Hardware Description of Event-driven Systems by Translation of UML Statecharts to VHDL

Cristinel Ababei and Susan C. Schneider
Electrical and Computer Engineering Dept.
Marquette University, Milwaukee, WI, USA
Email: {cristinel.ababei,susan.schneider}@marquette.edu

***Abstract*—We present a complete implementation prototype of the classic Fly-n-Shoot game on an FPGA. This is a famous game that has been described in the past using UML statecharts as an event-driven embedded system. Because it has a rather complex functionality, attempting to describe it using a hardware description language (HDL), such as VHDL or Verilog, with the goal of deploying on a real FPGA becomes challenging. As such, brute-force attempts to write HDL descriptions are prone to errors and subject to long design times. Hence, in this paper, we describe a practical approach for translating UML statecharts used to specify event-driven embedded systems into VHDL code written using the popular two-process coding style. This approach consists of a set of mapping rules from statecharts concepts into VHDL constructs. The efficacy and correct by design characteristics of the presented approach are due to the use of two-process VHDL coding to describe the hierarchical finite state machine (FSM) corresponding to the UML statecharts. This gives the designer better control over the current and next state signals of the FSMs, it is more modular or object oriented, and makes development and debugging much easier. We apply the proposed approach to implement a prototype of the classic Fly-n-Shoot game. The implementation is verified successfully on real hardware, the DE1-SoC FPGA development board, that uses a Cyclone IV FPGA chip.**

***Index Terms*—UML statecharts; VHDL code; FPGA prototyping; event-driven embedded systems;**

I. Introduction

In coding event-driven systems - which must react to incoming events in a timely fashion - the main challenge is to identify the correct actions to execute in response to a given event [1]. The challenging aspect of that stems from the fact that actions are determined by: 1) nature of the event and 2) current context, i.e., the history of events of the system. It is the *context* factor that is neglected or handled poorly by traditional programming approaches, which can result in multi-level if statements and convoluted or "bowl of spaghetti" code. State machines provide one of the most popular formalisms for specification and implementation of event-driven systems. That is because event handling is explicitly dependent on both the context (i.e., state) of the system and on the nature of the event [2].

Event-driven embedded systems are traditionally implemented on processors/microcontrollers and therefore they would typically be programmed in embedded C/C++ starting from unified modeling language (UML) statecharts [2]. Translating or generating code in various programming languages from UML statecharts has received a lot of attention and many techniques and tools already exist. However, if such systems are to be implemented on FPGAs, their description is done in a hardware description language (HDL) such as VHDL or Verilog. Converting statecharts to HDL code is considered an ongoing research effort because existing conversion tools still suffer from limitations [3].

II. Related Work

Previous literature can be generally classified into approaches that use automated techniques and approaches that are manual and require direct translation of UML statecharts into HDL code. Here, we discuss relevant previous work that also focused on translating UML statecharts into VHDL or Verilog descriptions. For example, the study in [4] presented HiCoS (hierarchical concurrent system) tool aimed at converting UML statecharts to RTL-VHDL for real time digital controllers. The study in [5] proposed a model-driven development (MDD) technique as a set of rules to automatically generate synthesizable VHDL code from UML notations. They used metamodels of the two languages and a transformation between the two metamodels. As a use case, they studied an early warning system (EWS). Their approach uses one-process coding approach, which tends to become one large process, actually possibly suffering from a lack of clarity. In contrast, in this paper, we advocate for the two-process VHDL coding approach [6], which gives the designer a better control over the current and next state signals, it is more modular or object oriented, and makes debugging easier. In addition, we directly convert or translate UM statecharts into two-process FSM or FSMD descriptions and do not need intermediate representations.

