System Level Design and Verification Using a Synchronous Language

Gérard Berry
Esterel Technologies
France
gerard.berry@esterel-technologies.com

Michael Kishinevsky
Intel Corp.
Hillsboro, OR
michael.kishinevsky@intel.com

Satnam Singh
Xilinx
San Jose, CA
satnam.singh@xilinx.com

ABSTRACT

Synchronous languages such as Esterel, Lustre, Signal, and others were originally developed for safety-critical embedded software and compiled into C. They have recently been extended to hardware with new language features and compilers to RTL. Contrary to traditional HDL languages (Verilog, VHDL) and recent system-level languages (SystemC, System Verilog), they have well defined formal semantics, which facilitate bug avoidance using correct-by-construction compilation and verification techniques.

The tutorial will demonstrate what the synchronous language offers for the modeling, design, analysis and implementation of systems that comprise hardware and software. It will be based on Esterel. Esterel models have proved to be useful for rapid design space exploration and verification at system level, without resorting to detailed implementation and slow bit-level event-based simulation. We show how to model control-dominated IP blocks at a higher level of abstraction and how to use the target C code or RTL in conjunction with other system-level tools. Case studies include examples of design space exploration by synthesizing equivalent hardware or software from the same Esterel description, with formal verification of safety properties such as bus protocol conformance. We conclude with a review of future research directions.

1. INTRODUCTION

Modern control-dominated designs implement complex communication protocols and contain multiple interacting finite state machines (FSMs). FSM specifications are typically represented using explicit state graphs based on case-statements and written either in RTL-level Verilog or VHDL for hardware implementation or in C for software implementation. Designing networks of communicating FSMs using state transition diagrams is error prone, as the designer needs to manage lots of communication signals between the FSMs and ensure their correct synchronization over time. Hierarchical description of control is not possible in Verilog or VHDL. As a result, minor modifications in the specification can lead to major changes in the design. For example, waiting for three concurrent signals that can arrive in any order instead of two requires five more states and fourteen more transitions. This makes debugging difficult, limits model reuse, and leads to hidden communication bugs that might be detected very late in the design flow, or even remain undetected. Traditional control-dominated design productivity is relatively low.

Recently there have been significant development in the area of system-level languages based on C++. SystemC [1] is used for specification and modeling of systems at the RTL and system level. It is based on providing special C++ libraries and a simulator oriented on specifying RTL and system-level designs. SpecC [2] is an extension to C and is targeted to the system-level design and manual refinement. Both systems allow to describe software and hardware components within single framework and co-simulate them, which might be an important source of productivity for system-level exploration. On a negative side semantics of neither systems is formally defined (as both are based on C++ to start with) and hence the specification is hardly suitable to formal verification. Also there is no special abstraction layers for specifying control-dominated designs and design of FSMs is still done at a pretty much the same level of abstraction as in RTL.

Esterel [3] is a high-level formal synchronous language created more than 15 years ago to program reactive systems at a cycle-accurate behavioral level. It encompasses state sequencing, signal emission and reception, concurrency, and preemption structures to drive the life and death of control component behaviors in a hierarchical way. The original textual language was later complemented by the SyncCharts hierarchical automata graphical formalism [4] (now called Safe State Machines or SSMs in Esterel). Textual and graphical specifications can be freely mixed. Esterel has a strong semantic basis that defines the behavior unambiguously and is appropriate for synthesizing either hardware or software with exactly that behavior. Esterel provides access to formal verification and test generation systems for design validation.

The Esterel Studio tool suite includes an Esterel compiler that translates programs into C for simulation or software implementation and into Verilog or VHDL RTL-logic for hardware synthesis. It also includes tools for graphical and textual design capture and visualization, sequential circuit

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

optimization, formal property verification, test generation, and project documentation. Esterel has been used for modeling, validation, and code generation in various industrial domains (hardware, telecom, avionics, automotive etc.).

One can use Esterel to first implement or simulate a system entirely in software, validate it, and then automatically turn it to hardware. Such a flow has many desirable properties, including the ability to keep a single specification for hardware and software while formally guaranteeing the same behavioral properties of the design. Furthermore, one has a lot of flexibility to partition the system description so that some of it is realized in hardware and the rest is mapped to software.

This new approach provides a promising way for designing control systems overcoming drawbacks of the current low-level RTL/C flows, since it has powerful design primitives, formal mechanisms to ensure correctness, and powerful synthesis algorithms to guarantee implementation quality.

