PUTTING HARDWARE-SOFTWARE CODESIGN INTO PRACTICE

G. Schrott * **T. Tempelmeier** **

* *Technische Universität München, Institut für Informatik, D-80290 Munich, Germany*
** *Fachhochschule Rosenheim, Fachbereich Informatik, D-83024 Rosenheim, Germany*

Abstract: This contribution reports on an approach to hardware/software codesign based on on a state transition system design language MAD. MAD has been used for software designs for some time, but is now being extended to hardware designs by adding VHDL as one of its target languages. The experience of adding a new dimension (i.e. hardware) to the design space and of dealing with some state-of-the-art hardware development methods and tools is presented. The approach is illustrated with an implementation of a control system for an elevator.

Keywords: Hardware-software codesign, Real-time languages, Real-time systems, Embedded systems, Distributed control, Hierarchical control, Process control, Control applications, Software engineering, Hardware description languages, Programmable logic devices, Micro-systems.

1. INTRODUCTION

Hardware description languages (HDLs) are increasingly used to describe behavior, structure, data-flow, or logic gates structure of hardware. Hardware in this context may mean programmable logic devices (PLDs) or application specific ICs (ASICs) in various forms or whole boards. From the abstract hardware descriptions lower level descriptions (down to the masks for programming the PLDs or for manufacturing ASICs) are generated automatically by synthesis systems. This has been established in practice for some time down from the register transfer level. More recently, high level synthesis systems, starting from a purely behavioral description to automatically synthesize register transfer level descriptions, have started to make their way into practice.

Today's hardware description languages are remarkably similar to (software) programming languages. This is especially true, if one considers real-time programming languages, because (only) these contain concepts such as parallelism and time. For the leading hardware description language VHDL, the similarity to Ada has been an explicit design principle (Menchini, 1993). A comparison of VHDL and Ada, showing the similarities and differences between these languages can be found in (Tempelmeier, 1994*a*).

The similarity of software and hardware description languages and the progress in high level synthesis have made it possible, to put design ideas to hardware or software, alike—at least theoretically. This gives rise to the new discipline of hardware-software codesign, the integrated design of hardware and software. Hardware-software codesign may start from a common system description in a suitable language. Numerous languages have been used for system descriptions, e.g. Ada, C, C^x, ESTEREL, LUSTRE, OOFS, Petrinets, PRAM, Promela, SDL, State-charts, VHDL ((Halbwachs, 1993), (Tempelmeier, 1994*b*)). From these system descriptions hardware and software descriptions (i.e. programs) can be generated in Ada, C, C++,

HardwareC, Verilog, VHDL (see figures 1a–b for two examples).

Using the term hardware-software codesign in a broader sense, it could also include hardware-software re-partitioning without assuming a common system description (figure 1c). Various criteria for the decision whether to use hardware or software are in use, e.g. performance (including hardware/software communication overhead), cost, form factor (space, weight, power consumption), safety, architectural cleanness and simplicity.