The study in [7], presented an approach integrated as an extension into GenERTiCA tool to automatically generate VHDL code from UML specifications. Their use case was a distributed embedded system used for maintenance systems. The extension consists of a new set of mapping rules to map UML metamodel elements to VHDL constructs. Our work is similar in that we use a direct mapping as well, where events are translated into entity ports and signals. However, our target VHDL code implements the two-process coding style and our use case is a more complex one, in addition to implementing a VGA driver as well. The work in [8] presented a methodology that generates a VHDL model with checkers from UML sequence diagrams with MARTE timing

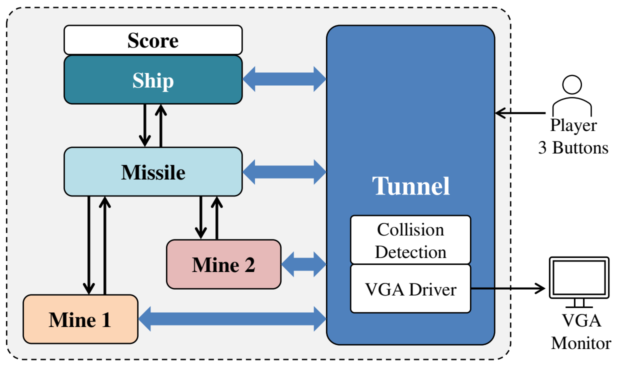

Figure 1. System level diagram of the Fly-n-Shoot game.

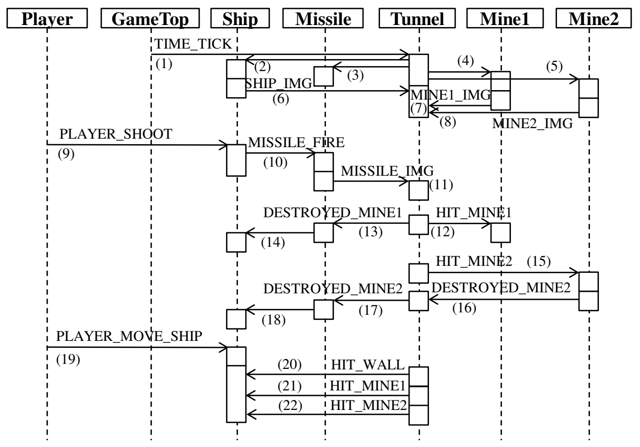

Figure 2. Sequence diagram of the Fly-n-Shoot game - showing the high-level interaction between the main objects.

constraints. The proposed methodology is based on a set of mapping rules between sequence diagrams and VHDL constructs. Their use case was a wireless sensor node.

The study in [9] presented a method to translate UML diagrams to Verilog, by first converting UML into hierarchical concurrent finite state machines (HCFSMs) as an intermediate model. This method was presented for the description of logic controllers as a use case. The work in [10] presents a synthesis methodology to automatically generate RTL VHDL/Verilog code from UML diagrams. The methodology employs CubedC tools to transform UML diagrams into ADA code, which in turn is transformed into an intermediate predicate format (IPF) database, and finally into VHDL/Verilog code. Simulations were presented for several use cases including an FIR filter and MPEG 4th routine. The study in [3] proposed a framework that uses Yakindu statechart tools (Yakindu SCT) as a graphical interface to input UML statecharts that are then converted into XML files, which then are parsed with Xerces-C++ parser to produce a VHDL file. However, to use case has been demonstrated. More recently, the study in [11] presented an automatic tool, which can generate VHDL code almost as efficient as human-written code from a variant of the state chart XML (SCXML) standard tailored to hardware systems. However, the produced VHDL code may be verbose and difficult to debug. Finally, a significant challenge with all these previously reported tools is that they not made publicly available - hence, they cannot be used on new designs or for comparison purposes.

