INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2026, VOL. 72, NO. 2, PP. 1-9
Manuscript received January 17, 2026; revised April 2026. doi: 10.24425/ijet.2026.157937

Designing safe finite state machines

Valery Salauyou

Abstract—**This paper addresses the design of safe finite state machines (SFSMs) using the Verilog hardware description language (HDL) alongside the Quartus design tool to implement FSMs on field-programmable gate arrays (FPGAs). Three styles of finite state machine (FSM) description are proposed (*safe*, *safe_error*, and *safe_idle*), which provide different options for SFSM implementation. Experimental studies on FSM benchmarks have shown that the presented approach reduces SFSM area by an average factor of 2.436 and improves performance by an average factor of 1.588 compared to synthesizing SFSMs in the Quartus design tool. Recommendations are also provided on the practical application of this approach for designing SFSMs.**

Keywords—**finite state machine (FSM); safe finite state machine (SFSM); hardware description language (HDL); illegal state; field programmable gate array (FPGA)**

I. Introduction

In digital systems, controllers and control units are crucial components that implement control algorithms for individual devices and subsystems, as well as for the entire system. Typically, controllers and control units are designed as finite state machines (FSMs).

Figure 1 shows the traditional structure of an FSM, where X is the input signals (input vector); Y is the output signals (output vector); Rs is the state register, which stores the code of the present state; λ is a combinational circuit (logic) that determines the code of the next state (λ: $X \times S \rightarrow S$); δ is a combinational circuit that forms the values of the output signals; S is the set of states of the FSM. For Moore FSM, the output signal values are formed based on the present state code (δ: $S \rightarrow Y$), and for Mealy FSM, based on the present state code and the input signal values (δ: $X \times S \rightarrow Y$). Therefore, for a Mealy FSM, the inputs X are connected to the inputs of the combinational circuit δ (marked with a dotted line in Fig. 1).

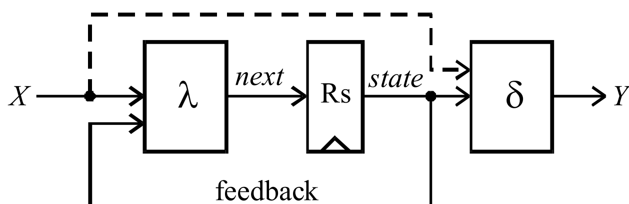

Fig. 1. Traditional structure of an FSM.

The set S of states of an FSM is the set of valid or legal states. If the state register Rs fails, the FSM may enter an *illegal* state, in which the FSM specification does not define transitions. In this case, the FSM may become stuck in an illegal state for an indefinite period, leading to its failure. The only ways to exit an illegal state are to reset the FSM or reset the entire system. This behavior can be unacceptable in many FSM applications, particularly when the system is operating in autonomous mode.

The causes of failure in the Rs state register can include solar radiation, cosmic rays, ionizing radiation from nuclear power plants and reactors, electromagnetic radiation, and laser beams from electronic warfare [1]. These factors are collectively referred to as external disturbances [2]. Additionally, equipment wear and design errors can also lead to failures in the state register.

An FSM is referred to as a safe FSM (SFSM) if it can always return to its initial state from any illegal state without resetting the FSM or the entire system [3]. The design of SFSMs is crucial in applications such as medical life support equipment, transportation systems (particularly avionics and unmanned aerial vehicles), and autonomous robotic systems. Notably, design tools such as Quartus [3] and Vivado [4] offer options for synthesizing SFSMs. In these synthesis modes, additional logic is incorporated into the FSM to ensure it exits illegal states. However, this extra logic can significantly increase the area of the FSM circuit and reduce its overall performance.

This paper addresses the problem of designing SFSMs at the hardware description language (HDL) level. It introduces three approaches for describing SFSMs in Verilog HDL that demonstrate improvements in both area and performance. These approaches outperform not only the FSM synthesis method available in the Quartus design tool [3] but also the traditional method of describing FSMs using three processes [5].

The primary objective of this work is to develop efficient styles for SFSM descriptions in HDL, with a focus on area and performance.

The main contributions of this work are:

- an assertion, derived from the analysis of the number of illegal states in FSMs, that minimum-length codes should be employed to encode the states of SFSMs;
- the definition of three types of SFSMs: *safe* SFSMs, which transition from any illegal state to the initial state, *safe_error* SFSMs, which also generate an *error* signal, and *safe_idle* SFSMs, which additionally transition to an *idle* state;
- an analysis of how to describe SFSMs using Verilog HDL in the Quartus design tool;
- three styles for describing SFSMs in Verilog HDL: *safe, safe_error*, and *safe_idle*;
- experimental studies assessing the effectiveness of the proposed styles for describing SFSMs;

This work was supported by a grant WZ/WI-III/5/2023 from Bialystok University of Technology and founded from the resources for research by Ministry of Science and Higher Education.

Valery Salauyou is with Bialystok University of Technology (e-mail: v.salauyou@pb.edu.pl).

© The Author(s). This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY 4.0, https://creativecommons.org/licenses/by/4.0/), which permits use, distribution, and reproduction in any medium, provided that the Article is properly cited.

- recommendations for the practical application of the proposed styles for describing SFSMs.

The proposed method for designing SFSMs is unuversal. It can be easily adapted for use with other HDLs, such as VHDL or SystemVerilog, as well as with different design tools, such as Vivado. Additionally, it is applicable across various hardware platforms, including systems-on-a-chip (SoC) and application-specific integrated circuits (ASICs).

The structure of this paper is as follows. Section II provides an overview of related work. Section III introduces key concepts and methods for designing SFSMs. Section IV presents the experimental results and their analysis, along with recommendations for the practical application of the discussed approach. Finally, Section V provides a summary of the paper.

II. RELATED WORKS

The design of SFSMs is directly related to the design of robust FSMs [6]. There is a wide variety of approaches to the design of robust FSMs. Let us consider some of them.