2. BASICS OF ESTEREL LANGUAGE

Esterel is based on an orthogonal and hierarchical set of sequencing, concurrency, and preemption primitives, To demonstrate them, let us consider the following small example:

> *Write to memory as soon as* `Addr` *and* `Data` *have arrived. Wait for memory* `Latency` *before iterating. Restart behavior each* `Replay`*.*

This specification can be implemented by the explicit automaton (state/transition graph) represented in Figure 1 (for conciseness, first letters of the corresponding signal names are used to denote the signals, e.g. `A` stands for signal `Address`). There are obvious drawbacks with this representation:

- the number of states and transitions explodes due to concurrency of `Addr` and `Data` signals (imagine three concurrent signals!);
- replay can occur from every state, hence repeating transitions labeled with `R`;
- the write operation `W` is copied three times as `A` and `D` can occur in any order;
- an explicit counter `X` needs to be introduced to count down latency of the write operation.

Expressing this automaton in a hardware description language like VHDL, Verilog, or System Verilog, requires tedious enumeration of all states and transitions, without any additional level of abstraction or conciseness.

Alternatively, in HDLs, one can write a sequential netlist for this automaton. This approach requires state encoding and logic synthesis for deriving next-state and output functions. The corresponding RTL netlist is cumbersome to design and is clearly not a high-level model. A conventional implementation of automata in C or other software languages does not offer a higher level of abstraction.

Esterel describes the automaton in a hierarchical and modular way — a must for the specification of larger system. The program is as follows:

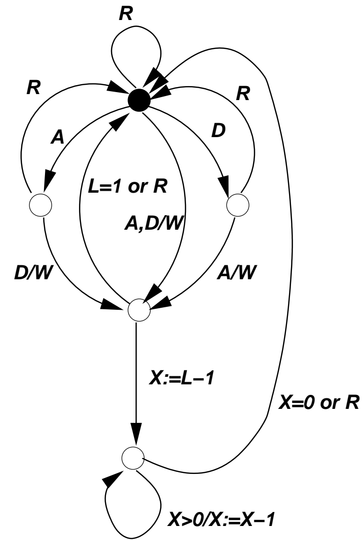

Figure 1: FSM for the specification

```
loop
   abort
      { await Addr || await Data };
      emit Write(funcW(?Addr,?Data));
      await Latency tick
   when Replay
end loop
```

Here, `Replay` is called a *pure signal*: at any clock tick it can be either present (encoded as '1' on the wire) or absent ('0'). The `Addr` and `Data` signals are called *valued signals*; they have a status bit (referred to as `Addr` and `Data`, correspondingly) and a value (referred to as `?Addr` and `?Data`).

The behavior is looped. The body of the loop contains an "`abort when Replay`" statement that preempts its body as soon as `Replay` occurs, thus factoring out the effect of `Replay` on the whole basic behavior. The body of the `abort` statement has three lines. The first line contains two concurrent `await` statements that wait for arrival (in any order) of signals `Addr` and `Data`. As soon as both have arrived, the program emits a `Write` signal and calls the function `funcW` with the address and data values, `?Addr, ?Data`. Then, the program waits for the `Latency` number of clock cycles for the completion of the write operation. All elements of the specification being represented exactly once, no copying is necessary unlike in state graphs. States are represented implicitly within the `await` statements. Interaction with a data path is done through the `funcW` function call that updates values in the memory.

In Esterel, all constructs such as sequencing, concurrency, waiting, preemption, are fully orthogonal: they can be freely mixed at any level. For example, arbitrary large statements or modules can be embedded within a single `abort` preemption statement, to define once for all the behavior of a preemptive signal like `Replay`. The Esterel model is clearly more abstract than the RTL one. The model is faster and easier to write and change since things are written only once. As a result, the specification contains fewer bugs and these bugs can be caught earlier in the design process.

Furthermore, in the RTL implementation, Esterel often allows for better design trade-off and reuse than RTL level

netlist or RTL FSMs due to both higher-level language constructs and capabilities to change state encoding.

3. THE ESTEREL V7 LANGUAGE

The classic Esterel language is called Esterel v5 [5]. Initial experiments for large-scale system-level or hardware designs showed that it lacked prominent features:

- the ability to describe data paths by traditional equations;
- support for bitvectors and integer encodings;
- direct support for Moore machines (Esterel v5 is Mealy-oriented);
- full modular structure for code reuse support;
- support for flexible reset and clocking schemes.