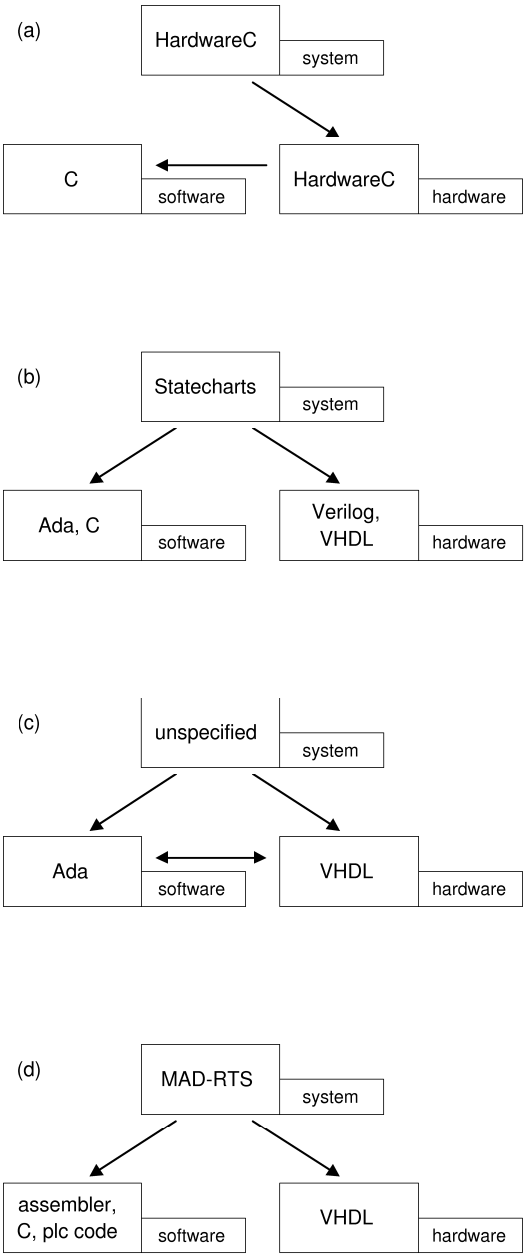

Fig. 1.
Some hardware-software codesign schemes
a) academic approach
b) commercial approach
c) hardware/software re-partitioning
d) the technique in this contribution
a) - c) taken from (Tempelmeier, 1994*b*)

This contribution reports on an actual case study, where hardware or software implementations have been derived from an already implemented system description for distributed real-time systems. It is based on the concept of multi-agent systems and allows for a uniform programming of complex process control systems on a microprocessor field-bus network. The specification of the agents is done in an hardware independent language using the notion of states and guarded commands. Till now the system is used for PCs, PLCs and micro-controllers. An already implemented code generator for a finite state machine is now extended to generate VHDL-code.

The experience from this case study includes:

- A naive and simplistic approach like "Write down any legal construct in a hardware description language and let high level synthesis tools synthesize some hardware" does not work, of course. Thus, what may seem easy or clear theoretically, becomes very interesting in practice.
- The standard criteria for assigning functionality to hardware or software are helpful in some cases, e.g. when performance or cost are critical factors. In our case study, some of the standard criteria did not apply, and deciding on hardware/software partitioning was based partly on intuitive arguments.
- It works!

2. THE APPROACH

2.1 *System Specification with MAD*

The proposed multi-agent distributed real-time system (MAD–RTS) is used to program complex distributed heterogeneous real-time applications (Schrott, 1995). It also highly supports structured design and test- and maintainability of the resulting system. Dedicated computers guarantee the needed response times in time-critical applications; the complexity of control applications is mastered. In MAD–RTS specification and coding of the control programs are closely related. The MAD-language supports the definition of small agents which communicate with each other by sending contracts to start activities of other agents. Agents are small autonomous units which are able to perceive, to plan, to communicate with each other, to decide and to act. If more than one agent can perform the needed activity, bids are sent and the 'cheapest' agent will get the contract. Each agent has a set of defined states

and executes the guarded actions of this state. It interacts via generic subagents with physical input/output channels, timers and other specific hardware. As in object oriented programming the agent executes the contract like a method without showing the real implementation.

A complete MAD program starts with the declaration of the target microprocessors and is followed by a number of agents. A syntactic definition of an agent written in MAD can be found in (Schrott, 1995), its semantic structure is briefly shown below. An example of an agent showing its functionality is given in chapter 2.3.

- Declaration of the target microprocessor
  The target processor chosen within the network to execute the agent is defined after the keyword `host`. More than one agent may of course run on one processor.
- Instantiation of subagents
  At lowest level generic subagents are defined to realize the interface to the technical process (e.g. digital and analog input/output, timer, stepper motor, pid-controller). They are instantiated in the `decls` definition of a MAD-program with the actual i/o-address and mnemonic identifiers to enhance self documentation of the program.
- Declaration of contracts
  The only interface between agents is the contract protocol. In the `contracts` definition every contract which can be called by other agents is listed. A contract transfers parameters and causes the execution of actions or in most cases a state transition in the agents action part. Included in the communication system is a contract net bidding protocol.
- Action part
  The action part is subdivided into a set of states. Control tasks usually change between different states of operation, e.g. at lowest level 'on' or 'off', at higher level 'open', 'closed' or 'error'. One state is active and the actions comprising it are executed. Each action is bound by a condition (guard) and only if it is true the corresponding statements are executed. At lower level these conditions will be signals from the controlled process, at higher level it may be timers or conditional expressions on variables. All conditions of the active state of one agent are tested cyclically and all agents fixed to one microprocessor are executed one after the other *to guarantee an exactly predictable time behavior*. Two distinguished states are obligatory on each agent: At start up the agent goes into the `initial` state, in case of emergency the `shutdown` state is entered.

