of the priority relation between transitions. After the firing transition *tf* choice, with multiplicity *tc*, we fire it, according to (2), using the following code snippet:

```
for(k=ibt[j];k<ibt[j+1];k++){
  p=b_icwsm[k][0];
  w=b_icwsm[k][1];
  mu[p]-=tc*w;
}
for(k=idt[j];k<idt[j+1];k++){
  p=d_icwsm[k][0];
  w=d_icwsm[k][1];
  mu[p]+=tc*w;
}
```

Thus, instead of processing *m* elements of the corresponding matrix column to find the fireability multiplicity of a transition with a conventional matrix, we process the nonzero elements only using our ICWSM. For sparse matrices characteristic for Petri/Sleptsov net applications, we have 2-10% of nonzero elements that yield 3-100 times speed-up besides the reduced matrix size. To reduce the memory usage, we can consider the maximal arc multiplicity that in majority of applications is modest and allocate 1 byte of memory with type *char*. In case of wider range, we can allocate 2 bytes of memory with *short int*. We can adjust memory usage separately for indexes and values, splitting each of arrays *b_icwsm* and *d_icwsm* into 2 corresponding arrays of different types.

Based on the described in this section technique and toolchain, a prototype of vending machine, considered in Section II-B, was implemented on MC and shown in Fig. 8. Green button represents the coin sensor, yellow button sells chocolate bars, and red button returns coins. Green LED represents coin return while yellow and blue LEDs represent milk and dark chocolate bars, respectively.

The prototype has been tested on correct and incorrect sequences of input events represented by a certain sequence of pressing buttons, debouncing facilities provided for convenience of work. Its behavior is rather stable and completely corresponds to the specifications.

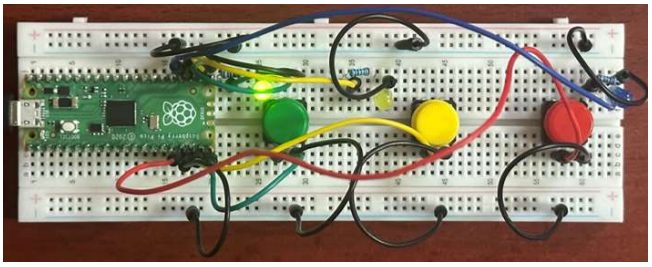

Fig. 8: Prototyping vending machine on MC Raspberry Pi Pico.

IV. ES Design on FPGAs

For SN implementation on FPGA, we use a completely different approach, compared to MC, which toolchain is represented in Fig. 9. Taken from *.h* file, declaration of an SN is compiled into Verilog [20] code. Thus, no general SN machine is applied; each SN is implemented individually in hardware according to its Verilog specification. Besides, the pin map declaration file is compiled into FPGS file of constraints. Thus two corresponding files *.v* and *.cts* are produced by our ad-hoc software *SNtoFPGA* and provided as input for *Gowin FPGA Designer* [21]. Here we use this specific tool because of prototyping with Tang Nano 9k and Tang Mega 138k Pro FPGAs [22]. However, for other types of FPGA, hardware specification in Verilog is applicable as well. Developed tools and example SN programs are uploaded on GitHub [29] for public use.

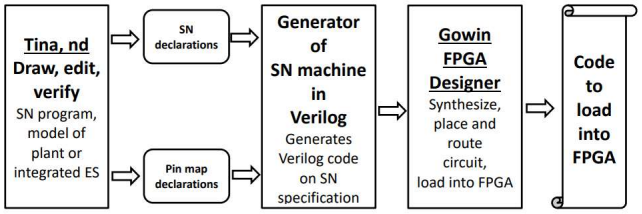

Fig. 9: ES design for FPGAs, general toolchain.