Most of these deficiencies have been corrected in the new release of the language, Esterel v7. The extension is fully semantically sound, leaving the semantics of the original portion of the language mostly unchanged. Most of the new constructs can be semantically viewed as high-level macro statements added to the core Esterel language.

The native Esterel v7 compiler was recently developed by Esterel Technologies as part of Esterel Studio 5.0. We will briefly review some features of the new language. From now on we will refer to the Esterel v7 as simply Esterel.

An Esterel program is made up of declarative and/or imperative statements, which exchange information using signals. Each signal carries a presence or absence status and can carry a value belonging to its data type. Programs are organized by three types of named units:

- *data units*, which group data-related declarations;
- *interfaces*, which group communication signals;
- executable *modules*, which have an interface and a behavior defined by an executable statement.

Units form a hierarchical structure. A data unit can extend another data unit by adding more components; an interface can extend another interface by adding signals; and a module can extend another module by adding behavior. Furthermore, data objects can be directly declared in an interface or module, in which case the interface or module unit can also be considered as a data unit of the same name by forgetting the signals and the behavior. Similarly, a module can directly declare interface signals, and it can be considered as an interface of the same name by forgetting the behavior. This helps in keeping the declaration-to-statements ratio relatively low. A few features of the unit definition language are demonstrated by the example below:

```
interface MemReadIntf :
  extends data MemData;
  input ReadAddress : integer;
  output value ReadValue : Type;
end interface

interface DualPortMemIntf :
  port ReadPort  : MemReadIntf;
  port WritePort : MemWriteIntf;
end interface
```

The `MemReadIntf` interface *extends* the `MemData` data, i.e. it imports all its components (type of objects, memory size, etc.). The `DualPortMemIntf` interface declares *ports*, which are groups of signals structured by interfaces. Here, `ReadPort` has two signal fields, an input `ReadPort.ReadAddress` and an output `ReadPort.ReadValue`. For memory control, the status of `ReadAddress` acts as a read enable. The `ReadValue` output signal (of generic type `Type`) is declared value-only, which means that its status is always present.

The fragment below illustrates how equations can be freely mixed with imperative definitions within the same program text. It is extracted from a FIFO design.

```
{ // equation style
  sustain{
    ReadEmpty = Read and pre(FifoIsEmpty)
    FifoIsEmpty = if (?Entries = 0) }
||
  // imperative style
  every DeltaEntries do
    emit next ?Entries <= ?Entries
                   + ?DeltaEntries;
  end every
}
```

Two statements are executed in synchronous parallel (`||`). The first statement is a set of equations to be executed at each tick (clock cycle), as indicated by the `sustain` keyword. The equations involve a sequential operator `pre`: for a signal `S`, the expression `pre(S)` is initially false and then turns true if `S` was present at the previous tick. In hardware translation, `pre` is implemented by a flip-flop with proper context-dependent surrounding control. The concurrent imperative statement `every` computes the next value of `Size` in function of its current value and value of `Delta`. The expression `next S` refers to the next value of signal `S`. Hence, `emit next S` corresponds to a Moore-style emission of a signal at the next clock cycle.

Esterel supports signal arrays with indexes ranging from 0 upward as in C, and static or compile-time for loops for replications. An example below extracted from a dual-port FIFO shows how array elements can be computed using replication loops. Static loops are mapped by the compiler to `generate` loops in VHDL, `for` loops in C/C++, or unfolded if the target language does not support replications (e.g. blif logic synthesis format).

```
for i < 2 dopar
  sustain {
    FifoEmpty[i] = if (pre(?Entries) = i)
    FifoFull[i]  = if (pre(?Entries) =
                        Size - i)
  }
end for
```

We refer the reader to [6] for a detailed description of the Esterel v7 language and to [7] for its earlier proposal.

4. ESTEREL FLOW

Given an Esterel specification, the compiler generates either an RTL netlist for hardware synthesis or statically scheduled C-code for simulation or embedding. Translation to a netlist is syntax-directed: every statement of the language is mapped to a logic netlist box. Esterel Studio's traceability features help to relate logic nets to the source

code. Because Esterel has a clean formal semantics, its circuit translation can be formally proven to be correct using a theorem prover (such a proof was indeed done in [8]). Translation of Esterel to C-code can be done based on different principles: internal generation of an explicit automaton (subject to code-size explosion for large specification), printing a sorted netlist into C-code (works for programs of any size, relatively slow code), or static scheduling of the concurrent control-data flow graph representing an Esterel program (fast code, works for specifications of any size). Esterel v7 currently only supports netlist-based C code.