In related work some similarities to MAD–RTS may be found, e.g. in all approaches which are based on finite state machines like SDL (Færgemand and Olsen, 1994) or Codesign Finite State Machines (CSFM's) (Chiodo *et al.*, 1994). Also, the concepts of synchronous real-time languages show some similarity to MAD–RTS, and they also allow for an automatic transition to silicon (Halbwachs, 1993).

2.2 *Object-Based Hardware-Software Assignment*

A thorough survey of the existing literature on hardware/software codesign showed that many of the proposed schemes could be used in principle. However, a decision was taken only to use target languages of primary importance in embedded systems (in terms of practical use). This restricted our choice to C and plc programs, and to VHDL, and may possibly exclude some hardware/software codesign approaches for our purposes. Further, it is not in the scope of this study to evaluate or even survey hardware/software codesign methods. The motto was rather, "Just do it! Do it as good as it is possible with today's commercially available tools and evaluate the necessary design effort!" This again ruled out some hardware/software codesign approaches which focus on hardware/software codesign research per se.

The above, very pragmatic, standpoint lead us to our hardware/software codesign approach, which might be described as *intuition-guided, object-based, coarse-grain hardware/software repartitioning and assignment,* based on the similarities of hardware description languages to (software) programming languages.

MAD objects are used as smallest design units for hardware or software implementation. This means that only whole agents including their subagents are candidates in the hardware/software assignment process. Fine-grained hardware/software assignment was not considered due to the negative experience with communication overhead, which one of the authors had encountered during a former work assignment in aerospace industry. Object-based hardware-software assignment partly avoids this problem and also preserves the existing overall system structure (Tempelmeier, 1994*b*).

As an example for the following, the door agent of an elevator control system from (Schrott, 1995) is taken as a design unit to be put into hardware. Some reasoning about the decision "Implement it in hardware or software?" will be presented later on.

2.3 *Translating MAD to VHDL*

The following MAD-program shows the hardware independent specification of the door agent of an elevator control system:

```
agent door2 host mealy;

decls
  DigOut motor(out1,on,off);
  DigOut direction(out2,open,close);
  DigIn open_key(in1,on,off);
  DigIn closed_key(in2,on,off);
  DigIn light_barrier(in3,interrupted,ok);
  Timer delay;

contracts
  open do newstate opening;

states
  closed/shutdown:
    true  => motor.off;

  opening:
    true  => {direction.open; motor.on;}
    open_key.on  => {motor.off;
                     delay(10000);
                     newstate waiting;}

  waiting:
    delay.tout   => newstate closing;

  closing/initial:
    true         => {direction.close;
                     motor.on;}
    closed_key.on
                 => {newstate closed;
                     elevator1.start;}
    light_barrier.interrupted
                 => {motor.off;
                     newstate opening;}
endagent;
```

Other agents may send the contract `door2.open` in order to induce the door agent to open the elevator door, wait for 10 seconds and close it again. If the light barrier is interrupted during closing the door is opened again. If the door is closed, the contract `elevator.start` is sent to agent elevator1.