III. Use case: Fly-n-Shoot Game

The use case studied in this paper is the classic Fly-n-Shoot game, whose system level diagram is shown in Fig. 1. We briefly describe it here because we will refer to it in the next sections to aid our presentation. The primary objects are the Ship, which flies inside a Tunnel with walls, the Missile that is fired from the Ship, and two mines, Mine1 and Mine2. While in this paper we focus on a number of two mines only, this number can be changed to a larger value. Our theoretical presentation would not change for different numbers of mine objects. The interactions between these objects and all possible paths through the interactions are shown in the sequence diagram from Fig. 2. The Ship flies at constant speed and the Player can move it up or down. It can fire only one missile, which also flies at a higher speed until it hits the Wall of the Tunnel, or any of the two mines, or it gets outside the screen area. Then, the Missile becomes armed and the Ship can fire it again. Mines are planted at random locations within the Tunnel. When the Missile hits mine type 1, Mine1, the mine explodes and tells the Ship via the Missile to increase the score with 20 points because it destroyed a mine. Mine2 needs to be hit twice by the Missile in order to destroy it, in which case the score is increased by 50 points. As shown in Fig. 2, the GameTop (the top level design entity of the game) provides the clock signal $TIME\_TICK$ to the Tunnel, which then passes it to all other objects. The Tunnel is the most complex object in the system because it integrates the logic for collision detection (signaled via events HIT_WALL, HIT_MINE1, HIT_MINE2) and a VGA graphics driver, which allows for all the game action to be displayed on the attached VGA monitor.

IV. UML Statecharts

Developed in the nineties [12], the Unified Modeling Language (UML) is a "*specification defining a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems*" [13] and a standard maintained by the Object Management Group (OMG). UML is a collection of various description techniques that can be used to model different aspects of software systems. It can be regarded as a formalization of diagrams and sketches of software systems, hand-drawn by software developers or programmers to describe and assess ideas before actual implementation. UML is a pictorial or visual language; it is a *specification* language rather than a *programming* language like for example C/C++.

However, UML alone suffers from ambiguous interpretation because it lacks consistency in describing the dynamics of a system; it lacks a formal semantics. And that is where *statecharts* come in to provide a semantics foundation based on a mathematical formalism. Statecharts is one of the two different kinds of state machine formalisms that UML provides; the other one being activity diagrams. Statecharts are a generalization of finite state machines (FSM - a type of models of computation), extended with constructs to support expression of concurrent processes and hierarchical organization of transition systems. They represent a visual formalism for describing states of communicating finite state machines as well as transitions in modular fashion [14]. Statecharts help to organize and suppress detail. It is important to note that one can view statecharts as both a model of computation (perhaps because of their underlying commonality with FSMs) and as a graphical specification language.

UML statecharts or state machines are improved versions, object-based variants of Harel statecharts [14] extended by UML. They mitigate limitations of traditional finite-state machines, expand the concept of actions, and introduce novel ideas of hierarchy, nested states, and orthogonal regions. They are used when transitions between states take place upon the occurrence of events of interest. The three basic elements of statecharts are: states, transitions, and actions. States represent distinct conditions of existence and they can last for certain periods of time. Transitions represent the mechanism of changing states in response to events of interest. Actions are executed upon entering or exiting a state or when the occurrence of an event triggers a transition. They are what is called atomic behaviors and can be for example simple statements or operations. As a example, we show in Fig. 3 the simplified UML statechart for the Ship object of the use case studied in this paper.

Translation of UML statecharts into code in different programming languages (e.g., C/C+, Java, SystemC, etc.) can be done manually/directly or by design automation tools [15]–[17]. An excellent example of such translation of UML statecharts into C/C++ code is demonstrated in [2], where *“the emphasis is on the role of UML state machines in practical, everyday programming rather than mathematical abstractions”*. Such translation or conversion is also supported by commercial tools. However, there has been less work done on translating UML statecharts into synthesizable VHDL or Verilog code.