For SN shown in Fig. 6, we obtain the following snippet of Verilog code, using FMA macros similar to one specified for MC, for computing transitions' firing multiplicities:

```
reg [5:0] p [0:5];
reg [5:0] y [0:3];
always @(posedge sys_clk) begin
  y[0] = MAX_MU;
  y[0] = min(y[0],`FMA(p[0],1));
  y[0] = min(y[0],`FMA(p[2],-1));
  y[1] = MAX_MU;
  y[1] = min(y[1],`FMA(p[3],1));
  y[1] = min(y[1],`FMA(p[2],-1));
  y[2] = MAX_MU;
  y[2] = min(y[2],`FMA(p[0],-1));
  y[2] = min(y[2],`FMA(p[2],-1));
  y[2] = min(y[2],`FMA(p[3],-1));
```

```
  y[2] = min(y[2],`FMA(p[4],1));
  ...
end
```

Here we use a predefined constant *MAX_MU* with the required number of bits to represent an infinite firing multiplicity. To compose the Verilog code, we developed a generator *SNtoFPGA* in C language, implemented as Tina plug-in, which code snippet to generate the above Verilog code follows:

```
printf("reg [%d:0] p [0:%d];\n",bits,m);
printf("reg [%d:0] y [0:%d];\n",bits,n);
printf("always @(posedge sys_clk) begin\n");
for(j=0;j<n;j++){
  printf("y[%d] = MAX_MU;\n",j);
  for(i=0;i<n;i++){
    if(b[i][j]!=0)
      printf("y[%d] = min(y[%d],FMA(p[%d],
                %d));\n",j,j,i,b[i][j]);
  }
}
printf("end\n");
```

In a similar way, we generate a Verilog snippet to fire a chosen transition *tf*, decrementing the incidental place markings according to matrix *b* and incrementing them according to matrix *d*, which follows:

```
case(tf)
  0: begin
       mu[0] = mu[0] - tc*1;
       mu[1] = mu[1] + tc*1;
     end
  1: begin
       mu[3] = mu[3] - tc*1;
       mu[1] = mu[1] + tc*1;
     end
  2: begin
       mu[4] = mu[4] - tc*1;
       mu[2] = mu[2] + tc*1;
     end
endcase
led = ~mu[1][5:0];
```

Note that here and within other snippets, for generality of the narrative, we print division and multiplication by unit for arcs of unit multiplicity; within an optimized code, we omit these "no-data-change" operations. A C code snippet that generates the above Verilog code follows:

```
printf("case(tf)\n");
for(j=0;j<n;j++){
  printf("%d: begin\n",j);
  for(i=0;i<n;i++){
    if(b[i][j]>0)
      printf("mu[%d] = mu[%d] -
                tc*%d;\n",i,i,b[i][j]);
    if(d[i][j]>0)
      printf("mu[%d] = mu[%d] +
                 tc*%d;\n",i,i,d[i][j]);
  }
  printf("end\n");
}
printf("endcase\n");
printf("led = ~mu[%d][%d:0];\n",res_p,bits_mu);
```

Thus, on the SN graphical representation in Tina, we generate a Verilog program that, when compiled and routed by *Gowin FPGA Designer*, implements a given SN in hardware, i.e., SN runs on FPGA directly without using an intermediate software, such as a generalized SN machine for MCs. Besides, on the additional annotations of SN input and output places, we produce the constraints file that specifies the contact places mapping into FPGA pins. Since varying instrumental software for various types of FPGA supports Verilog, the approach is applicable to other types of FPGA, for instance Xilinx and Altera.

V. Benchmarks

After prototyping vending machines and lifts, we are looking for opportunities to apply SN based ES design to real-life systems such as robotic arms and integrated shops of automated manufacture, as well as vehicle control systems.

We consider obtaining comprehensive benchmarks as a necessary stage of proving the approach's robustness to motivate its application at the enterprise level. For preliminary testing, we apply SNs for basic arithmetic operations [1]: addition, multiplication, and division. Then, we use a series of SN benchmarks for parametric nets shown in Table I, developed and applied for performance evaluation [14] of laptop and desktop SN processors for CPU and SN machines for GPU. Detailed specifications of benchmark nets for a given size parameter are shown in Table II. We compare laptop (desktop) performance with MC and MC with FPGA.

TABLE I: SNs for benchmarks

Net abbreviation	Net description
de	Exact computer of double exponent after Lipton
pol	Polynomial computed on nonegative integers
mm	Multiplication of nonnegative integer matrices

TABLE II: Parameters of benchmark SNs

Net	Places	Transitions	Arcs		
			Regular	Inhibitor	Priority
de2	46	34	180	0	8
de3	68	53	281	0	12
de4	90	72	382	0	16
pol2	117	124	477	178	27
pol3	215	231	891	330	54
pol4	340	368	1422	524	90
pol5	492	535	2070	760	135
pol6	671	732	2835	1038	189
pol10	1657	1820	7065	2570	495
pol15	3497	3855	14985	5430	1080
pol20	6012	6640	25830	9340	1890
pol25	9202	10175	39600	14300	2925
mm2	294	310	1203	445	72
mm6	8966	9542	36723	13789	1944
mm8	21570	22978	88323	33217	4608