In [7], a multi-level system of FSMs for robot control is proposed to improve robustness. FSMs at all levels receive the same input signals. In the event of a failure, a higher-level FSM can subsume the functions of a lower-level FSM, suppressing its output signals. In [8], the design of robust FSMs that are resistant to uncertainties is considered. For this purpose, a deterministic model of uncertainty is introduced, yielding a dynamic game formulation of the robustness control problem. The task of stochastic control with risk for hidden Markov models is formulated and solved when the uncertainty model is stochastic. In [9], an embryonics (embryonic electronics) approach to creating robust integrated circuits capable of self-reproduction and self-repair is proposed; in particular, the concept of creating robust self-reproducing FSMs is considered. In [10], the theoretical foundations of FSM robustness are considered when FSMs are approximated by linear time-invariant (LTI) models of finite order; a parallel approach is developed in which stochastic FSMs are modeled as LTI systems. In [11], the robust FSMs are considered from the point of view of stability. To this end, three concepts of input/output stability are introduced: finite-gain input/output stability, external stability, and incremental input/output stability.

The robustness of FSMs is often used to improve the robustness of hardware watermarks implemented as FSMs. In [12], a robust watermarking scheme based on FSMs is proposed for protecting intellectual property in sequential circuit designs.

The robustness of FSMs can also be improved by FSM state encoding, which is particularly important for asynchronous FSMs. Reference [13] describes methods of anti-racing encoding of states of the asynchronous FSMs to enhance the robustness of digital systems.

The design of robust FSMs is also related to the development of robust controllers. In [14], the concept of robust controllability for FSMs is introduced, including robust controllability between states and robust controllability between states and sets of states. The condition for the existence of robust controllers for FSMs is established.

A robust deadlock-avoidance strategy based on the state space of timed Petri nets that accounts for unreliable resources is presented in [15]. The proposed approach involves classifying state classes into two sets: the set of legal state classes and the set of illegal state classes. The design of FSMs (controllers) to prevent illegal state classes is considered, thereby ensuring the operability of the controlled system despite resource failures. In [16], a methodology for designing robust, low-latency controllers based on asynchronous FSM is presented.

The theoretical basis for designing robust and safe FSMs is robust discrete-event systems. In [17], the problem of constructing a robust state-feedback controller for discrete-event systems (objects) with internal uncertainty is considered; a sufficient condition for the existence of such a robust supervisor is also presented. In [18], an approach to robust supervisory control of discrete-event systems is presented. The proposed approach assumes that the current state is known only within a non-empty subset of the system states. Reference [19] presents results on fault-tolerant and robust control to study the functionality of distributed discrete-event systems under fault conditions. In [20], feedback control systems are investigated in which an attacker's intent can compromise sensor readings by damaging the system. The problem is studied at the control level using methods from discrete-event systems.

Reference [21] discusses the standard behavior of synthesis tools with regard to changes caused by optimization. It shows that aggressive optimization negatively affects the robustness of FSMs in the case of external disturbances.

The following works are closely related to the topic of designing SFSMs. In [22], an efficient representation is proposed that incorporates a finite set of parameters into FSMs when modeling discrete-event systems. The paper also presents algorithms for the autonomous and online synthesis of security management policies. In [23], FSMs extended with variables are discussed, and an algorithm for synthesizing a supervisor is proposed. This algorithm enhances the protective functions of an FSM by making prohibited or blocking states in the controlled object unattainable. In [24], the SFSM is utilized to control an elevator. In [25], FSMs are employed to analyze the safety of complex systems, such as aircraft engines. Reference [26] examines the safe configuration of asynchronously interacting FSM systems. An FSM-based method for modeling and analyzing process safety is presented in [27].

An analysis of the most relevant works on this topic shows that there are currently no methods that are directly aimed at designing safe FSMs. Although some design tools, such as Quartus and Vivado, offer SFSM synthesis, enabling this option adds logic to the FSM circuit that allows transitions from invalid states back to the initial state. However, this approach results in a significant increase in area and a degradation in FSM performance. This article proposes SFSM design methods that offer new possibilities for SFSM design, significantly reducing area and improving performance compared to the Quartus design tool approach.

III. METHODS

This section analyzes the number of illegal states in FSMs and asserts that minimum-length state codes should be used for the synthesis of SFSMs. Three types of SFSMs are presented,

based on which three styles of describing SFSMs in Verilog HDL are proposed, taking into account the peculiarities of describing FSMs in the Quartus design tool. The proposed styles for describing SFSMs are compared with the traditional description and the method for synthesizing SFSMs in the Quartus design tool.

A. Number of illegal states of FSMs

Let Sp be the set of potential states to which the FSM can transition. The number of elements in Sp is determined by the number of binary codes that can be stored in the state register Rs:

$$|Sp| = 2^R , \tag{1}$$

where $|A|$ is the cardinality of set A; R is the number of bits in the state codes (code length).

For binary codes

$$R = \lceil \log_2 M \rceil, \tag{2}$$

where $\lceil B \rceil$ is the smallest integer greater than or equal to B, and M is the number of states of the FSM.

For one-hot codes, the code length R is determined using the following expression:

$$R = M, \tag{3}$$

i.e., the length of the code is equal to the number of states of the FSM.

The set of Si illegal states of an FSM is defined as follows:

$$Si = Sp \backslash S. \tag{4}$$

For sets S, Sp, and Si, the following equalities hold:

$$\begin{aligned} Si \cap S &= \emptyset; \\ Si \cup S &= Sp; \end{aligned} \tag{5}$$

where $\emptyset$ is the empty set.

The number of illegal states of an FSM is determined by the cardinality of the set Si:

$$|Si| = |Sp| - |S| = 2^R - M. \tag{6}$$

From expression (6) it follows that the number of illegal states depends on the code length R. Therefore, for one-hot encoding we have:

$$|Si| = |Sp| - |S| = 2^R - M. \tag{7}$$

and in the case of binary encoding, the number of illegal states is determined by the following expression:

$$|Si| = 2^{\lceil \log_2 M \rceil} - M. \tag{8}$$

Note that when the number of states M is a power of two and binary encoding is used, an FSM cannot enter an illegal state. However, in this case, a failure in the state register Rs leads to an invalid transition from one legal state to another, which also causes the FSM to fail.

Table I presents a comparison of the number of illegal states as a function of the number of states and the FSM state encoding method.

From expression (7) and Table I, it follows that when using one-hot encoding, the number of illegal states of FSMs increases exponentially with increasing length of state codes. Therefore, to reduce the number of illegal states and improve the safety of FSMs, the states of FSMs should be encoded using minimum-length codes, such as binary or Gray codes.