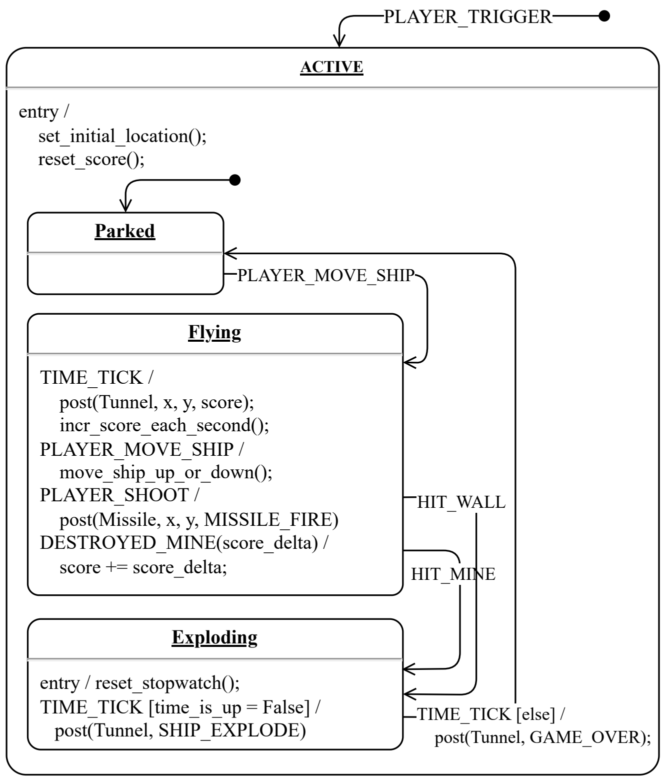

Figure 3. UML statechart for the Ship object from the Fly-n-Shoot game; adapted from [2].

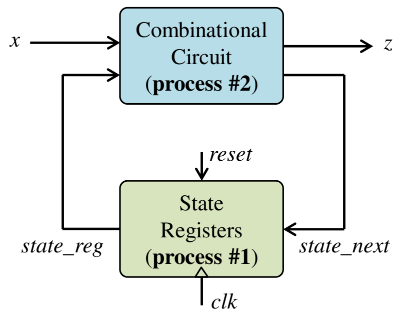
x
Combinational Circuit (process #2)
z
reset
State Registers (process #1)
state_reg
state_next
clk

Figure 4. Block diagram of the HW implementation of a Mealy FSM.

V. VHDL Description of FSMs Using Two Processes

We briefly describe the *two-process VHDL coding* approach because it is the approach that we advocate for and employ in this work. This approach is not new. Our contribution here is to identify it as the best approach to use in the process of translating UML statecharts to VHDL constructs; previous literature used single-process VHDL coding, which, while more compact is more complex and prone to errors. This VHDL coding style has been presented in [6] and its idea is to describe any finite state machine (FSM) using precisely two processes, which correspond to the *registers* (i.e., sequential) portion and the *combinational circuit* portion that synthesizes the primary output z and the next state $state\_next$ signals - from the hardware (HW) implementation of the FSM, which is generally described by a block diagram as shown in Fig. 4 for a Mealy type FSM (our discussion remains valid for Moore or combined Mealy and Moore type FSMs). The separation of the sequential and combinational portions of the FSM implementation allows the designer to completely control these portions, which in turn minimizes design errors.

The functionality of the two portions is captured by the two processes used in the VHDL description. For example, the two VHDL processes for the FSM describing the Ship object (from Fig. 3) are shown in Fig. 5 and Fig. 6. In Fig. 5, $process\ \#1$ models all necessary registered signals (i.e., denoted as $\_reg$ signals). In Fig. 6, $process\ \#2$ captures the

```
-- state register; process #1
process (TIME_TICK, reset)
begin
  if (reset = '1') then
    superstate_reg <= INACTIVE;
    state_reg <= Parked;
    x_reg <= (OTHERS=>'0');
    y_reg <= to_unsigned((MAX_Y-SHIP_HEIGHT)/2,10);
    score_reg <= (OTHERS=>'0');
    local_ctr_reg <= (OTHERS=>'0');
  elsif (TIME_TICK' event and TIME_TICK = '1') then
    superstate_reg <= superstate_next;
    state_reg <= state_next;
    x_reg <= x_next;
    y_reg <= y_next;
    score_reg <= score_next;
    local_ctr_reg <= local_ctr_next;
  end if;
end process;
```

Figure 5. VHDL code of process #1 that describes the sequential portion for the Ship object.

