POLIS

Ömer Doğan
5040622203
Introduction

• Embedded controllers for reactive real-time applications are implemented as mixed software-hardware systems.
• Generally, software is used for features and flexibility, while hardware is used for performance.
• Design of embedded systems can be subject to many different types of constraints, including timing, size, weight, power consumption, reliability, and cost.
Introduction

- Current methods for designing embedded systems require to specify and design hardware and software separately.
- Specification, often incomplete and written in non-formal languages, is developed and sent to the hardware and software engineers.
- Hardware-software partition is decided *a priori* and is adhered to as much as is possible, because any changes in this partition may necessitate extensive redesign.
- Designers often strive to make everything fit in software, and off-load only some parts of the design to hardware to meet timing constraints.
Introduction

• Problems with the mentioned methods are:
  – Lack of a unified hardware-software representation, which leads to difficulties in verifying the entire system, and hence to incompatibilities across the HW/SW boundary.
  – \textit{A priori} definition of partitions, which leads to sub-optimal designs.
  – Lack of a well-defined design flow, which makes specification revision difficult, and directly impacts time-to-market.
Introduction

• There are many different academic approaches to try to solve the problem of embedded system design.
• None of the current methods address satisfactorily the issues of unbiased specification and efficient automated synthesis for control-intensive reactive real-time systems.
• POLIS designers are developing the methodology for specification, automatic synthesis, and validation of this subclass of embedded systems.
• Design is done in a unified framework, POLIS, with a unified hardware-software representation, so as to prejudice neither hardware nor software implementation.
• This model is maintained throughout the design process, in order to preserve the formal properties of the design.
POLIS Structure

- The **POLIS** system is centered around a single Finite State Machine-like representation, CFSM (Co-design Finite State Machine)
- A Co-design Finite State Machine (CFSM), like a classical Finite State Machine, transforms a set of inputs into a set of outputs with only a finite amount of internal state.
- The difference between the two models is that the synchronous communication model of classical concurrent FSMs is replaced in the CFSM model by a finite, non-zero, unbounded reaction time.
- This model of computation can also be described as Globally Asynchronous, Locally Synchronous
- Each element of a network of CFSMs describes a component of the system to be modeled. The CFSM specification is a priori unbiased towards a hardware or software implementation.
POLIS Structure

• While both perform the same computation for each CFSM transition, hardware and software exhibit different delay characteristics.
  – A synchronous hardware implementation of CFSM can execute a transition in 1 clock cycle, while a software implementation will require more than 1 clock cycle.
• The design flow that is currently implemented in the POLIS system is depicted in the following figure.
POLIS Structure
High Level Language Translation

- In **POLIS**, designers write their specifications in a high level language (e.g., ESTEREL, graphical FSMs, subsets of Verilog or VHDL) that can be directly translated into CFSMs.
- Any high level language with precise semantics based on extended FSMs can be used to model individual CFSMs (currently ESTEREL is supported directly).
Formal Verification

- The formal specification and synthesis methodology embedded within POLIS makes it possible to interface directly with existing formal verification algorithms that are based on FSMs.
- POLIS includes a translator from the CFSM to the FSM formalism which can be fed directly to verification systems.
- In addition to uncovering bugs in a design, we also use formal verification to guide the synthesis process. Since the abstract CFSM model covers the behavior of all possible hardware-software implementations at once, it is possible to refine the specification base on the output of formal verification.
- Formal verification tools today still have problems with complexity. We have developed a methodology which incorporates a set of abstraction and assumption rules specific to POLIS and CFSMs. With this formal verification methodology we are able to verify designs which are larger than is previously possible.
System Co-simulation

- System level HW-SW Co-simulation is a way to give designers feedback on their design choices.
- These design choices include HW-SW partitioning, CPU selection, and scheduler selection.
- We currently utilize PTOLEMY as a simulation engine, but we are not limited to PTOLEMY. VHDL code including all the co-simulation information is also an output of the system, so any commercial VHDL simulator can be adapted for this purpose.
Design Partitioning

• By design partitioning we mean making system-level design decisions such as HW-SW partitioning, target architecture selection, and scheduler selection.
• We therefore provide the designer with an environment to quickly evaluate any such decision through various feedback mechanisms from either formal verification or system co-simulation.
A CFSM sub-network chosen for HW implementation is implemented and optimized using logic synthesis techniques from SIS.