In design tools, one-hot encoding is typically used by default to enhance FSM parameters, including area, performance, and power consumption. This means that the goals of improving FSM parameters and increasing FSM safety often conflict.

TABLE I
NUMBER OF ILLEGAL STATES OF FSMS

M	Binary $\lvert Si \rvert$	One-hot $\lvert Si \rvert$
2	0	2
3	1	5
4	0	12
5	3	27
6	2	58
7	1	121
8	0	248
9	7	503
10	6	1014
11	5	2037
12	4	4084

B. Types of SFSMs

Fig. 2 demonstrates the structure of the state transition graph (STG) for an SFSM of the *safe* type.

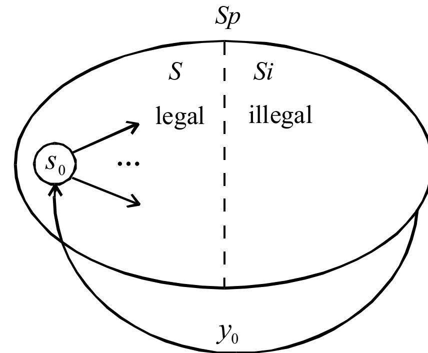

Fig. 2. STG of the *safe* type SFSM.

In Fig. 2, the set Sp of potential states of an FSM is divided into two subsets: S, the set of legal states, and Si, the set of illegal states. Transitions from all illegal states are unconditional. They lead to the initial state s_0, and a zero output vector y_0 is formed on these transitions. In the event of a failure in the state register Rs, the *safe* type SFSM returns to the initial state s_0, creating a zero vector y_0 at its outputs.

The disadvantage of an SFSM of the *safe* type is that the user (i.e., the external environment) cannot explicitly determine whether there was actually a failure in the state register Rs and whether the FSM entered an illegal state. The only indication of this event is the zero vector y_0 at the output of the FSM. However, if some legal transitions of the FSM provide for the formation of a zero output vector, it is impossible to detect a failure in the state register Rs.

One solution to this problem could be to explicitly generate an additional *error* output signal, as shown in Fig. 3.

In an SFSM of the *safe_error* type, on unconditional transitions from illegal states to the initial state s_0, in addition to the zero output vector y_0, the *error* signal equal one. The

error signal indicates an error when the FSM enters an illegal state.

The disadvantage of *safe_error*-type SFSMs is that the *error* signal is set for a very short period of time (especially for Mealy FSMs), and the external environment may not have time to perform the necessary actions to compensate for the consequences of the FSM entering an illegal state.

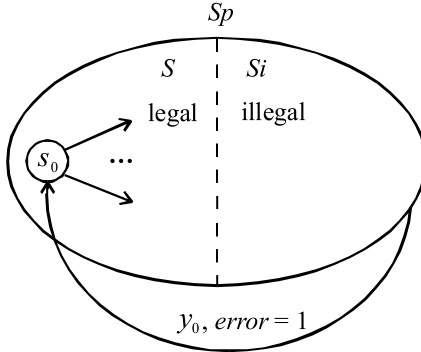

Fig. 3. STG of the *safe_error* type SFSM.

Figure 4 shows the structure of the STG of a *safe_idle* type SFSM when an additional *idle* state is used.

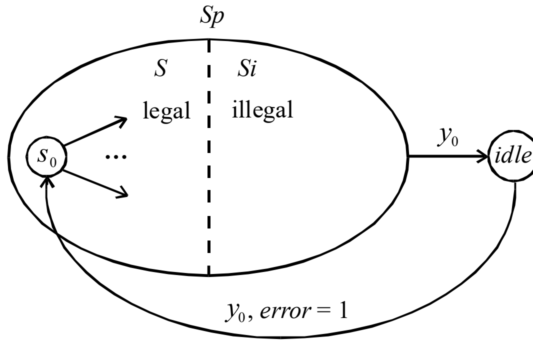

Fig. 4. STG of the *safe_idle* type SFSMs.

In the event of an illegal state, the SFSM in Fig. 4 transitions to an additional *idle* state, from which it returns to the initial state s_0, generating the signal *error* equals one and a zero output vector y_0. For Moore FSMs, the value of signal *error* is equal to one is generated in the *idle* state. If necessary, in the *idle* state, in addition to the zero output vector y_0 and *error* = 1, additional output signals or output vectors can be generated to compensate for the consequences of the FSM entering an illegal state.

C. Traditional description of FSMs

As an example, consider an FSM whose STG is shown in Fig. 5.

Our FSM is a Mealy FSM (the most common case), has three states $s_0,\ldots,s_2$, two input variables x_0,x_1, and three output variables $y_0,\ldots,y_2$. The vertices of the STG in Fig. 5 represent the states of the FSM, and the edges of the STG correspond to the transitions between the states of the FSM. Next to each arc is the input vector that initiates the transition, and the output vector resulting from the transition is written after a slash. A hyphen ("-") in the input or output vector indicates a don't care value.

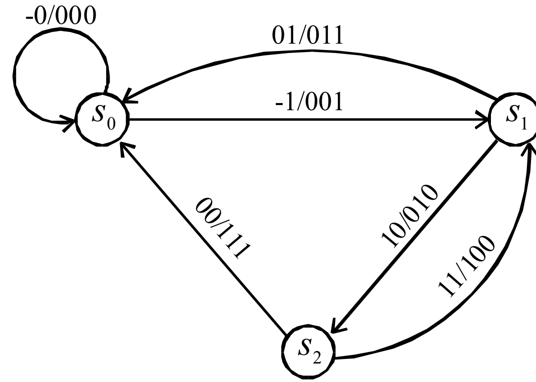

Fig. 5. STG of the Mealy FSM.

Traditionally, an FSM in Verilog HDL is described using three separate processes [5], as shown in Listing 1.