Net *de* represents a busy beaver [23] construct for Petri/Sleptsov nets. It was composed after the seminal net of Lipton [24], a strong computer of double exponent function 2^{2^n}, supplied with $4n$ priority arcs to make it an exact

computer of double exponent. Net *pol* computes a polynomial of degree n. Net *mm* represents a square matrix multiplier, the matrix size equals to n. Both models are composed using the multiplication and addition nets. Note that for the size parameter, specified in Table II after the net name, each net is generated using a dedicated program in C language [14]. Compared to *de*, the two last models represent rather memory consuming tasks which run fast. We consider polynomials and matrix multiplication also because of the fact that linear control approach [25] to ES design is easily expressed using these constructs.

Let us specify the hardware employed in benchmarks. We use an x64 laptop Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz, 8G. We use MC Raspberry Pi Pico [19] with RP2040 SoC, 2MB Flash, Dual ARM Cortex-M0+ @ 133MHz, 264kB on-chip SRAM. We use FPGA Tang Mega 138K [22] with 50MHz FPGA chip GW5AST-LV138PG484A, 138,240 LUTs, 300 DSP units, 1080 S-SRAM, 6120 B-SRAM. Some preliminary benchmarks were made for Tang Nano 9k which runs twice slower because of about two time fewer frequency (27MHz). Though, Tang Nano 9k capabilities stop sufficing *de5* because of its limitation on the number of LUTs.

Benchmarks for MC are shown in Table III, where implementations using a conventional matrix and indexed sparse data structures are compared. Dash means run is impossible because of memory overflow. We conclude that the indexed sparse matrix data structure optimizes both memory consumption and performance. For instance, with *mm3*, we obtain memory overflow using conventional matrices, while, with ICWSM, we consume only 4% of MC memory. We obtained similar benchmarks for the laptop as well. On average, the laptop runs about 200 times faster compared to MC, showing approximately the same speed-up when using sparse matrices, under the laptop C program compiled without the code optimization.

TABLE III: Benchmarks on MC.

Net	Conventional matrix		Indexed sparse	
	Time, s	Memory usage	Time, s	Memory usage
de2	0.1647	2%	0.03738	2%
de3	8.03744	3%	1.12876	2%
de4	3556.78	5%	444.374	2%
pol2	1.53078	6%	0.04204	2%
pol3	10.2341	17%	0.15720	3%
pol4	43.6115	39%	0.42297	3%
pol5	134.080	80%	1.06515	3%
pol6	-	-	2.74920	4%
pol10	-	-	21.0305	6%
pol15	-	-	95.5430	10%
pol20	-	-	284.561	16%
pol25	-	-	671.616	23%
mm2	31.4402	29%	0.36683	3%
mm3	-	-	8.81818	4%
mm6	-	-	677.976	22%
mm8	-	-	3863.73	49%

Benchmarks for FPGA are represented in Table IV. For FPGA, the most critical parameter is space complexity, expressed in the resulting circuit consumption of register memory and LUTs. Though, when a task suits FPGA space limitations, its performance is about 10^4 times better than that of the MC. Thus, SN based ES design can suit the requirements of extremely fast plants, such as a hypersonic flying object, collider, thermonuclear fusion reaction, etc. One time cycle of device corresponds to time required to make one SN step. For 50MHz FPGA clock, we make SN step in about 20 ns for rather big nets. We show the number of cycles multiplied by 2, 3, and 4 for bigger nets, which require the clock division by the corresponding constants.

TABLE IV: Benchmarks on FPGA.

Net	Memory	Cycles	Time
de3	6%	6019	$1.2038 \cdot 10^{-4}$ s
de4	7%	1737685	0.0347537 s
de5	27%	130670194729x2	87.1135 min
pol3	12%	259x2	$1.036 \cdot 10^{-5}$ s
pol4	19%	416x3	$2.496 \cdot 10^{-5}$ s
pol5	25%	606x4	$4.848 \cdot 10^{-5}$ s
mm2	11%	333x2	$1.332 \cdot 10^{-5}$ s
mm3	59%	574x4	$4.592 \cdot 10^{-5}$ s

In Fig. 10, speed-up because of using our ad-hoc indexed sparse data structure ICWSM is shown for SN machine written in C language, when it runs on MC Raspberry Pi Pico. Speed-up from 4 to 120 times is provided. Note that MC measurements are more precise compared a laptop because it is nearly bare hardware with Arduino small runtime code and libraries. For big nets, the speed-up increases because of longer time for nonzero elements search with conventional matrix. Reduction of required memory varies from 3 to 50 times, growing up for bigger data.