The programming system MAD–RTS is hosted on a PC under MS-DOS. It contains the compiler for MAD and code generators and run time systems for different targets. The compiler uses the contract specification to generate automatically the code for the transmission of contracts between agents on the same or on different targets. Code generators are available for Intel 80x86, MC68HC11 and programmable logic controllers (figure 1d); the communication link is implemented for RS 232 serial link and the CAN–field–bus. If only digital input/output, no numeric expressions and no parameter passing to contracts are used, a further code generator is available which translates an agent program into a table of a *Mealy* state machine $M = (\mathcal{Q}, \Sigma, \Delta, \delta, \lambda, q_0)$ with

$\mathcal{Q}$	set of states
Σ	input alphabet
Δ	output alphabet
$\delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$	state transition
$\lambda : \mathcal{Q} \times \Sigma \to \Delta$	output function
q_0	initial state

It is relatively easy, to transcribe the finite state machine implementation into VHDL. The code is included in the appendix. The translation to VHDL has been done manually for the prototype. An automatic transition from MAD to VHDL is easily possible and may be envisaged for the future (figure 1d).

Some remarks on the VHDL code are given here from the point of view of software engineers (cf. listing in the appendix).

- The ports of the entity form the interface to this hardware unit. They closely resemble the declarations in the MAD program. Contracts and subagents for digital input/output and for the timer are implemented as interface bits of the hardware entity.
- In the architecture part, the states of the finite state machine are defined as an enumeration type.
- In software one would perhaps implement the finite state machine as nested case-statements. An outer case-statement would cover all possible states, and the nested case- or if-statements would cover all possible inputs in a particular state. The VHDL representation follows this scheme in the process `NextStateDecode`.
- Two more processes (`RegisteredState` and `DecodeOutputs`) are required by the particular coding style suggested by the synthesis tool.

Generally, not all VHDL constructs are synthesizable. So each synthesis tool places certain restrictions on the designer. By following the style guide for finite state machines of the synthesis tool manufacturer (Skahill, 1996), no unsolvable synthesis problems were encountered in our case study.

We used Model Technology's V–System for VHDL simulation and Cypress Semiconductor's Warp 2, rel. 3.5 and 4.1 for synthesis and post-synthesis simulation. Wilson WindowWare's WinEdit served

as editor providing syntax colouring, error capture, etc. and also as integration tool, allowing to call the other tools. All tools were running under Windows95/NT.

As hardware targets, Cypress Semiconductor's Flash370 CPLDs with 128 macro-cells were used. These could easily be programmed from a PC. The use of CPLDs further reduced the low end development effort, because no "place and route" as for FPGAs was necessary.

Experience with V–System was very good.

Experience with Warp 2, rel. 3.5 was rather bad. The problem was not that only a subset of VHDL was synthesizable (of course), but that many undocumented and undetected restrictions did exist. Admittedly, the tool is available at almost no cost. But, it is *the* design tool for programming Cypress ICs. How could anybody have done a large design with this tool? Probably, designers have used this tool with a severely restricted subset of VHDL, e.g. only with basic boolean equations.

Experience with Warp 2, rel. 4.1 (Skahill, 1996) was much better. However, some annoying restrictions as compared to full VHDL remain. As an example, variables cannot be used to hold or pass information, only as value holders. This means that no memory elements are synthesized for variables and that signals must be used, instead. But, in contrast to variables, signals cannot be declared inside processes, and thus no ideal encapsulation and structuring of the design is possible. As a second example, procedures did not work properly in release 3.5 and should not work properly in release 4.1 according to the documentation, when nested inside other statements. However, they did work properly, at least in the few cases we dared to try out.

As an intermediate conclusion, generating application specific hardware has come within easy reach of software engineers by virtue of modern hardware description languages. We had no problems of fitting the hardware agents into our chips and did not evaluate the efficiency of silicon area usage of our VHDL compiler. We consider restrictions of available silicon area of minor importance in the future, and did not take into consideration to use hand-crafted hardware or software implementations.

3. THE RESULTING ARCHITECTURE

To test the MAD–RTS system a twin elevator with four floors built with FischerTechnik (see figure 2) was used. In the previous MAD software approach (Schrott, 1995) it was controlled by one micro-controller MC68HC11 on each floor, one in each cage and one at each motor platform, all connected via the CAN-field-bus. The agents were distributed over these micro-controllers.