The most interesting part of the netlist translation is a *group-hot* state encoding that exploits the hierarchical program structure to provide a good balance between 1-hot encoding (large state, simple logic), and dense encoding (small state, large logic). Although a Esterel-generated FSM uses more sequential elements than a densely encoded one, most of our test cases show that its overall cell area is actually smaller at equal speed after running Design Compiler.

5. A HARDWARE/SOFTWARE CASE STUDY

5.1 A UART State Machine

In this section we present an example of a hardware/software trade-off experiment based around a peripheral controller which can be implemented either in hardware or software. We chose a reconfigurable fabric realized by Xilinxs Virtex^{TM}-II FPGA [9] and it is on this device that we perform the hardware/software trade-offs. We use a specific development board manufactured by Xilinx called the MicroBlaze Multimedia Development Board which contains a Virtex-II XC2V2000 FPGA.

Software threads execute on a 32-bit soft processor called MicroBlaze which is realized as a regular circuit on the Virtex-II FPGA. For the purpose of this experiment we need to choose an interface that runs at a speed which can be processed by a software thread running on a soft processor. We selected the RS232 interface on this board which has all its wires (RX, TX, CTS, RTS) connected directly to the FPGA (there is no dedicated UART chip on the board). Now we have the choice to read and write over the RS232 serial port either by creating a UART circuit on the FPGA fabric or by driving and reading the RX and TX wires directly from software.

The send and receive portions of an RS232 interface were described graphically using Esterel's Safe State Machine notation. A simplified version of the receive sub-component is illustrated in Figure 2. This version does not use hardware flow control.

The receive RS232 receiver starts in a state which waits for the RX input line to go low. This synchronous system will be provided with a clock which over-samples the RX input by 16 times the baud rate. When the RX input is determined to be zero during one of the over-sampling periods the system then makes a series of transitions that recognize a character and check the parity bit. Some of these operations are specified graphically although we represent the process of shifting in the newly read bit into a buffer using a textual macrostate.

The simplified model for the RS232 sender is shown in Figure 3. This state machine waits for a control signal to initiate the transfer and then proceeds to send the start bit and the bits of the character. A textual macro-state is used to extract a bit at a time from a temporary variable that represents the character to be transmitted.

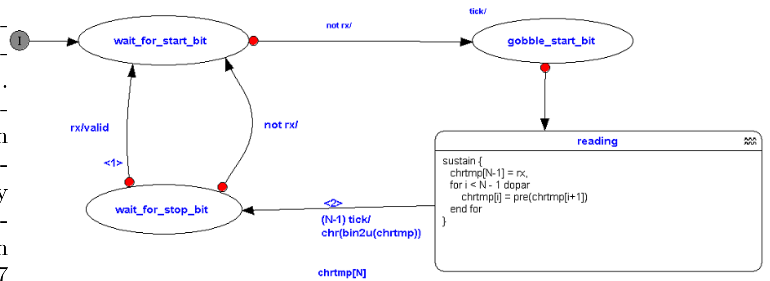

Figure 2: A simplified version of the RS232 receiver.

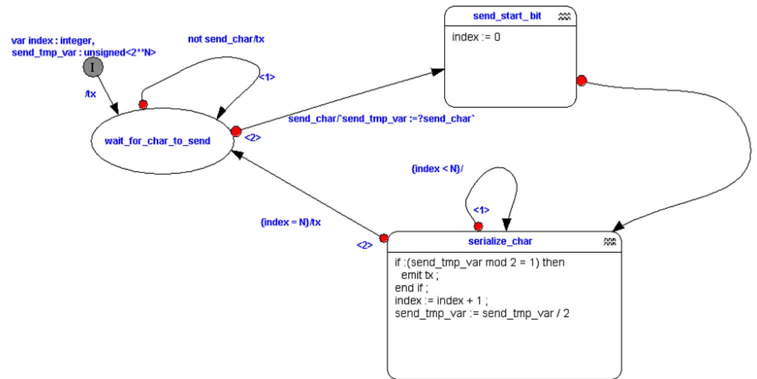

Figure 3: A simplified version of the RS232 sender.