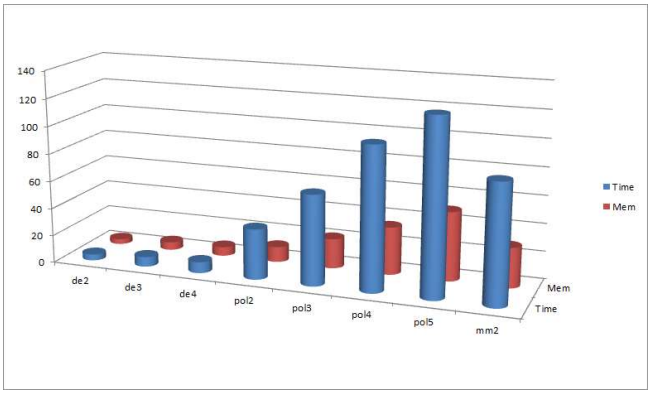

Fig. 10: Speed-up of computations (Time) and reduction of memory utilization (Mem) because of using indexed sparse matrix on MC (measured in times).

In Fig. 11, the speed-up of FPGA compared to MC is shown. For small, computationally busy tasks, such as *de*, about 5 thousand times speed-up is observed. For big models, which compute rather fast, we obtain speed-up of about 30 thousand times.

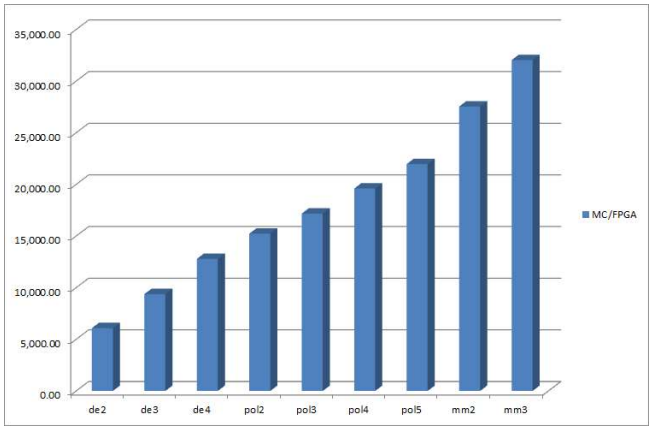

Fig. 11: Speed-up of computation because of using FPGA instead of MC.

Thus, FPGA is preferable when fast control is required. Though, we can upload considerably bigger models into Raspberry Pi Pico, whose cost is 15 times less than that of Tang Mega Pro.

VI. Conclusions

Thus, we presented Sleptsov net-based design of reliable Embedded Systems for both microcontrollers and FPGAs. Basic toolchains have been specified, including well known software as Tina modelling system for Petri and Sleptsov nets, Arduino IDE, and Gowin FPGA Designer. Additional tools have been developed, which include: converters of Sleptsov net file format, SN machine for microcontrollers, and generator of Verilog code for FPGA. Integration of other well-known software for microcontrollers and FPGAs, which supports programming in C/C++ and Verilog, is provided.

Among Sleptsov net based design advantages, we mention the vivid graphical representation of concurrent systems, the possibility of modeling discrete plants and the verifying integrated models, grounded by wide spectrum of formal techniques for Petri and Sleptsov net analysis, which include deadlock search and liveness enforcing. Thus, correctness of control or of entire control system, including plant, can be proven. Collateral research studies methods for generating a priori correct control specified by Petri/Sleptsov nets.