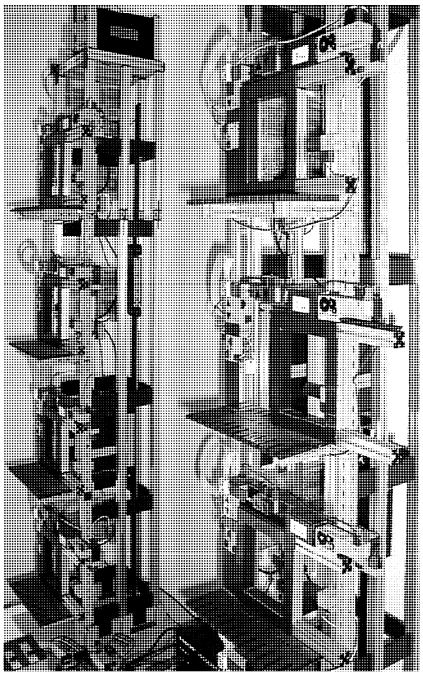

Fig. 2. FischerTechnik model of twin elevator

After the transfer of the door agents into hardware the micro-controllers on each floor are substituted by a CPLD Cypress 374 for each door (see figure 3). To connect them to the CAN-field-bus an additional CAN-SLIO node is used on each floor which translates the contracts sent via CAN into digital signals connected to the CPLD. The contract to open the door and to start the elevator received resp. sent by the door agent are thus available to agents running on the two remaining micro-controllers. Supposing that the CAN protocol will be soon available in VHDL the interface to the field bus can be integrated into the CPLD.

4. HARDWARE OR SOFTWARE?

In the described approach, it was a bit tedious to get to a really synthesizable VHDL description, but this was not the big challenge. It was also intuitively clear which parts of the system could or should be implemented in hardware or software. But is was hard to find objective criteria for hardware/software assignment, which would—ideally in an algorithmic way—produce the answer "hardware" or "software" for each part of the system. In the following, the standard criteria of hardware-software codesign publications will be investigated for their suitability to our case study.

- Performance
  If there is a performance problem, it may

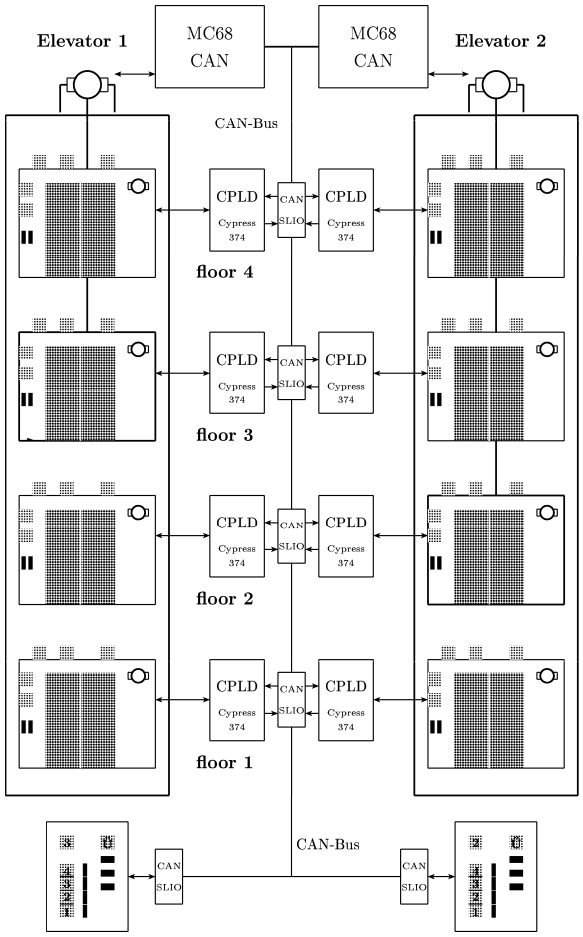

Fig. 3. Microprocessor network for twin elevator

be a good idea to place time critical parts in hardware. This may also be done, when there is an indirect performance problem, e.g. when the requirement to stay below some prescribed processor utilization (Lions, 1996) [1], cannot be fulfilled without sacrificing safety . Sometimes even slow mechanical processes like elevators may encounter performance problems, e.g. a too slow reaction to interrupting the light barrier during closing of the door. This could result from a centralized synchronous control system (typically a PLC), where polling all sensors brings cycle time to the upper acceptable limit.