Each CFSM, interpreted as a Register-Transfer Level specification, can be mapped into BLIF, XNF, VHDL or Verilog.
Software Synthesis

• A CFSM sub-network chosen for SW implementation is mapped into a software structure that includes a procedure for each CFSM, together with a simple Real-time Operating System:
  – CFSMs. The reactive behavior is synthesized in a two-step process:
    • Implement and optimize the desired behavior in a high-level, processor-independent representation of the decision process similar to a control/data flow graph.
    • Translate the control/data flow graph into portable C code and use any available compiler to implement and optimize it in a specific, micro-controller-dependent instruction set.
Software Synthesis

- A timing estimator quickly analyzes the program and reports code size and speed characteristics. The algorithm uses a formula, with parameters obtained from benchmark programs, to compute the delay of each node in the control/data flow graph for various micro-controller architectures (characterization data for MIPS R3000 and Motorola 68HC11 and 68332 are already available). The precision of the estimator, with respect to true cycle counting, is currently on the order of plus or minus 20 percent.

- The estimator allows one to obtain accurate estimates of program execution times for any characterized target processor, by first appending to each statement in the C code generated from the control/data flow graph instructions that accumulate clock cycles, then compiling and execute the software on the host workstation. The same method is used to synchronize hardware and software blocks within the PTOLEMY-based co-simulation environment.
Software Synthesis

- Real-time Operating System. An application-specific OS, consisting of a scheduler (e.g. Rate-Monotonic and Deadline-Monotonic) and I/O drivers, is generated for each partitioned design.
Interfacing Implementation Domains

- Interfaces between different implementation domains (hardware-software) are automatically synthesized within POLIS. These interfaces come in the form of cooperating circuits and software procedures (I/O drivers) embedded in the synthesized implementation. Communication can be through I/O ports available on the micro-controller, or general memory mapped I/O.
CFSM Structure

OFF

NOT key_on AND
(key_off OR belt_on) / alarm(0)

/ alarm(0)

NOT key_on AND
(end_10 OR belt_on OR key_off) / alarm(0)

WAIT

key_on / start_timer

key_on / start_timer

ALARM

key_on / start_timer

NOT key_on AND
NOT key_off AND
NOT belt_on AND
end_5 / alarm(1)
module belt_control:
    input reset, key_on, key_off, belt_on, end_5, end_10;
    output alarm(boolean), start_timer;
    loop
        do
            emit alarm(false);
            every key_on do
                do
                    emit start_timer;
                    await end_5;
                    emit alarm(true);
                    await end_10;
                    watching [key_off or belt_on];
                    emit alarm(false);
                end
            watching reset
        end
    end.
CFSM Structure

Complete design needs a timer model that can be represented:

```plaintext
module timer:
constant count_5:integer;
input msec, start_timer;
output end_5, end_10;
every start_timer do
    await count_5 msec;
    emit end_5;
    await count_5 msec;
    emit end_10;
end.
```
PTOLEMY Co-Simulation Environment

- Cosimulation is used in POLIS both for functional debugging and for performance analysis during the architecture selection process.
PTOLEMY Co-Simulation Environment

```c
/* defines event emission and detection macros */
#include "os.c"
...
extern unsigned int8bit v_alarm;
...
void _t_s_belt_control_0(int proc, int inst)
{
    static unsigned int8bit v_state_tmp; /* buffer of v_state */
    v_state_tmp = v_state; /* update the state variable buffer */
    startup(proc);
    while (1); /* * update the input event buffers */
    L1: if (detect_e_reset_to_s_belt_control_0) { /* did reset occur */
        DELAY(40); goto L4;
    } else {
        DELAY(20); goto L2;
    }
    L3: switch (v_state_tmp) { /* branch based on current state */
        case 1: DELAY(41); goto L6;
        case 0: DELAY(63); goto L4;
        case 3: DELAY(86); goto L14;
        default: DELAY(107); goto L7;
    }
    ...
    L10: DELAY(12); v_state = 2; goto L0; /* go to next state */
    L11: DELAY(10); v_alarm = 1; goto L12; /* alarm is true */
    L12: DELAY(14); emit_e_alarm(); goto L13; /* emit event */
    ...
    end:
    always_cleanup(proc); /* clear input event buffers */
    return; /* return to RTOS/simulator */
}
PTOLEMY Co-Simulation Environment

The control and data flow graph of the seat belt controller
PTOLEMY Co-Simulation Environment

The PTOLEMY Interface code for the seat belt controller

