A Logic-based Approach for Hardware/Software Co-design[1]

Hussein Zedan[2] and Antonio Cau[2]

1 Introduction

In the hardware industry, simulation is still all too frequently considered synonymous with verification. The design process usually consists of developing an implementation from a specification without the use of any formal proof techniques. Both are then simulated for a number of inputs (an approach known as *co-simulation* [1]). Bugs discovered are removed and the simulation process is repeated over again. However, formal verification cannot completely replace the existing simulation approach. This is because simulation provides powerful and more accessible tools for rapid prototyping and testing. What is needed is an approach where the design process is soundly based upon formal techniques, but includes integrated support for simulation. This combination would bring more reliability within an environment which does not require a complete changeover from current practice.

In this paper we describe a sound technique with a supporting tool, **AnaTempura**, for hardware/software co-design. Using our technique, we can validate and analyse system's behaviours of interest. The validation and analysis are performed within a *single* logical framework using Interval Temporal Logic (ITL) [2, 3] and its executable subset, Tempura [4]. Behavioral properties such as safety, liveness, timeliness, are expressed in ITL as theorems which can then be validated and tested *compositionally*.

In particular, using our technique, we are able to provide support for

- Hardware/software system level migration of existing *legacy* software solutions giving designers feedback on their design choices, such as hardware/software partitioning and CPU and scheduler selections.
- Bridging the gap that currently exist between high level specification/design and implementation. In particular, for a Verilog environment, we show how to bridge between behavioural Verilog and its structural gate level implementation.

2 Fundamentals

2.1 Interval Temporal Logic

As we mentioned earlier, our proposed approach is based on a single logical framework whose underlying logic is Interval Temporal Logic. In this section we provide a short description of the logic but more detailed exposition may be found in [4].

Interval Temporal Logic is a flexible notation for both propositional and first order reasoning about periods of time found in descriptions of hardware and software systems. It can handle both sequential and parallel composition unlike most temporal logics. It offers powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and timeliness.

The syntax of ITL is defined in Table 1 where μ is an integer value, a is a static variable (doesn't change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol and p is a predicate symbol. An interval is a sequence of states.

Expressions	*Formulae*
$e ::= \mu \mid a \mid A \mid g(e_1,\ldots,e_n) \mid \iota a{:}\, f$	$f ::= p(e_1,\ldots,e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid \mathsf{skip} \mid f_1 ; f_2 \mid f^*$

Table 1: Syntax of ITL

The informal semantics of the most interesting constructs are as follows:

- skip: unit interval (length 1).
- $f_1 ; f_2$: holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval.
- f^*: holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds.

[1]Funded by the U.K. Engineering and Physical Sciences Research Council (EPSRC) through the Research Grant GR/M/02583

[2]Software Technology Research Laboratory, De Montfort University, The Gateway, Leicester LE1 9BH, England, {zedan,cau}@dmu.ac.uk

For example, in an interval, if the variable I *always* equals 1 and in the next state the variable J equals 2 then it follows that the expression I + J equals 3 in the next state:

$$[\Box(I=1) \wedge \bigcirc(J=2)] \quad \supset \quad \bigcirc(I+J=3)$$

With these operators we can define the usual temporal operators $\Box$, $\Diamond$ and $\bigcirc$. These constructs enables us to define programming constructs like assignment, if then else, while loop etc. [4]. These constructs define what we call **concrete ITL**.

2.2 Computation, Behaviours and Properties

The process of modelling a system, albeit sequential or concurrent, timed or untimed, a suitable computational model must be established. We take the view that a computation defines mathematically an abstract architecture upon which applications will execute. A *system* is a collection of *agents* (which is our unit of computation), possibly executing concurrently and communicating (a)synchronously via communication links. Systems can themselves be viewed as single agents and composed into larger systems. Systems may have timing constraints imposed at three levels; system wide communication deadlines, agent deadlines and sub-computation deadlines (within the computation of an individual agent).

At any instant in time a system can be thought of as having an unique *state*. The system state is defined by the state variables of the system and, for concurrent system, by the values in the communication links. *Computation* is defined as any process that results in a change of system state. An agent is described by a computation which may transform a private data-space and may read and write to communication links during execution. The computation may have both minimum and maximum execution times imposed.

It is important to note that when we talk about *system* we do not make any distinction between software or hardware. We simply talk of a set of *agents* collaborating to achieve the desired behaviour. Some of those agents may be realised (or implemented) in software and some in hardware.