- Cost
  Cost is a good criterion in mass production or, concerning development cost, in situations where only the software or the hardware paradigm is mastered by the designers.
- Form factor
  The term form factor stands for space, weight, power restrictions, and similar properties of the system under consideration. The form factor is of importance in aerospace applications or for portable or battery-powered devices, for instance.
- Safety
  Hardware solutions are often deemed safer than software solutions. Given the modern design methods for digital hardware, with hardware description languages much alike software languages, this obviously does not hold with respect to design errors. However, there may be additional safety against transient errors, because a direct logic implementation will perhaps more easily recover from a fault than a microcontroller with a destroyed stack, for instance. A further gain in safety could be expected from improvements in architectural cleanness and simplicity.
- Architectural cleanness and simplicity.
  Object-based hardware/software re-partitioning can improve the cleanness and simplicity of the overall system architecture. For instance, in (Tempelmeier, 1994*b*) it is suggested to move a time management unit from software to hardware, because in this architecture software time management needs four different interrupts to derive one valid time value [2]. Getting rid of these interrupts makes it easier to verify the software. Similarly, as soon as several programs are intermingled on one processor, which implies some form of time multiplexing, verification becomes difficult again. Moreover, any changes in any program require a reverification of the time behavior of all intermingled programs in the processor. Essentially, the fine-grained true parallelism, which is available in hardware solutions, constitutes the most important advantage of hardware over software in the architectural domain. One can reach true parallelism in software by using multiple processors, but this results only in coarse-grained parallelism (with the standard von Neumann architectures).

In our case, there were no serious performance problems due to the decentralized control system, and cost and form factor did not matter. It were thus safety considerations and the desire for architectural cleanness that governed the hardware/software assignment process. Due to the coarse-grained assignment strategy, an intuition-guided process seems appropriate. Additional computer-generated metrics on performance, cost, and form factor would be helpful in the general case.

[1] A maximum processor utilization of 80% was allowed in this case.

[2] The four interrupts were necessary due to the complex scheme of operating modes in the system.

5. CONCLUSION

Our case study has shown that about the same effort is necessary to implement a hardware solution or a software solution for a logical unit in our lift control system. MAD-RTS allows for a hardware independent programming; the compiler can generate code for agents running on conventional processors, but also via finite state machine and VHDL create a *hardware agent*. The communication between the agents is automatically included.

More difficult is the question of whether to assign a unit to hardware or to software. Many standard guidelines for this problem do not apply to our example. Safety considerations and architectural cleanliness and simplicity give clues that hardware could be used, but these are rather fuzzy criteria. However, from our experience with this case study, we would not accept the question "Why not leave everything in software?" without replying "Why not implement everything in hardware?".

6. REFERENCES

Chiodo, M., P. Giustro, A. Jurecska, H.C. Hsieh, A. Sangiovanni-Vincentelli and L. Lavagno (1994). Hardware-software codesign of embedded systems. *IEEE Micro* **14(4)**, 26–36.

Færgemand, O. and A. Olsen (1994). Introduction to SDL-92. *Computer Networks and ISDN Systems* **26**, 1143–1167.

Halbwachs, N. (1993). *Synchronous Programming of Reactive Systems*. Kluwer Academic Publisher. Dordrecht.

Lions, J.L. (1996). Ariane 5 flight 501 failure. Technical report. Report by the Inquiry Board. http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html.

Menchini, P.J. (1993). An introduction to VHDL. In: *NATO ASI Series, Series E: Applied Sciences* (Jean P. Mermet, Ed.). Vol. 249. Kluwer Academic Publisher. Dordrecht. pp. 359–384.