Listing 1 A traditional description of the FSM from our example.

```
module Mealy_3proc (   // declaration of ports
  input clk, reset,            // clock and reset signals
  input [1:0] x,               // input vector
  output reg [2:0] y);         // output vector
  reg[1:0] state, next;        // state variables
  localparam [1:0] s0 = 0, s1 = 1, s2 = 2;   // FSM states
  always @(posedge clk, negedge reset)    // state register
    if (~reset)  state <= s0;
    else         state <= next;
  always @(*)    // description of transition functions
    case(state)
    s0: casex(x)
          2'b?0:   next = s0;
          2'b?1:   next = s1;
        endcase
    s1: casex(x)
          2'b01:   next = s0;
          2'b10:   next = s2;
        endcase
    s2: casex(x)
          2'b00:   next = s0;
          2'b11:   next = s1;
        endcase
    endcase
  always @(*)    // description of output functions
    case(state)
    s0: casex(x)
          2'b?0:   y = 3'b000;
          2'b?1:   y = 3'b001;
        endcase
    s1: casex(x)
          2'b01:   y = 3'b011;
          2'b10:   y = 3'b010;
        endcase
    s2: casex(x)
          2'b00:   y = 3'b111;
          2'b11:   y = 3'b100;
        endcase
    endcase
endmodule
```

In Listing 1, the first process (the ‘always’ procedure) defines the state register, the second process defines the transition functions, and the third process defines the output functions of the FSM. Here, the transition and output functions are described using ‘case’ statements at two levels. The first-level ‘case’ statements check the *state* variable (the present state code) and, depending on its value, determine the necessary actions in each state. The second-level ‘case’ statements check the input vector *x* and, depending on its value, calculate the value of the *next* variable (the next state code) in the second process and the value of the *y* variable (the value of the output vector) in the third process. Considering that ‘case’ and ‘if’ statements are interchangeable in Verilog HDL, the ‘if-else-if’ chain can be used instead of the ‘case’ statement throughout the proposed approach. For clarity, the ‘case’ statement is used in this work.

Note that in our FSM, transitions from states s1 and s2 are not defined for all possible combinations of input variables. This is often encountered in practice, for example, in MCNC benchmarks [28]. Therefore, the Synthesizer of the design tool automatically inserts latches into the FSM circuit to preserve the previous values of the variables *next* and *y*. As a result, the area of the FSM increases significantly, and its performance decreases.

D. Features of describing FSMs in verilog of the Quartus design tool

Automatic insertion of latches into the FSM circuit can be avoided by explicitly defining the values of the variables *next* and *y* for all possible values of the input variables (input vectors). For example, for the variable *next* in second-level ‘case’ statements, the ‘default’ construct can be used to define the transition to the state in which this transition begins. This is expressed in the behavior of the FSM as follows. When an input vector that does not initiate transitions from the present state is received at the input of the FSM, the FSM remains in the present state.

However, this style is not suitable for describing the output functions of Mealy FSMs, since different output vectors can be formed for varying transitions from a given state. Therefore, latches can be used to preserve the last value of variable *y*.

The situation is more complicated when determining the default value for first-level ‘case’ statements that check the value of the *state* variable. The fact is that the Synthesizer does not consider *state* variable values that are not explicitly defined [29]. In other words, the Synthesizer assumes that the FSM will never intentionally transition to an illegal state. Therefore, to detect an illegal state, the project code should be considered not as a description of an FSM, but as a description of an arbitrary project, i.e., the descriptions of transition and output functions should be considered as descriptions of arbitrary combinational circuits. In the Quartus design tool, this can be done by setting the synthesis option ‘Extract Verilog State Machine’ to ‘Off’.

Another feature of the Verilog language is the way default values are defined in ‘case’ statements. The standard solution is to use the ‘default’ construct of the ‘case’ statement, for example:

```
case (state)
  s0: … next = s1;
  …
  s2: …
  default: next = s0;
endcase
```

However, preliminary research has shown that a much more effective way in terms of reducing area and increasing performance is to use a blocking assignment statement (‘=’) to determine the default value, which is placed before the ‘case’ statement in the ‘begin-end’ block, for example:

```
begin
  next = s0;
  case (state)
    s0: … next = s1;
    …
    s2: …
  endcase
end
```

In Mealy FSMs, the outputs are asynchronous with respect to the inputs. Therefore, undesirable output vectors may appear at the outputs of Mealy FSMs. To avoid this, registers can be installed at the outputs of the FSM circuit. To do this, instead of the ‘always(*)’ construct, it is sufficient to use the ‘always(posedge clk)’ construct in the output function description. However, installing registers at the outputs of an FSM will introduce a one-clock-cycle delay in the formation of output signals, i.e., a Mealy FSM will function as a Moore FSM.

E. Description of SFSMs in Verilog HDL

The description of the SFSM from our example in the *safe* style is given in Listing 2.

Listing 2. Description of the SFSM in the *safe* style

```
module Mealy_safe
  … // declaration of ports as in Listing 1
  … // description of state variables, FSM state,
  … // and state register as in Listing 1
  always @(*)     // description of transition functions
  begin
    next = s0;      // default value of variable next
    case(state)
    s0: casex(x)
          2'b?0:   next = s0;
          2'b?1:   next = s1;
          default: next = s0;
        endcase
    s1: casex(x)
          2'b01:   next = s0;
          2'b10:   next = s2;
          default: next = s1;
        endcase
    s2: casex(x)
          2'b00:   next = s0;
          2'b11:   next = s1;
```

```
            default: next = s2;
          endcase
      endcase
    end
    always @(*)     // description of output functions
    begin
      y = 0;     // default value of variable y
      … // a case statement with descriptions
      … // of the output functions as in Listing 1
    end
  endmodule
```

In Listing 2, the default values for the variables *next* and *y* are defined using blocking assignment statements before the 'case' statement in the 'begin-and' block of the 'always' procedure. This allows the FSM, when the 'Extract Verilog State Machine' parameter is set to 'Off', to return from illegal states to the initial state s0 with a zero vector at the output.

Listing 3 shows a description of the SFSM from our example in the *safe_error* style.