```c
name { belt_control }
desc { Star generated by POLIS 68hc11:
    min_time: 142
    max_time: 363
    tot_time: 478
}
input {
    name { e_reset }
    type { int }
}
...
output {
    name { e_alarm }
    type { int }
}
public {
    ...
    /* increment clock count only if SW */
    #define DELAY(d) {if (belt_control_Star->impl1) _delay += (d);} ...
    /* include C code for behavior */
    #include "z_belt_control_0.c"
    begin {
    /* setup code */
    ...
    }
    go {
    /* transition execution code */
    ...
    _delay = 1.0;
    _t_z_belt_control_0(0,0);
    end_time = floor( now ) + _delay;
    ...
    }
}```
CFSM Implementation in POLIS

- System specification is given in an input language
- System specification is translated to SHIFT format
- Implementation in POLIS
CFSM Implementation in POLIS

Following ESTEREL specification can be considered for the explanation.

It implies that state machine waits for A then that of B and then emits output 0. State transition graph can be obtained like this.

```
module wait_A_then_B:
  input A, B;
  output O;
  loop
    await A;
    await B;
    emit O
  end loop
end module
```
CFSM Implementation in POLIS

• SHIFT code implementing the this behaviour can be given as

```
Trivial lines explaines that inputs are consumed, no outputs are emited no state change is made.

.r 0
#transition: A B st => 0 st
.trans
0 - 0 0 0 0 # trivial
1 - 0 0 1
- 0 1 0 1 # trivial
- 1 1 1 0
```
CFSM Implementation in POLIS

• To illustrate initial transition, following ESTEREL program should be considered.

```elixir
module emit_then_wait_A_then_B:
  input A, B;
  output O;

  emit 0;
  loop
    await A;
    await B;
    emit 0
  end loop
end module
```

```
Module emit_then_wait_A_then_B:
  Input A, B:
  Output O:

  Emit 0
  Loop
    Await A
    Await B
    Emit 0
  End Loop
End Module
```
CFSM Implementation in POLIS

• As the state graph is modified, SIFT transition relation is also modified:

• New line below placed is added.

```
.r I

#transition: A B st => 0 st
- - I  1 0
```

Machine now begins in stateI and executes the initial transition when it is first invoked. The initial transition emits 0 and moves to state 0.
CFSM Implementation in POLIS

• Now, consider the following ESTEREL module, where instead of waiting for A and B in succession, we wait for them in either order and emit 0 when each has occurred at least once.
CFSM Implementation in POLIS

• The completely specified state machine SHIFT file becomes:

```
.r 0
#transition: A B st => C st
.trans
 1 0 0 0 1
 0 1 0 0 2
 1 1 0 1 0
-1 1 1 0
-0 1 0 1 # trivial
 1 -2 1 0
 0 -2 0 2 # trivial
```
CFSM Implementation in POLIS

• For an incompletely specified machine, the unspecified transitions are converted to empty executions in the default implementation, that is, the inputs are not consumed. As a result, the above machine could be specified more compactly in SHIFT as follows.
CFSM Implementation in POLIS

This machine waits for A and B, never consuming any inputs until both have occurred at some time in the past. It then emit 0, consumes inputs, and begins waiting again. Note that this new machine has only one state in the transition relation and effectively uses the input buffers two store the other two states.

```
.r 0
#transition: A B st => 0 st
.trans
1 1 0 1 0
```
CFSM Implementation in POLIS

• While this form of specification can be more compact and adheres to the semantics of the interpretation of incompletely specified transition relations in the CFSM model, it can also have unplanned side effects, as the following example illustrates
CFSM Implementation in POLIS

XOR operator denotes the presence of only one of A and B. This code emits 0 or P if A or B occurs alone. This could be used to signal an error for example. A possible SHIFT representation might be:

```
module wait_A_xor_B;
input A, B;
output O, P;

loop
    await (A xor B);
    present A and not B then emit O end;
    present B and not A then emit P end
end loop
end module

#transition: A B st => O P st
.trans
1 0 0 1 0 0
0 1 0 0 1 0
```
CFSM Implementation in POLIS

- In this case, an incorrect implementation would result. If the input $A,B = 1,1$ occurs, the inputs are never consumed and the machine deadlocks. A correct SHIFT specification would include a trivial transition for the $1,1$ input.

```plaintext
#transition: A B st => O P st
.trans
1 0 0 1 0 0
0 1 0 0 1 0
1 1 0 0 0 0 # trivial
```
References

• Readings in hardware/software co co-design / Giovanni Micheli, Rolf Ernst, Wayne Wolf, San Francisco: Morgan Kaufmann Publishers, c2002