It is possible to hierarchically build layers on top of these descriptions to add additional features e.g. a FIFO for the read and send channels. Graphical and textual descriptions can be freely mixed with the graphical descriptions being automatically converted into their equivalent Esterel textual equivalents. The actual RS232 interface that we use in this tutorial includes a FIFO for reading and writing and a more sophisticated policy for determining when the start bit has arrived.

The send and receive portions of the UART can be composed together to form a complete UART design, as shown in Figure 4.

5.2 A Hardware Implementation

The state machine described in the previous section can be synthesized to either hardware or software. The Esterel simplified RS232 description shown in the previous section was synthesized to RTL level hardware using Esterel Studio. We generated RTL VHDL output which was then submitted to the XST synthesis tool which forms part of Xilinx's implementation tool chain. This ultimately resulted in an implementation netlist which is shown placed and routed on a XC2V1000 FPGA in Figure 5.

The basic implementation uses 111 look-up tables (a look-up table can implement any four input one output combinational function) and 92 flip-flops and operates at a maximum frequency of 125MHz on the XC2V1000 FPGA.

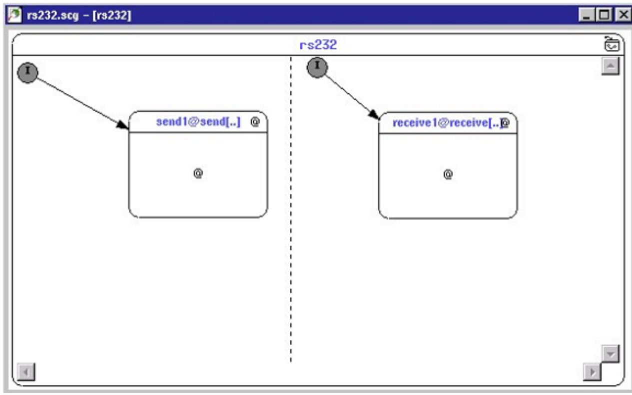

Figure 4: The top level RS232 UART design.

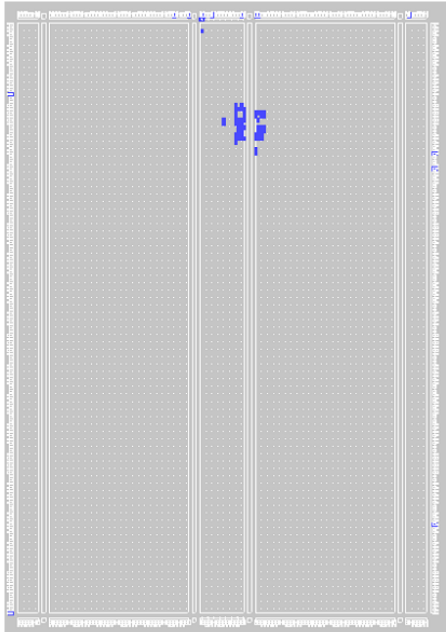

Figure 5: Hardware implementation of the UART.

5.3 A Software Implementation

The same Esterel simplified RS232 description shown before was synthesized to embedded C code using the Esterel Studio. The generated C code was used with Xilinx's Embedded Developer Kit to make a software realization of the UART which executes on a soft 32-bit microprocessor called MicroBlaze. The RX input and TX output are made available to the software through a simple general purpose I/O peripheral which allows memory mapped I/O for the RX and TX pins. This ultimately resulted in an implementation netlist which is shown placed and routed on a XC2V1000 FPGA in Figure 6.

This system was also tested on actual hardware and was able to deal with serial data at up to 9600 baud in software using the soft MicroBlaze processor clocked at 50MHz. Examples of other systems that are currently being realized on FPGAs using Esterel include peripherals (either the entire peripheral or just the bus interface) and hardware implementations of high speed serial protocols [10].