Listing 3. Description of the SFSM in the *safe_error* style.

```
  module Mealy_safe_error ( // declaration of ports
    input clk, reset,            // clock and reset signals
    input [1:0] x,               // input vector
    output wire error,           // error signal
    output reg [2:0] y);         // output vector
    … // description of state variables, FSM state,
    … // and state register as in Listing 1
    … // description of transition functions as in Listing 2
    … // description of output functions as in Listing 2
    assign error =               // generating the error signal
      (state == s0
      || state == s1
      || state == s2)
      ? 1'b0 : 1'b1;
  endmodule
```

Unlike Listing 2, the *Mealy_safe_error* project port list includes an *error* output signal of type 'wire', which is generated using the continuous assignment statement 'assign'. The need to use the statement 'assign' to generate the *error* signal is due to the asynchronous nature of the outputs of Mealy FSMs. When generating the *error* signal in the traditional way in the 'case' statement, the *error* signal may be falsely set to 1. This can be avoided by using the 'assign' statement, as shown in Listing 3.

Listing 4 presents a description of the SFSM from our example in the *safe_idle* style.

Listing 4. Description of the SFSM in the safe_idle style.

```
  module Mealy_safe_idle (  // declaration of ports
    input clk, reset,            // clock and reset signals
    input [1:0] x,               // input vector
    output reg error,              // error signal
    output reg [2:0] y);           // output vector
    localparam [1:0] s0 = 0, s1 = 1, s2 = 2;   // FSM states
    localparam [1:0] idle = 3;   // idle state
    … // description of state variables and state register
    … // as in Listing 1
    always @(*)     // description of transition functions
    begin
      next = idle;    // default value of next variable
      case(state)
        s0: casex(x)
              2'b?0:   next = s0;
              2'b?1:   next = s1;
              default: next = s0;
            endcase
        s1: casex(x)
              2'b01:   next = s0;
              2'b10:   next = s2;
              default: next = s1;
            endcase
        s2: casex(x)
              2'b00:   next = s0;
              2'b11:   next = s1;
              default: next = s2;
           endcase
        idle:       next = s0;
      endcase
    end
    always @(*)     // description of output functions
    begin
      error = 1'b0;  y = 0;   // default values of error and y
      case(state)
        s0: casex(x)
              2'b?0:   y = 3'b000;
              2'b?1:   y = 3'b001;
           endcase
        s1: casex(x)
              2'b01:   y = 3'b011;
              2'b10:   y = 3'b010;
           endcase
        s2: casex(x)
              2'b00:   y = 3'b111;
              2'b11:   y = 3'b100;
           endcase
        idle: begin y = 0; error = 1'b1; end
      endcase
    end
  endmodule
```

Unlike Listing 2, the *Mealy_safe_idle* project port list includes the *error* signal, and the *idle* state is added to the FSM state set. In the transition function description, the default transition to the *idle* state is implemented by assigning the variable *next* the value *idle* using a blocking assignment statement, placed before the 'case' statement. Similarly, in the description of the output functions, default values are defined for the *error* signal and the variable *y*.

The results of implementing the FSM from our example using the Quartus Prime version 24.1 design tool in a Cyclone 10 LP family FPGA for different description styles are shown in Table II, where L is the number of look-up tables (LUTs) in the FSM circuit (area); F is the maximum operating frequency of the FSM in megahertz (performance).

TABLE II
PARAMETERS OF SFSM PROJECTS FOR OUR EXAMPLE

FSM project	L	F	Warning
Mealy_3proc	13	252	30: 10270, 10240, 13012
Mealy_safe_Quartus	15	905	30: 10270, 10240, 13012
Mealy_safe	5	1256	4: 10270
Mealy_safe_error	6	1241	4: 10270
Mealy_safe_idle	6	1241	2: 10270

Table II also includes the parameters of the *Mealy_3proc* project, when the FSM is described traditionally with three processes, as well as the parameters of the *Mealy_safe_Quartus* SFSM project, which is generated by the Quartus design tool when the 'Safe State Machine' option of Synthesizer is set to 'On'. Table II (in the Warning column) shows the number of messages and their numbers generated by the Synthesizer of the Quartus design tool, where message 10270 indicates an incompleteness of the 'case' statement, and messages 10240 and 13012 are related to the automatic insertion of latches into the FSM circuit.

Table II shows that the proposed styles for describing SFSMs significantly outperform the *Mealy_safe_Quartus* project in terms of area and performance, as well as the traditional FSM description (the *Mealy_3proc* project). The reduction in area and increase in performance in the proposed styles of describing SFSMs compared to the traditional description is explained by the use of a blocking assignment statement to determine the default values for the variables *next* and *y* before the 'case' statement in the 'begin-end' block of the 'always' procedure.

The optimization of combination circuits λ and δ in the proposed styles is explained by the fact that the Synthesizer of the design tool generates only the logic that is explicitly defined in the 'case' statements, since the implicit value has already been described earlier using the blocking assignment statement ('=') in the corresponding 'begin-end' block before the first-level 'case' statement. As a result, the logic of combinational circuits λ and δ is simplified, the area is reduced, and the performance of the FSM is increased.

IV. RESULTS AND DISCUSSION

This section presents the results of experimental studies of the proposed styles of describing SFSMs using FSM benchmarks [28], as well as recommendations for the practical use of the proposed styles of describing SFSMs.

A. Results of experimental studies

The effectiveness of the *safe, safe_error*, and *safe_idle* styles for describing SFSMs was compared with the traditional description of FSMs (*3proc* style) and SFSMs generated by the Quartus design tool when the 'Safe State Machine' option is set to 'On' (*safe_Q* style). The experiments were conducted on FSM benchmarks from the Microelectronics Center of North Carolina (MCNC) [28] using the Quartus Prime design tool version 24.1 to implement FSMs in the Cyclone 10 LP FPGA family. When using the *safe, safe_error*, and *safe_idle* styles, the 'Extract Verilog Machine' synthesis option was set to 'Off' to enable the detection of illegal states in FSMs. The area of FSMs was measured by the number of look-up tables (LUTs) used, and the maximum operating frequency was used to determine performance.

The results of experimental studies are presented in Tables III and IV, where *L3p, Lsq, Ls, Le*, and *Li* are the areas of FSMs when using the styles *3proc, safe_Q, safe, safe_error*, and *safe_idle*, respectively; *Lmin* is the minimum area value achieved by one of the styles *safe, safe_error*, or *safe_idle*; *F3p, Fsq, Fs, Fe*, and *Fi* are the performance (in megahertz) of FSMs when using the styles *3proc, safe_Q, safe, safe_error*, and *safe_idle*, respectively; *Fmax* – maximum performance value achieved by one of the *safe, safe_error*, or *safe_idle* styles; *L3p/Lmin, Lsq/Lmin, Fmax/F3p*, and *Fmax/Fsq* – ratios of the corresponding parameters; *Av* – arithmetic mean value of the parameter; *Max* – maximum value of the parameter.