References

[1] D.A. Zaitsev, "Sleptsov Nets Run Fast," in IEEE Transactions on Systems, Man, and Cybernetics: Systems, vol. 46, no. 5, pp. 682-693, May 2016.
[2] C.A. Petri, Kommunication mit Automaten, Technischen Hoschule Darmstadt, 1962, Ph.D. thesis.
[3] M. Diaz, Petri Nets: Fundamental Models, Verification and Applications (John Wiley & Sons, 2013).
[4] D.A. Zaitsev, "Strong Sleptsov nets are Turing complete," in Information Sciences, Volume 621, 2023, pp. 172-182.
[5] B. Berthomieu, D.A. Zaitsev, "Sleptsov nets are Turing-complete," in Theoretical Computer Science, vol. 986, 2024.
[6] D.A. Zaitsev, "Universal Sleptsov net," in International Journal of Computer Mathematics, vol. 94, no. 12, pp. 2396-2408, 2017.
[7] S. S. Peng and M. C. Zhou, Sensor-based stage Petri net modeling of PLC logic programs for discrete-event control design, Int. J. Prod. Res., vol. 41, no. 3, pp. 629-644, Feb. 2003.
[8] A.I. Sleptsov, A.A. Yurasov. Computer-Aided Design of Flexible Computer-Aided Manufacturing Systems. Tekhnika, Kyiv, 1986.
[9] Jensen K, Kristensen LM. Coloured Petri nets: modelling and validation of concurrent systems. New York: Springer; 2009.
[10] D.A. Zaitsev, "Sleptsov Net Computing resolves problems of modern supercomputing revealed by Jack Dongarra in his Turing Award talk in November 2022," International Journal of Parallel, Emergent and Distributed Systems, vol. 38, no. 4, 2023, pp. 275–279.
[11] D.A. Zaitsev, D.E. Probert, "Preface for special issue Petri/Sleptsov net based technology of programming for parallel, emergent and distributed systems," International Journal of Parallel, Emergent and Distributed Systems, 36(6), 2021, pp. 495-497.
[12] A.A. Kostikov, N.D. Zaitsev, O.V. Subotin, Realisation of the double sweep method by using a Sleptsov net, Int. J. Parallel, Emergent Distributed Systems 36 (6) (2021) 516–534.
[13] T.R. Shmeleva, J.W. Owsinski, A.A. Lawan, Deep learning on Sleptsov nets, Int. J. Parallel, Emergent Distributed Systems 36 (6) (2021) 535–548.
[14] D.A. Zaitsev, T.R. Shmeleva, Q. Zhang, and H. Zhao, "Virtual Machine and Integrated Developer Environment for Sleptsov Net Computing," in Parallel Processing Letters, vol. 33, no. 3, 2023.
[15] Z. W. Li and M. C. Zhou, Deadlock Resolution in Automated Manufacturing Systems (Springer, 2010).
[16] M. Uzam, A. M. El-Sherbeeny, W. Guo and Z. Li, Design of an Improved Think Globally Act Locally Approach for the Computation of Petri Nets Based Liveness Enforcing Supervisors of FMSs, in IEEE Access, vol. 12, pp. 74367-74388, 2024.
[17] B. Berthomieu, P.-O. Ribet, F. Vernadat, The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets, Int. J. Production Res. 42 (14) (July 2004).
[18] M. Banzi, M. Shiloh. Getting Started With Arduino, O'Reilly, 2022.
[19] Raspberry Pi Pico, https://www.raspberrypi.com/products/raspberry-pi-pico/.
[20] J. Cavanagh. Verilog HDL: Digital Design and Modeling,CRC Group, 2017.
[21] R. Snider. Advanced Digital System Design using SoC FPGAs An Integrated Hardware/Software Approach, Springer, 2023.
[22] Tang Mega 138K Pro, https://wiki.sipeed.com/hardware/en/tang/tang-mega-138k/mega-138k.html.
[23] Pascal M. Small Turing machines and generalized busy beaver competition. TCS. 2004;326(1–3):45–56.
[24] R.J. Lipton. The reachability problem requires exponential space. Tech. Rep., 1976.
[25] Y. Bavafa-Toosi. Introduction to Linear Control Systems, Elsevier, 2017.
[26] Embedded Software Verification and Debugging, Eds.: Djones Lettnin, Markus Winterholer, Springer New Yourk, 2018.
[27] KCS Murti, Design Principles for Embedded Systems, Springer Nature Singapore, 2021.
[28] Ruiyao Xu, SN Machine for Arduino. Release v.1.1, Oct 21, 2024, https://github.com/Akalababa
[29] Si Zhang, SN Machine for FPGA: Verilog code generator. Release v.1.1, Oct 21, 2024, https://github.com/ZhangS2000