Table I
MAPPING FROM UML STATECHART ELEMENTS TO VHDL CONCEPTS.

UML Statechart Element	**VHDL Concept**
UML statechart	FSM or Hierarchical FSM *(coded in two-process VHDL style)*
Event	Signals, In/Out Ports
Default State	Initial State
History	Counters

logic that generates primary outputs (i.e., indicated as z in Fig. 4) and the next vaues of all registered signals, denoted as $\_next$ signals. The one to one correspondence between the two portions of the block diagram of the FSM and the two VHDL processes can be easily be seen in Fig. 4 and Fig. 5, Fig. 6. This correspondence gives this two-process VHDL coding approach the benefits of clarity and easy debugging.

VI. Rules to Translate UML Statecharts to VHDL

In this paper, a direct approach for translating UML statecharts into VHDL code is used. This approach essentially consists of a set of mapping rules between statecharts elements into VHDL constructs. While this approach does require a larger up-front effort than automated tools, it does not require intermediate formats used by automated tools. Also, a direct approach does not need third party libraries and gives the VHDL programer better control over the coding patterns and lower level details. Automated tools are usually restricted to using templates for conversion, which limits the amount of optimization of the VHDL code. In addition, most VHDL code generated automatically still requires some amount of manual intervention in order to integrate into actual practical designs. Below, we list the mapping rules, which are summarized in Table I. In this discussion, we refer to examples from the Fly-n-Shoot use case that we present in the results section.

Statechart to Hierarchical FSM. VHDL handles concurrency by design: the statecharts' property of having more than one state active at the same time (e.g, expressed using *AND* states) is naturally handled in VHDL through the concept of concurrent statements, which include processes as well.

```
-- next state and output logic; process #2
process (superstate_reg, state_reg, exp_ctr_reg,
  PLAYER_FIRE, PLAYER_SHIP_MOVE, TAKE_OFF,
  DESTROYED_MINE, HIT_MINE, HIT_WALL)
begin
  -- default initializations not shown
case superstate_reg is

    when INACTIVE => -- superstate
      if PLAYER_SHIP_MOVE = '1' then
        superstate_next <= ACTIVE;
      end if;

    when ACTIVE => -- superstate;
      case state_reg is

        when Parked =>
          -- place Ship at default, initial location
          x_next <= (OTHERS=>'0');
          y_next <= to_unsigned((MAX_Y-SHIP_HEIGHT)/2,10);
          if PLAYER_SHIP_MOVE = '1' then
            state_next <= Flying;
            score_next <= (OTHERS=>'0');
            SCORE <= '1'; -- generate event SCORE
            local_ctr_next <= (OTHERS=>'0');
          end if;

        when Flying =>
          if btn(1)='1' and
             (y_reg + SHIP_HEIGHT - 1)<(MAX_Y-1-SHIP_DELTA_V) then
            y_next <= y_reg + SHIP_DELTA_V; -- move down
          elsif btn(0)='1' and y_reg>SHIP_DELTA_V then
            y_next <= y_reg - SHIP_DELTA_V; -- move up
          end if;
          ship_flying<='1';
          SHIP_IMG <= '1'; -- generate event SHIP_IMG;
          local_ctr_next <= local_ctr_reg + 1;
          if (local_ctr_reg = 30)  then
            score_next <= score_reg + 1; -- increment score each sec
            SCORE <= '1'; -- generate event SCORE to Tunnel
            local_ctr_next <= (OTHERS=>'0');
          end if;
          if PLAYER_FIRE = '1' then
            MISSILE_FIRE <= '1'; -- generate event MISSILE_FIRE
          end if;
          if DESTROYED_MINE = '1' then
            -- add to local score the amount passed thru event;
            -- which depends on the type of mine destroyed;
            score_next <= score_reg + unsigned(score_inc_val);
          end if;
          if (HIT_MINE = '1' or HIT_WALL = '1') then
            state_next <= Exploding;
            exp_ctr_next <= (OTHERS=>'0');
            timer_2sec_start <='1'; -- start cowntdown counter 2 sec
          end if;

        when Exploding =>
          EXPLOSION_SHIP <= '1'; -- post event to Tunnel
          -- wait for 2 sec to display exploding ship
          if timer_2sec_up='1' then
            state_next <= Parked;
            GAME_OVER <= '1'; -- generate event to Tunnel
            superstate_next <= INACTIVE;
          end if;

      end case;
  end case;
end process;
```