A *behaviour* in our model is defined as a sequence of states, i.e., an interval in ITL. Hence, a behaviour could be finite or infinite. A behaviour is called *full* behaviour if it contains all the state variables of the system otherwise it is called *partial*. A partial behaviour can be obtained by hiding some state variables (formally it is a projected behaviour over state variables).

A property *P* can be either a *state* or *temporal* ITL formula, i.e., a set of behaviours. A general classification of properties is readily available: **safety** (*something bad does not happen*) and **liveness** (*something good will eventually happen*) property.

2.3 Logical Framework for Hardware/Software Co-design

Our logical framework for Hardware/Software Co-design is illustrated in Fig. 1. We can either start from

1. a single description at an appropriate level of abstraction. At the highest level of abstraction we make no distinction between software or hardware components and the system's description is expressed as a formula ITL. The formula describes what the system does, desired behavioural properties and constrains. System design is then achieved through 'correct' stepwise refinement of the description by applying appropriate 'sound' refinement laws. The choice of an appropriate law depends on various aspects, including the target architecture and performance issues. Each refinement step adds more implementation details towards the final realisation of the system. In our refinement environment therefore we have two levels of representations: An abstract level which uses ITL as given in Table 1 above and a concrete level represented in **Tempura code**. Analyse/simulate the Tempura code to identify suitable candidates for software and hardware implementation (see Sec.3). Modules chosen for software implementation are translated into equivalent modules in suitable programming language such as C, C++, Ada, etc. Modules chosen for hardware implementation are dealt with in Sec. 4.2.
2. a pure software description (C/ADA/JAVA etc.). We can transform part (or full if you wish) of this description, after analysis, into a Tempura description that can be transformed again into a hardware description (Verilog). See Sec. 4.1 for an example.
3. a pure hardware description (Verilog). We can transform part (or full if you wish) of this description, after analysis, into a Tempura description that can be transformed again into a software description (C/ADA/JAVA etc.). See Sec. 4.2 for an example.
4. a mixture of 1., 2. and 3., since we have a single logical framework we can handle this easily.

For the transformation process in each step we use a refinement/abstraction calculus that allows us to systematically calculate the desired system description. The refinement relation $\sqsubseteq$ is defined on a system: A system $\mathcal{X}$ is *refined* by the system $\mathcal{Y}$, denoted $\mathcal{X} \sqsubseteq \mathcal{Y}$, if and only if $\mathcal{Y} \supset \mathcal{X}$. A set of sound refinement laws have been derived [3] to transform an abstract system specification into concrete systems. The abstraction calculus is just the inverse of the refinement calculus.

Two observations are in order:

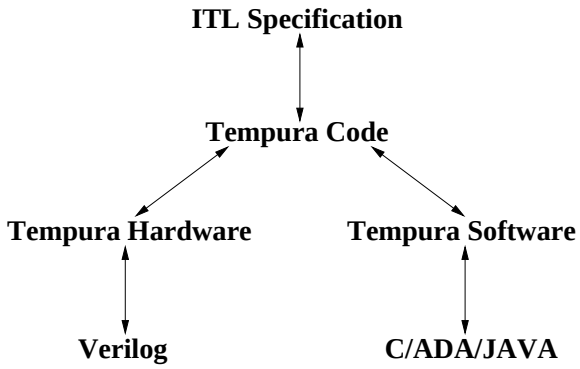

Figure 1: Logical Framework

1. Once we completed the formal specification phase, various properties could be proven about the specification itself. This can provide an extra assurance that the final specification meets the required informal requirements.
2. At each refinement/abstraction step, we can simulate the resulting (sub)system. This gives some guidelines on the choice of the subsequent refinement/abstraction rules.

3 Analysis

A fundamental issue in our approach is the ability to capture a possible behaviour of a (sub-)system. Once the behaviour is captured then we can assert if such behaviour satisfies a given property. And as a property is a set of behaviours, *satisfaction* is achieved by checking if the captured system's behaviour is an element of this set. We are not dealing here with the formal verification of properties which requires that all possible behaviours of system satisfy the properties. The formal verification of these properties may also be performed using an ITL verifier. We are only concerned with validating properties which requires that only interesting behaviours satisfy the properties.

The states of a (sub-)system to be analysed are captured by inserting *assertion points* at suitably chosen places. These divide the system into several *code-chunks*. Properties of interests are then validated over this behaviour.

Our general framework can be systematically described as follows.

1. Establish all desirable properties of the system under consideration and express them in Tempura.
2. Identify suitable places in the code and insert assertional points.
3. Using Tempura, check that the behaviour satisfies the desired properties.