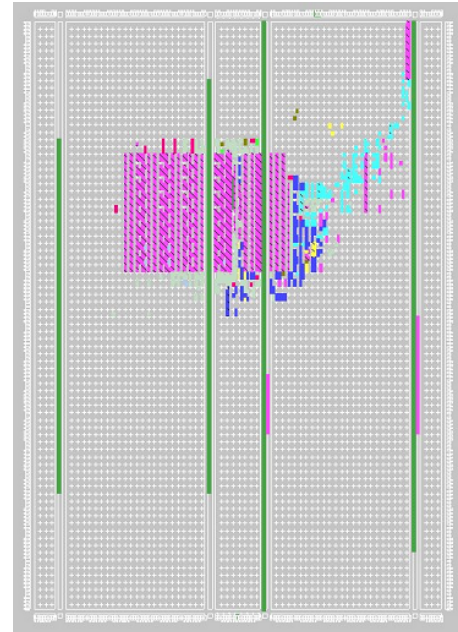

Figure 6: Software implementation of the UART.

It is of course possible to implement the same design directly in a conventional hardware description language like VHDL or Verilog. However, the Esterel description allows the behavior of a control dominated system to be expressed at the higher level of abstraction and to efficiently synthesize the specification into either hardware of software. An HDL implementation requires the designer to take care of many details of how FSM states are represented and how transitions occur by using unwieldy nested case statements. Not only does the Esterel flow provide a more productive method for specifying and implementing state machines but the discipline it imposes on these descriptions makes it easier to apply advanced automated verification techniques.

5.4 Co-Design

One could partition a system in Esterel such that some components are realized in hardware and others in software. The co-design capabilities of Esterel makes it easy to experiment with different architectures to determine a good balance of area versus performance. Communication between hardware and software can be modeled down to the level of bus transactions using OPB arbiter models that we have developed for Esterel. Alternatively they can be left as abstract communications based on events which are then fleshed out using back-end tool flows. For example, one could generate CoWareC [11] or SystemC [1] and use the capabilities of CoWare's N2C to perform interface synthesis to flesh out abstract communications in Esterel in terms of a specific bus-based transaction.

The UART implementation shown here could easily be implemented across hardware and software. For example, the FIFO feature could be implemented in software for both the send and receive channels and the raw low level serial operations performed in hardware. One could design the complete hardware/software system in one language and verify the whole system including the hardware/software interaction using a single methodology based on static analysis.

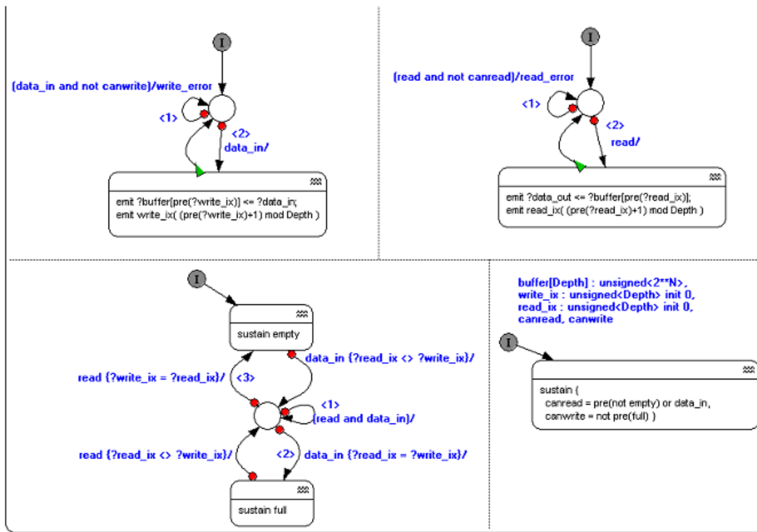

Figure 7: The FIFO used in the full UART design.

6. VERIFICATION

In previous work [12] we have reported successful formal verification of formal properties of an OPB bus interfaces designed in Esterel. This work showed that hardware circuits implementing bus-based protocols are amenable to sophisticated static analysis such as model-checking which are facilitated by having a formally based design description in Esterel. We now make the case that it is also possible to do system-wide static analysis to prove properties about the interaction of hardware and software. To test the co-verification capabilities of Esterel we selected a FIFO component that resides at the interface of hardware/software decomposition. Indeed, this component could reside in either hardware or software. The top level implementation of the FIFO is shown in Figure 7. It comprises several threads which control when data can be written in the FIFO and read from the FIFO.

The verification methodology in Esterel is based around the notion of synchronous observers. Rather than writing formal properties in a special mathematical language or as assertions in some special language one just develops more regular state machine descriptions. These descriptions receiver the same environmental input as the system under test and they observe the progress of the system check interesting properties. One can then check to see if a property is valid informally by simulation or formally by performing a static analysis.