Figure 6. VHDL code of process #2 that describes the combinational portion for the Ship object.

This makes VHDL (or Verilog) an intuitive target language to translate UML statecharts. Therefore, UML statecharts are mapped into simple FSMs or hierarchical FSMs. These FSMs are coded directly in VHDL, using the *two-process VHDL coding* style discussed earlier. Processes, as VHDL constructs, are concurrent statements, which by design support parallelism or concurrency of hardware. This concurrency allows us to translate an arbitrary number of UML statecharts to VHDL code; for example, in our use case we have several statecharts (i.e., Tunnel, Ship, Missile, Mine1, and Mine2). Hierarchical FSMs easily capture hierarchy in statecharts; for example, the Ship statechart from Fig. 3 can be translated into a hierarchical FSM that has the upper level FSM with two states $\{INACTIVE, ACTIVE\}$ while the $ACTIVE$ state has internally another FSM with three states $\{Parked, Flying,$

$Exploding\}$. Note that the *orthogonality* (i.e., independence and concurrency) property of statecharts can be easily supported by communicating FSMs in VHDL. Also note that hierarchical FSM provide an intuitive way of supporting the *clustering* property of statecharts.

Event to Signals. The concept of *event* naturally lends itself to the concept of *signal* from VHDL. In the proposed direct translation approach, events from UML statecharts are simply replaced with signals internal to design entities or input/output ports of design entities. Some events may require the introduction of more than one signal. For example, the event of a mine being destroyed (example from the use case presented later) includes the signals that carries that information, plus an additional signal that conveys the value that the score to be increased by, which needs to be passed to the Ship object.

Default States to Initial States. This is a straightforward mapping rule because FSM described using the two-process coding style in VHDL can be easily initialized in any desired initial state that precisely corresponds to the default state of the chart. This is usually done in process #1 (see Fig. **??**) and different default states can be used in different conditions, using just a simple “if" statement.

History to Counters. One way to support the *history* concept (as *refinement* property of statecharts) is by using counters in VHDL. Counters are cheap and easy to describe in VHDL. Their values can be used to control entrance or transitions between FSM states. For example, in our use case, a counter is used to keep track of how many times mine type 2 has been hit by the missile; only after the mine gets hit twice, does explode and the score is increased by 50 points. The counter keeps track of the history of hits. Desired delays and timeouts are easily implemented using timers/stopwatches, which are also easy to describe in VHDL.

VII. Results: Hardware Verification

The proposed approach presented in section VI was used to write the complete VHDL description of the Fly-n-Shoot game by translating all the UML statecharts describing the game from [2] into synthesizable VHDL code. The VHDL code was synthesized, placed and routed successfully using the Intel Quartus Prime Lite Edition Design Software 21.1 [18]. The summary of the final implemented design is shown in Table II. Note that the prototype uses less than 10% of logic elements. The target FPGA family for verification on real hardware was the Intel Altera Cyclone V (device 5CSEMA5F31C6), used by the DE1-SoC development board from Terasic [19]. The board was programmed with the programming file generated by the Quartus tool and the game was tested successfully. The experimental setup, which uses the DE1-SoC board together with a VGA monitor and an NEC Controller (NC) is shown in Fig. 7. The Player can use the NEC Controller to send commands Up, Down, and Shoot; or, alternatively can use the board's pushbuttons $KEY0$, $KEY1$, and $KEY2$ for the same controls. A screen snapshot taken during play action is shown in Fig. 8. The entire system worked correctly as expected with minimal debugging effort. We attribute this success to the very modular and encapsulated coding approach provided by the two-process VHDL coding style. The complete design files, including the VHDL code, pin assignments, as well as a link to a video demonstration will be made publicly available at [20].