Table III illustrates that the use of the proposed styles for describing SFSMs results in an average decrease in FSM area by a factor of 1.793 (or 79.3%) compared to the traditional description method (parameter *L3p/Lmin*), and by a factor of 2.436 when compared to the Quartus design tool method (parameter *Lsq/Lmin*). Notably, for the train4 example, the area was minimized by as much as 5.333 times compared to the traditional description and by 6.667 times compared to the Quartus design tool method.

TABLE III
AREA OF FSMS

FSM	*3proc*	*safe_Q*	*safe*	*safe_ error*	*safe_ idle*	*Lmin*	*L3p/ Lmin*	*Lsq/ Lmin*
	L3p	*Lsq*	*Ls*	*Le*	*Li*			
bbara	26	37	32	34	32	32	0.813	1.156
bbtas	8	13	6	7	7	6	1.333	2.167
beecount	36	45	19	21	21	19	1.895	2.368
cse	96	121	71	79	79	71	1.352	1.704
dk14	42	52	34	34	34	34	1.235	1.529
dk15	28	33	15	15	15	15	1.867	2.200
dk16	63	101	70	70	76	70	0.900	1.443
dk17	21	29	13	13	13	13	1.615	2.231
dk27	8	15	5	6	6	5	1.600	3.000
dk512	18	33	15	16	16	15	1.200	2.200
ex1	119	152	131	132	134	131	0.908	1.160
ex2	64	94	45	40	51	40	1.600	2.350
ex3	34	43	19	20	20	19	1.789	2.263
ex4	44	67	36	37	40	36	1.222	1.861
ex5	32	43	14	15	18	14	2.286	3.071
ex6	57	69	55	55	52	52	1.096	1.327
ex7	32	48	19	20	26	19	1.684	2.526
keyb	64	81	80	83	72	72	0.889	1.125
lion	15	18	3	3	3	3	5.000	6.000
lion9	47	62	13	14	15	13	3.615	4.769
mc	9	12	6	6	6	6	1.500	2.000
pma	114	167	105	104	110	104	1.096	1.606
s208	52	72	35	37	26	26	2.000	2.769
s27	20	27	7	6	12	6	3.333	4.500
s298	611	799	291	357	885	291	2.100	2.746
s386	39	57	57	55	59	55	0.709	1.036
s420	38	52	35	36	22	22	1.727	2.364
s510	69	128	76	76	75	75	0.920	1.707
s820	87	121	119	126	132	119	0.731	1.017
sand	208	267	164	164	160	160	1.300	1.669
shiftreg	9	14	3	3	3	3	3.000	4.667
styr	183	237	187	189	171	171	1.070	1.386
tav	10	10	9	9	9	9	1.111	1.111
tbk	246	285	86	86	94	86	2.860	3.314
tma	128	173	78	79	78	78	1.641	2.218
train11	38	55	19	20	22	19	2.000	2.895
train4	16	20	6	3	3	3	**5.333**	**6.667**
Av	73.811	98.703	53.459	55.946	70.189	51.676	**1.793**	**2.436**
Max							**5.333**	**6.667**

TABLE IV
AREA OF FSMS

FSM	3proc F3p	safe_Q Fsq	safe Fs	safe_error Fe	safe_idle Fi	Fmax	Fmax/F3p	Fmax/Fsq
bbara	539	350	483	458	459	483	0.896	1.380
bbtas	937	534	912	916	900	916	0.978	1.715
beecount	251	542	627	632	641	641	2.554	1.183
cse	174	273	509	457	457	509	2.925	1.864
dk14	541	431	579	557	556	579	1.070	1.343
dk15	651	637	885	895	895	895	1.375	1.405
dk16	583	278	398	393	397	398	0.683	1.432
dk17	637	455	631	652	652	652	1.024	1.433
dk27	937	528	922	931	931	931	0.994	1.763
dk512	726	359	660	639	639	660	0.909	1.838
ex1	258	300	384	392	351	392	1.519	1.307
ex2	224	280	348	584	419	584	2.607	2.086
ex3	240	368	529	538	552	552	2.300	1.500
ex4	221	364	534	483	477	534	2.416	1.467
ex5	245	406	640	638	579	640	2.612	1.576
ex6	197	359	519	521	539	539	2.736	1.501
ex7	214	349	504	497	488	504	2.355	1.444
keyb	455	315	348	342	381	381	0.837	1.210
lion	299	595	1305	1302	1302	1305	4.365	2.193
lion9	149	452	596	633	550	633	4.248	1.400
mc	1250	720	831	823	823	831	0.665	1.154
pma	86	247	343	276	320	343	3.988	1.389
s208	461	280	428	428	574	574	1.245	2.050
s27	537	471	815	824	644	824	1.534	1.749
s298	318	159	161	144	207	207	0.651	1.302
s386	524	330	430	428	434	434	0.828	1.315
s420	527	307	435	407	621	621	1.178	2.023
s510	917	232	341	362	365	365	0.398	1.573
s820	426	269	333	315	317	333	0.782	1.238
sand	173	213	334	336	355	355	2.052	1.667
shiftreg	1078	482	1321	1319	1319	1321	1.225	**2.741**
styr	166	239	287	323	324	324	1.952	1.356
tav	1259	850	1280	1276	1276	1280	1.017	1.506
tbk	325	254	361	375	348	375	1.154	1.476
tma	139	273	341	358	396	396	2.849	1.451
train11	295	370	580	591	524	591	2.003	1.597
train4	263	609	1093	1302	1302	1302	**4.951**	2.138
Av	465.5	391.4	595.3	604.0	603.1	627.1	**1.834**	**1.588**
Max							**4.951**	**2.741**

Table IV demonstrates that the use of the proposed styles for describing SFSMs results in an average improvement in FSM performance by a factor of 1.834 (or 83.4%) compared to traditional descriptions (parameter *Fmax/F3p*), and by a factor of 1.588 (or 58.8%) when compared to the Quartus design tool method (parameter *Fmax/Fsq*). Notably, the performance increased by as much as 4.951 times for the train4 example compared to traditional descriptions, and by 2.741 times for the shiftreg example compared to the Quartus design tool method.