Establishing system properties can be a hard task, however we suggest to follow the main characterisation of properties given above, namely safety, liveness and timing properties. Obviously, some level of understanding of the (sub-)system under consideration is assumed. These properties could be invariants that need to be true at all levels of system's abstraction.

The locations of assertion points could be chosen, for example, at the entry and exit points of a procedure or function. In this case assertions are in fact *pre*- and *post*- conditions, and what we are asserting is: If the system starts at a state satisfying the *pre*- condition then it terminates properly in a state satisfying the *post*- condition.

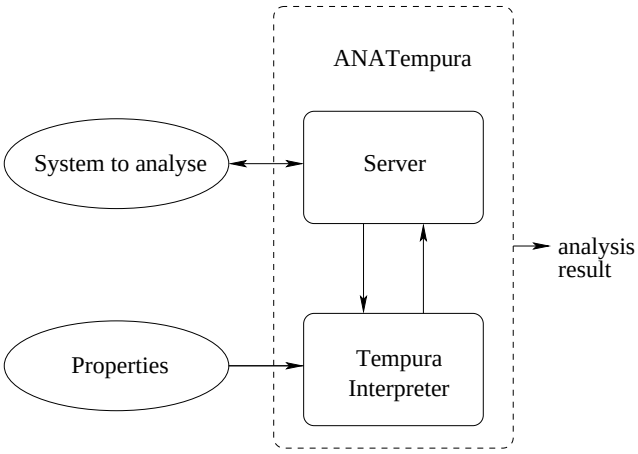

Figure 2: Basic Functions

We have designed and implemented a tool, known as **AnaTempura**, that support the approach described above. Figure 2 shows the general structure of the tool. The inputs are the system description (either source code plus assertion points or an ITL specification) and the properties we want to check. The result of the analysis is whether the properties hold for the system. Optionally the behaviour of the system can be animated.

4 Applications

4.1 Legacy Hardware/Software Partitioning

The sorter consists of 2 sensors for detecting respectively the class of a letter and whether a new letter has arrived. Furthermore it has 2 solenoids for respectively holding up a letter temporarily (solenoid 4) and switching the direction of the tray (solenoid 3). A fragment of the C code is shown below.

```
scan_csensor (&class_sensor);
if (class_sensor < 2)
   {
     assertion("class", 1);
     SolOff(4); Delay(delay4,1);
     SolOn(4);  Delay(delayF,4);
     scan_lsensor (&letter_sensor);
     assertion("lsens",letter_sensor);
      if ( !YellowSet )
        { Delay(delay3A,2); SolOff(3);
          Delay(delay3B,3); YellowSet = 1; }
   }
else
   {
     assertion("class",2);
     SolOff(4); Delay(delay4,1);
     SolOn(4); Delay(delayF,4);
     scan_lsensor (&letter_sensor);
     assertion("lsens",letter_sensor);
      if ( YellowSet )
        { Delay(delay3A,2); SolOn(3);
          Delay(delay3B,2); YellowSet = 0; }
   }
```

The delays in the code are crucial in that they ensure that once a first class letter has been detected it ends up in the first class tray. The problem occurs when migrating the software to a new hardware platform. As the delays are implemented in software and the speed of the new machine is different from the old one, our software solution becomes invalid.

A test and change cycle was adopted to the delay till the sorter works again. The results of these change were also used to implement the delays in hardware to avoid having the problem in the future. For this test/change and the hardware implementation we inserted assertion points in the code.

AnaTempura was used to check properties of the sorter, i.e., what is the correct value of the delays in order for the sorter to work correctly.

Since the hardware we need is simple, i.e., just a timer we don't design it but use of the shelf ones. But it should be stressed that the AnaTempura/ITL approach is able to produce a high level specification of the hardware. This specification can be used to design the requested hardware.

4.2 Bridging the Abstraction Gaps in the Verilog Environment

Before we discuss the way how to bridge the abstraction gaps in Verilog we will discuss the various abstraction levels as discussed in [5].