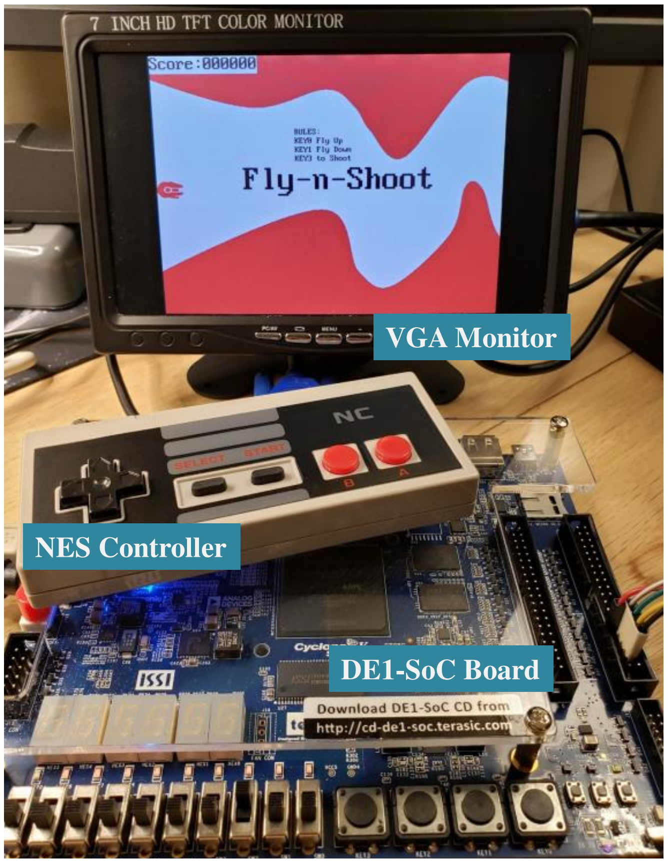

Figure 7. Experimental setup includes DE1-SoC board, VGA monitor, and NES controller. The monitor shows the welcome screen.

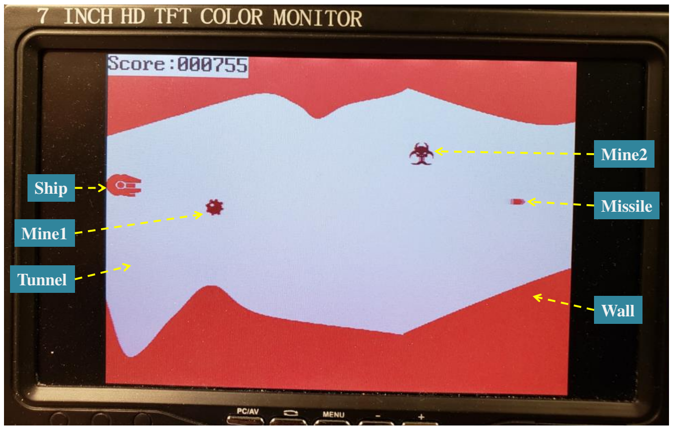

Figure 8. Snapshot of game in action, showing the two types of mines and missile being shot.

Table II
SUMMARY OF RESOURCES USED BY THE FINAL IMPLEMENTATION.

Logic utilization (in ALMs)	Total registers	Total pins	Total PLLs
2,512/32,070 (8 %)	650	42/457 (9 %)	1/6 (17 %)

VIII. CONCLUSION

We presented an FPGA prototype of an event-driven embedded system. The prototype was developed using a practical approach for translating UML statecharts into synthesizeable VHDL code. This approach is based on a set of mapping rules from UML constructs to VHDL code using specifically the two-process VHDL coding style. This coding style creates clean and correct-by-design VHDL code. The approach was demonstrated successfully on a complex event-driven embedded system use case: the Fly-n-Shoot game, which was verified on real hardware, the Intel Altera Cyclone IV FPGA. In future work, we plan to implement the mapping rules presented here into an automated tool, which has advantages on its own, including shorter translation times, elimination of potential error due to designer manual coding, and elimination of the need for VHDL programming skills from designers who would need only to master UML statecharts.