Section III outlined the reasons for the reduction in area and increase in performance of FSMs resulting from the use of the proposed SFSM description styles compared to the traditional description and the Quartus design tool method.

Figure 6 shows the average area, and Fig. 7 shows the average performance for the considered styles of FSM description.

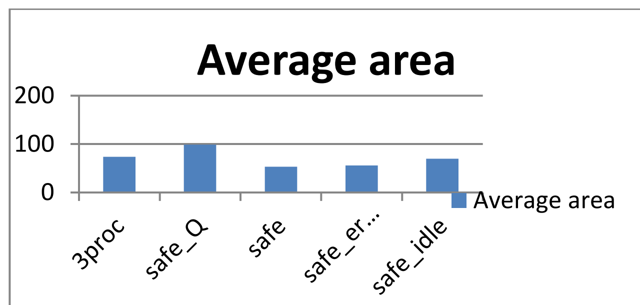

Fig. 6. Average area for the considered styles of FSM description.

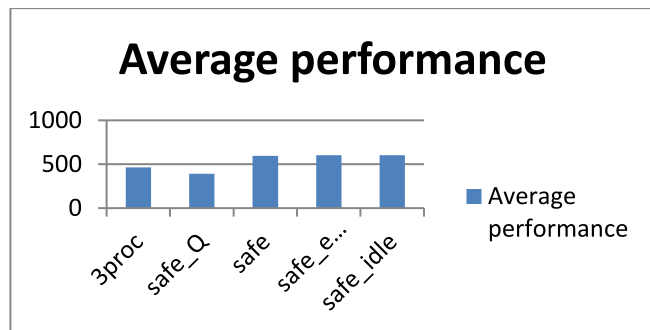

Fig. 7. Average performance for the considered styles of FSM description.

Fig. 6 illustrates that the most significant reduction in FSM area is achieved with the *safe* style, followed by the *safe_error* and *safe_idle* styles. Fig. 7 demonstrates that the performance of FSMs when using the *safe, safe_error*, and *safe_idle* styles is approximately the same.

Thus, experimental studies have shown that the proposed styles of describing SFSMs significantly surpass traditional descriptions of FSMs in terms of area and performance, as well as the method of synthesizing SFSMs implemented in the Quartus design tool.

B. Recommendations for the practical application of SFSM description styles

To encode the states of SFSMs, it is necessary to use codes of minimum length, such as binary codes or Gray codes. This practice minimizes the number of invalid states and reduces the probability of the FSM transitioning to an invalid state.

The *safe* style can be used to describe SFSMs in which a zero output vector is not formed during any FSM transition. In this case, a zero output vector can indicate that the FSM has entered an illegal state. Otherwise, the *safe_error* style should be used. In the *safe_error* style, a value of one on the additional output signal *error* indicates that the FSM is in an illegal state. This style is better suited for Moore FSMs, since in Mealy FSMs, the time interval when the *error* signal is equal to 1 can be very short.

The *safe_idle* style should be used when it is necessary to perform additional actions in the *idle* state to reduce the negative consequences of the FSM entering an illegal state. In this case, the *idle* state can generate values for additional output signals or a special output vector.

V. CONCLUSION

This paper discusses three types and their corresponding three styles of describing SFSMs in Verilog HDL: *safe*, *safe_error*, and *safe_idle*. The *safe* style corresponds to the traditional implementation of SFSMs, where, when an FSM enters an illegal state, it returns to its initial state. In the *safe_error* style, an additional output *error* signal is generated, with a value of 1 indicating that the FSM has entered an illegal state. The *safe_idle* style provides for a transition to an additional *idle* state before the FSM returns from an illegal state to its initial state. If necessary, the *idle* state can be used to perform actions that neutralize the FSM when it enters an illegal state.

Experimental studies using FSM benchmarks demonstrated the high efficiency of the proposed SFSM description styles. Compared with the SFSM synthesis method in the Quartus design tool, the FSM area is reduced on average by a factor of 2.436 (6.667 times for individual examples), while performance is increased on average by a factor of 1.588 (2.741 times for individual examples).

Future research will aim to develop structural models of safe, robust, and fault-tolerant FSMs.

REFERENCES

[1] S. Park, H. T. Kim, S. Lee, H. Joo, and H. Kim, “Survey on anti-drone systems: Components, designs, and challenges,” *IEEE Access*, vol. 9, pp. 42635-42659, 2021, https://doi.org/10.1109/access.2021.3065926

[2] S. Yin, B. Xiao, S. X. Ding, and D. Zhou, “A review on recent development of spacecraft attitude fault tolerant control system,” *IEEE Transactions on Industrial Electronics*, vol. 63. no. 5, pp. 3311-3320, 2016, https://doi.org/10.1109/TIE.2016.2530789

[3] Intel Quartus Prime Standard Edition User Guide. Design Compilation. 3.4.12. Safe State Machine. 2021. https://www.intel.com/content/www/us/en/docs/programmable/683283/18-1/safe-state-machine.html

[4] Xilinx. Vivado Design Suite User Guide: Synthesis; UG901 (v2019. 1), 2022. https://docs.amd.com/r/en-US/ug901-vivado-synthesis/FSM_SAFE_STATE

[5] C. E. Cummings and H. Chambers, “Finite State Machine (FSM) Design & Synthesis Using SystemVerilog—Part I,” In Proceedings of Synopsys Users Group (SNUG), San Jose, CA, USA, 2019, pp. 1-77.

[6] W. Hu, C. H. Chang, A. Sengupta, S. Bhunia, R. Kastner, and H. Li, „An overview of hardware security and trust: Threats, countermeasures, and design tools,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 6, pp. 1010-1038, 2020, https://doi.org/10.1109/TCAD.2020.3047976

[7] R. A. Brooks, “A Robust Layered Control System for a Mobile Robot,” *IEEE Journal of Robotics and Automation*, vol. 2, no. 1, pp. 14-23, 1986, https://doi.org/10.1109/JRA.1986.1087032

[8] J. S. Baras and M. R. James, “Robust and risk-sensitive output feedback control for finite state machines and hidden Markov models,” Tech. Rep. T.R. 94-63, Institute for Systems Research, University of Maryland, Maryland, MD, USA, August, 1994.