1. Cycle-Accurate (scheduled) behavioral description (CAB).
   This is the highest level of description and is very close to the level of description as produced by the AnaTempura/ITL approach. The description that the system in a clock-cycle-by-clock-cycle way, specifying the behaviour that is to occur in each Verilog state. A Verilog state is the computation between two clock events. These clock event statements appear to be at the same place as our assertion points.
2. Finite State Machine and Datapath (FSM-D).
   This is also known as register transfer level specification. A datapath can do computations, and store results in registers, A typical datapath contains basic modules like registers, adders and comparators etc. Each such module can be either described using behavioral combinational statements (assign) or combinational statements using gate primitives or a mixture of both. A finite state machine will control the datapath.
3. Combinational module description (CM). This comes in three flavours:

- Structural gate level description.
  Lowest abstraction level (i.e., the most concrete level). Primitive gates connected by nets. Examples of gates: nand, not, or, etc. Nets consists of wire, wired-and, wired-or and trireg connections.
- Behavioral description.
  Is an abstraction of how a module works. The outputs of the module are described in relation to its inputs, but no effort is made to describe how the module is implemented in terms of structural logic gates.
- A mixture of structural gate level and behavioral description, i.e., a description where part of the combinational module has already been implemented in logic gates.

We would like to move from the top level CAB description to a low level CM description via the FSM-D description. But in order to do that we must have a semantic for each level. We aim to provide an ITL-based semantic. We will use AnaTempura to check whether the semantics is correct.

The idea is as follows:

- Insert assertion points in the Verilog specification.
- Describe the semantics of the Verilog specification in Tempura.
- Use AnaTempura to check that the behaviour generated by the Verilog specification is the same as the behaviour corresponding to the Tempura specification.

Once we have fixed the semantics of each level we can use refinement to move between the various abstraction levels of Verilog. This also give a formal underpinning to hardware design in Verilog.

An ITL-based semantic for the Behavioural description in CAB is given in [7]. An example is the following Verilog specification of a 4 bit adder.

```
module four_bit_adder (A4, B4, SUM5, NULL_FLAG);
input          A4, B4;
output         SUM5, NULL_FLAG;
wire     [3:0] A4, B4;
wire     [4:0] SUM5;
wire     [2:0] CARRY;
  one_bit_adder  Bit0 (A4[0], B4[0],     1'b0, SUM5[0], CARRY[0]);
  one_bit_adder  Bit1 (A4[1], B4[1], CARRY[0], SUM5[1], CARRY[1]);
  one_bit_adder  Bit2 (A4[2], B4[2], CARRY[1], SUM5[2], CARRY[2]);
  one_bit_adder  Bit3 (A4[3], B4[3], CARRY[2], SUM5[3], SUM5 [4]);
  nor   #1 Nor_for_zeroflag (NULL_FLAG, SUM5[0], SUM5[1], SUM5[2], SUM5[3], SUM5[4]);
endmodule

module one_bit_adder (A, B, CARRY_IN, SUM, CARRY_OUT);
input          A, B, CARRY_IN;
output         SUM, CARRY_OUT;
   assign #1  {CARRY_OUT, SUM} = A + B + CARRY_IN;
endmodule

module test;
    reg  [3:0]  A4, B4, CARRY_IN;
    wire [4:0]  SUM5;
    wire        FLAG;

    initial begin
        $monitor("!PROG: assert sum:%d:%d::", SUM5, $time);
        #20 A4 = 0; B4 = 0;
             while ( (A4<=14) || (B4 <=14) )
             begin
              if (A4>14)
                begin #20 A4=0;  if (B4<=14) B4=B4+1;  else B4=15;  end
              else #20 A4=A4+1;
             end
        #20 A4=0; B4=0; #1 $finish;
    end
   four_bit_adder adder(A4, B4, SUM5, FLAG);
endmodule
```

The specification is straightforward, i.e., we use a 'nor' gate and 4 one bit adders. The one bit adders are described behaviourally. The 'nor' and one bit adders each have a propagation delay of 1. The initial will feed the 4 bit adder with appropriate values every 20 time units.