REFERENCES

[1] Miro Samek, "A Crash Course in UML State Machines," *Application Note*, 2015.
[2] ——, "Practical UML Statecharts in C/C++," *Second Edition, Elsevier*, 2009.
[3] J. Cereijo Garcia and R.R. Osorio, "Hardware Implementation of Statecharts for FPGA-based Control in Scientific Facilities," *IEEE Conf. on Design of Circuits and Integrated Systems (DCIS)*, 2019.
[4] Grzegorz Labiak, "From UML statecharts to FPGA - the HiCoS approach," *FDL*, 2003.
[5] S. Wood, D. Akehurst, O. Uzenkov, G. Howells, and K. McDonald-Maier, "A Model-Driven Development Approach to Mapping UML State Diagrams to Synthesizable VHDL," *IEEE Trans. on Computers*, vol. 57, no. 10, pp. 1357-1371, Oct. 2008.
[6] Pong P. Chu, "FPGA Prototyping by VHDL Examples," *Wiley*, 2008.
[7] T.G. Moreira, M.A. Wehrmeister, C.E. Pereira, J.-F. Petin, and E. Levrat, "Generating VHDL Source Code from UML Models of Embedded Systems," *IFIP Advances in Information and Communication Technology, Springer*, 2010.
[8] E. Ebeid, D. Quaglia, and F. Fummi, "Generation of VHDL code from UML/MARTE sequence diagrams for verification and synthesis," *Euromicro Conference on Digital System Design*, 2012.
[9] G. Bazydlo, M. Adamski, M. Wegrzyn, and A. Rosado Munoz, "From UML State Machine Diagram into FPGA Implementation," *IFAC Conference on Programmable Devices and Embedded Systems*, 2013.
[10] M. Dossis, "Custom Hardware Synthesis from UML," *Int. Journal of Engineering Research And Management (IJERM)*, 2014.
[11] J. Cereijo Garcia and R.R. Osorio, "Comparison of Hardwired and Microprogrammed Statechart Implementations," *MDPI electronics*, 2020.
[12] G. Booch, J. Rumbaugh, and I. Jacobson, "The Unified Modeling Language User Guide (1st edition)," *Addison-Wesley Professional*, 1998.
[13] Unified Modeling Language, "Balancing the Equation: Where are Women and Girls in Science, Engineering, and Technology," [Online]. Available: *https://www.omg.org/spec/UML/2.5.1/About-UML/*, 2022.
[14] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, no. pp. 231-274, June 1987.
[15] D. Bjorklund, J. Lilius, and I. Porres, "A Unified Approach to Code Generation from Behavioral Diagrams," *Chapter 2 in Selected Contributions on UML, SystemC, System Verilog, Mixed-Signal Systems, and Property Specification from FDL'03*, Springer, 2004.
[16] I.A. Niaz and J. Tanaka, "Java Code From UML Statecharts," *Int. Journal of Computer and Information Science*, vol. 6, no. 2, June 2005.
[17] N.A. Jagannathan, "A UML driven ASIC design methodology aided by an automated UML-SystemC translator," *M.S. Thesis, National University of Singapore*, 2005.
[18] Intel Quartus Prime Lite Edition Design Software, Version 21.1, Intel Altera, [Online]. Available: *https://www.intel.com*, 2022.
[19] DE1-SoC FPGA Development Board, Terasic, [Online]. Available: *https://www.terasic.com.tw*, 2022.
[20] Fly-n-Shoot Game Implementation in VHDL, GitHub Repository, https://github.com/eigenpi/fly-n-shoot, 2023.