[9] D. Mange, M. Sipper, A. Stauffer, and G. Tempesti, “Toward robust integrated circuits: The embryonics approach,” *Proceedings of the IEEE*, vol. 88, no. 4, pp. 516-543, 2002, d https://doi.org/10.1109/5.842998

[10] A. Megretski, “Robustness of finite state automata,” In *Multidisciplinary Research in Control: The Mohammed Dahleh Symposium*, Berlin, Germany, 2002, pp. 147-160, https://doi.org/10.1007/3-540-36589-3_12

[11] D. C. Tarraf, M. A. Dahleh, and A. Megretskr, “Stability of deterministic finite state machines,” In *Proceedings of the 2005, American Control Conference*, Portland, OR, USA, 2005, pp. 3932-3936), https://doi.org/10.1109/ACC.2005.1470590

[12] A. Cui, C. H. Chang, S., Tahar, and A. T. Abdel-Hamid, “A robust FSM watermarking scheme for IP protection of sequential circuit design,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 5, pp. 678-690, 2011, https://doi.org/10.1109/TCAD.2010.2098131

[13] V. Bychko, R. Yershov, Y. Gulyi, and M. Zhydko, “Automation of anti-race state encoding of asynchronous FSM for robust systems,” In: *IEEE International Conference on Problems of Infocommunications. Science and Technology (PIC S&T)*, Kharkiv, Ukraine, 2020, pp. 501-506, https://doi.org/10.1109/PICST51311.2020.9467908

[14] Y. Yan, P. Xu, J. Yue, and Z. Chen, “Robust control: from continuous-state systems to finite state machines,” *IEEE Transactions on automation science and engineering*, vol. 21, no. 2, pp. 2156-2163, 2024, https://doi.org/10.1109/TASE.2024.3362975

[15] X. Zhang, Y. Wang, B. Xu, and G. Liu, “Robust Liveness Controllers for Time Petri Nets With Unreliable Resources,” In *IEEE 20th International Conference on Automation Science and Engineering (CASE)*, Bari, Italy, 2024, pp. 3400-3405, https://doi.org/10.1109/CASE59546.2024.10711796

[16] D., Sokolov, V., Khomenko, and M. Sautto, “Easy async for busy engineers: AFSM-based design of low-latency robust controllers in Workcraft,” In *29th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, Portland, OR, USA, 2025, pp. 63-64, https://doi.org/10.1109/ASYNC65240.2025.00017

[17] L. Dong and J. Shen, “Robust control of uncertain vector discrete event systems,” In *Fifth World Congress on Intelligent Control and Automation (IEEE Cat. No. 04EX788)*, Hangzhou, China, 2004, vol. 2, pp. 1051-1055, https://doi.org/10.1109/WCICA.2004.1340771

[18] S. Abdelwahed, G. Karsai, and G. Biswas, “Robust state-based supervisory control of discrete event systems,” In *Proceedings of 2005 IEEE Conference on Control Applications (CCA)*, Toronto, ON, Canada, 2005, pp. 922-927, https://doi.org/10.1109/CCA.2005.1507247

[19] M. Karimadini, A. Karimoddini, and A. Homaifar, “A survey on fault-tolerant supervisory control,” In *IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS)*, Windsor, ON, Canada, 2018, pp. 733-738, https://doi.org/10.1109/MWSCAS.2018.8624110

[20] R. Meira-Góes, S. Lafortune, and H. Marchand, “Synthesis of supervisors robust against sensor deception attacks,” *IEEE Transactions on Automatic Control*, vol. 66, no. 10, pp. 4990-4997, 2021, https://doi.org/10.1109/TAC.2021.3051459

[21] K. Liszewski and T. McDonley, “Understanding tool synthesis behavior and safe finite state machine design,” *arXiv preprint arXiv:2108.04042*. 2021.

[22] Y. L. Chen and F. Lin, “Safety control of discrete event systems using finite state machines with parameters,” In *Proceedings of the American Control Conference.(Cat. No. 01CH37148)*, Arlington, VA, USA, 2001, vol. 2, pp. 975-980, https://doi.org/10.1109/ACC.2001.945847

[23] L. Ouedraogo, R. Kumar, R. Malik, and K. Akesson, “Nonblocking and safe control of discrete-event systems modeled as extended finite automata,” *IEEE Transactions on Automation Science and Engineering*, vol. 8, no. 3, pp. 560-569, 2011, https://doi.org/10.1109/TASE.2011.2124457

[24] S. R. Anreddy, K. S. Charan, N. N. Sai, and N. Panda, “Modelling and Design of a Finite State Machine Based Elevator Control System for Efficient Vertical Transportation,” In *5th IEEE Global Conference for Advancement in Technology (GCAT)*, Bangalore, India, 2024, pp. 1-7, https://doi.org/10.1109/GCAT62922.2024.10924000

[25] X. Song, L. Qi, S. Liu, S. Ding, and D. Li, “Simple analysis of complex system safety based on Finite State Machine Network and phase space theory,” *Reliability Engineering & System Safety*, vol. 249, pp. 110205, 2024, https://doi.org/10.1016/j.ress.2024.110205

[26] F. Barbanera and R. Hennicker, “Safe Composition of Systems of Communicating Finite State Machines,” arXiv preprint arXiv: 2412.08234, 2024, https://doi.org/10.4204/eptcs.414.3

[27] N. Zhao, F. Wu, Y. Zhang, and W. Wang, “Process safety modeling and analysis method based on finite state machine,” *Journal of Physics: Conference Series*, vol. 2977, no. 1, pp. 012110, March 2025, https://doi.org/10.1088/1742-6596/2977/1/012110

[28] S. Yang, “Logic Synthesis and Optimization Benchmarks User Guide. Version 3.0,” Microelectronics Center of North Carolina (MCNC): Research Triangle Park, NC, USA, 1991.

[29] Quartus PrimePro Edition User Guide: Design Recommendation. Section 1.6.4. Machine HDL Guidelines. 2025. https://www.bing.com/search?q=ug-683082-850788&form=ANNTH1&refig=6950f44eb4614366adbd581c03101b20&pc=U531