Schrott, G. (1995). A multi-agent distributed real-time system for a microprocessor field-bus network. In: *Proc. of 7th Euromicro Workshop on Real-Time System, Juni 14–16, 1995, Odense, Denmark*. IEEE Computer Society Press. Los Alamitos, California. pp. 302–307.

Skahill, K. (1996). *VHDL for Programmable Logic*. Addison-Wesley. Reading, Mass.

Tempelmeier, T. (1994*a*). VHDL and Ada. In: *Proc. of the Fourth Symposium 'Ada in Aerospace', Nov 8–11, 1993, SAS Royal Hotel, Brussels*. Eurospace. Paris, France. pp. 103–111.

Tempelmeier, T. (1994*b*). A note on hardware-software codesign. In: *Proc. of the 19th IFAC/IFIP Workshop on Real Time Programming, June 22–24, 1994, Isle of Reichenau, Germany* (W.A. Halang, Ed.). Pergamon Press, Elsevier Science. Oxford, UK. pp. 121–126.

APPENDIX

VHDL Listing of the Door Agent

```
---------------------------------------
-- Door agent in VHDL
-- Version, 30-May-1997
-- Functionality: ok
-- Synthesis: ok
-- Cypress FSM Encoding Style: 3a
--    Three processes;
--    outputs decoded from state bits
---------------------------------------

ENTITY door2 IS

  PORT (

  -- contract interface
  open_door   : IN  bit;
  elev1_start : OUT bit;

  -- subagents
  motor,                  -- DigOut
  direction   : OUT bit; -- DigOut

  open_key,               -- DigIn
  closed_key,             -- DigIn
  light_barr  : IN  bit; -- DigIn

  delay_done  : IN  bit;
  delay_start : OUT bit;
                -- delay implemented
                -- as DigIn/Out from
                -- external timer

  -- "technology bits"
  clk, reset  : IN  bit

  );

END ENTITY door2;

---------------------------------------

ARCHITECTURE mealy OF door2 IS

  TYPE states IS ( closed, opening,
                   waiting, closing);
  SIGNAL current_state,
         next_state      : states;
  SIGNAL timer_st :
              integer range 0 to 2 :=0;
```

```
BEGIN

NextStateDecode:
  PROCESS (current_state, open_door,
           delay_done, open_key,
           closed_key, light_barr)
  BEGIN
    IF (open_door = '1') THEN
        next_state <= opening;
    ELSE
      CASE current_state IS
        WHEN closed =>
          next_state <= closed;
        WHEN opening =>
          IF (open_key = '1') THEN
            next_state <= waiting;
          ELSE
            next_state <= opening;
          END IF;
        WHEN waiting =>
          IF (delay_done = '1') THEN
            next_state <= closing;
          ELSE
            next_state <= waiting;
          END IF;
        WHEN closing =>
          IF (light_barr = '0') THEN
            next_state <= opening;
          ELSE
            IF (closed_key = '1') THEN
              next_state <= closed;
            ELSE
              next_state <= closing;
            END IF;
          END IF;
        WHEN OTHERS =>
          next_state <= closing;
      END CASE;
    END IF;

  END PROCESS NextStateDecode;

RegisteredState:
  PROCESS (clk, reset)
  BEGIN
    IF (reset = '1') THEN
      current_state <= closing;
    ELSIF (clk'event AND clk = '1') THEN
      current_state <= next_state;
    END IF;
  END PROCESS RegisteredState;


DecodeOutputs:
  PROCESS (current_state)
  BEGIN
    IF (current_state = waiting) THEN
      delay_start <= '1';
    ELSE
      delay_start <= '0';
    END IF;
    IF current_state = opening OR
            (current_state = closing
             AND light_barr = '1')  THEN
      motor <= '1';
    ELSE
      motor <= '0';
    END IF;
    IF (current_state = opening) THEN
      direction <= '1'; -- direction open
    ELSE
      direction <= '0';
    END IF;
    IF (current_state = closing AND
                  closed_key = '1') THEN
      elev1_start <= '1';
    ELSE
      elev1_start <= '0';
    END IF;
  END PROCESS DecodeOutputs;

END ARCHITECTURE mealy;

---------------------------------------
```