The semantics of above Verilog program is given below as a Tempura program. First we read the first 4 assertion points (representing the states when the adder is using undefined inputs). The always block is the actual formal semantics. The boxed line is the part that takes checks whether the sum computed by the Verilog program is equal to the sum computed by the Tempura version. Figure 3 shows a screen dump of the test.

```
...
define four_bit_adder(A4, B4, SUM5, NULLFLAG) =
{  exists CARRY0,CARRY1,CARRY2 :
    {   CARRY0 = ((A4[3] + B4[3] + 0) div 2) and
        CARRY1 = ((A4[2] + B4[2] + CARRY0) div 2) and
        CARRY2 = ((A4[1] + B4[1] + CARRY1) div 2) and
```

```
        SUM5   =  [ (A4[0]+B4[0]+CARRY2) div 2, (A4[0]+B4[0]+CARRY2) mod 2,
                    (A4[1]+B4[1]+CARRY1) mod 2, (A4[2]+B4[2]+CARRY0) mod 2,
                    (A4[3]+B4[3]+     0) mod 2  ] and
        nor (NULLFLAG, SUM5[0], SUM5[1], SUM5[2], SUM5[3], SUM5[4])
    }
}.
define test() = {
    exists X,T,S,A,B,SUM5,NF :{
         list(SUM5,5) and stable( struct(SUM5)) and
        {T=0 and S=0 and A=0 and B=0 and  veri_dot()};no_ch_4(A,B,T,S);
        {check_sum(X) and T:= atime(X) and S:=aval(X) and no_ch_2(A,B)};
        {check_sum(X) and T:= atime(X) and S:=aval(X) and no_ch_2(A,B)};
        {check_sum(X) and T:= atime(X) and S:=aval(X) and no_ch_2(A,B)};
        {check_sum(X) and T:= atime(X) and S:=aval(X) and no_ch_2(A,B)};
        always ( check_sum(X) and T:= atime(X) and S:=aval(X) and
                 four_bit_adder(inttobits(A),inttobits(B),SUM5,NF) and
                 if (atime(X)-T>1) then
                 { if strint(S)~=bitstoint(SUM5) then false and
                   A:=(A+1) mod 16 and
                   B:=if A mod 16 <= 14 then B else B+1 }
                 else { A:=A and B:=B }
               )
    }
}.
```

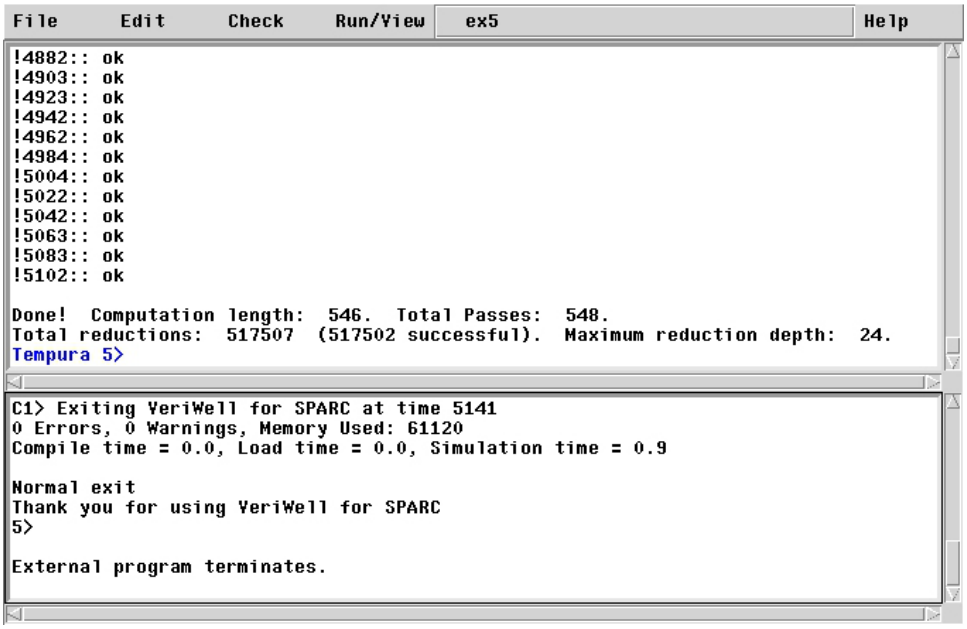

Figure 3: Screen dump of 4 bit adder

References

[1] **INSYDE**. *Technology Assessment*, **INSYDE**-Deliverable 1.1, *Application guidelines*, **INSYDE**-Deliverable 1.2, *Selection of a telecom application*. **INSYDE**-Deliverable 3.1, 1994, ESPRIT P8641.

[2] B. Moszkowski. *A Temporal logic for multilevel reasoning about hardware*. IEEE Computer 1985;18:10-19.

[3] A. Cau and H. Zedan. *Refining Interval Temporal Logic specifications*. In M. Bertran and T. Rus (eds.) *Transformation-Based Reactive Systems Development*, LNCS Vol. 1231, pp. 79-95, 1997.

[4] B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge Univ. Press, Cambridge, UK, 1986.

[5] D.E. Thomas and P.R. Moorby. The Verilog Hardware Description Language. Kluwer Academic Publishers, 4th Edition, 1998.

[6] J. Dimitrov. *Compositional reasoning about events in Interval Temporal Logic*, Proceeding of JCIS 2000, 675-678.

[7] J. Dimitrov. *ITL Semantics for Behavioural Verilog*, digest of IEE Workshop on Hardware/Software Codesign, 2000.