For the FIFO implementation we produced a check to ensure that the FIFO never returned any uninitialized values (shown textually in Figure 8). Within 30 seconds on a 3GHz Pentium 4 PC the Esterel Studio system was able to prove that this FIFO implementation never returned an uninitialized value. We then produced a broken version of the FIFO and we were able to show within two seconds that the FIFO implementation violated the property. The user could debug the implementation either in Esterel through a graphical simulator that plays a simulation trace that exposes the bug, or through execution of the generated embedded C with an automatically generated scenario input file or for hardware through simulation using an automatically generated VCD file.

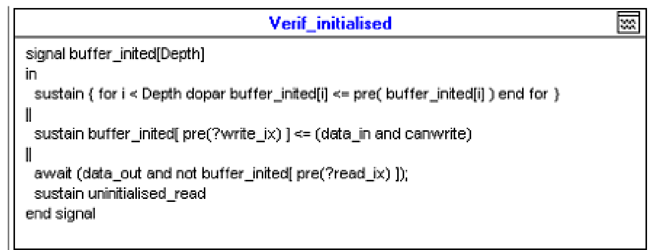

Figure 8: Initialization property for the FIFO.

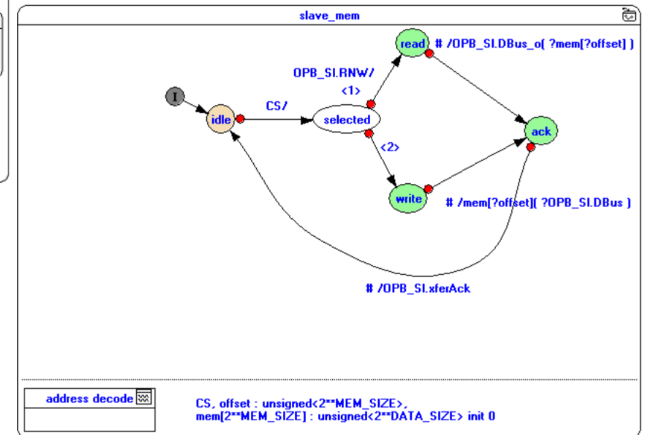

Figure 9: A typical CoreConnect OPB Slave.

Esterel has proved to be convenient for the specification, implementation and verification of protocols. Figure 9 shows a typical state machine for a slave peripheral used on IBM's OPB bus which is a component of the CoreConnect™ [13] IP bus standard. This simple peripheral supports reading and writing a value encapsulated by the peripheral. The description is broken into two parts. The upper part of the diagram describes the state transitions that can occur when a bus transaction for this peripheral is initiated. The lower part of the diagram performs some calculations to facilitate address decoding and selection for this peripheral. This description is not just a specification of the slave's behavior, it can also become the implementation by using the Esterel tools to translate it into VHDL or Verilog. We have implemented several CoreConnect peripherals which have been described in Esterel and then completely mapped to working FPGA implementations.

An important property of OPB slave peripherals is that they should acknowledge the bus transfer within 16 cycles (or request an extension). Using Esterel safe state machines we were able to specify a simple synchronous observer that checks that an OPB acknowledge signal is emitted within 16 cycles after a slave is selected by the arbiter. We were then able to use the built-in verifier to formally prove that a given slave always acknowledged a transfer within 16 cycles. The verification took less than two seconds. This is a powerful result since it shows that this peripheral can never be the source of a bus timeout error which can cause a system to crash.

Rather than writing individual properties of OPB slave in-

terfaces we can instead *model* the important aspects of the OPB arbiter behavior in Esterel and then use this model to help verify the behavior of OPB peripherals. This factors out all the common properties that any OPB peripheral should possess. Such a model was produced and the verification system was still able to prove bus time-out properties very quickly (again within two seconds). This shows yet another use for Esterel: to model IP blocks to facilitate verification. We could also produce an arbiter implementation from our description but this is not required since the arbiter is already available as an optimized IP block in our library.

7. CONCLUSION

We have shown how embedded software and hardware can be described in a synchronous language which can be synthesized to either embedded software or hardware. The quality of the embedded software was known to be good from previous research work done on Esterel and the initial results for the quality of the RTL hardware code is promising although there are minor points that still need improvement. Our experience with Esterel on a few control-dominated examples in the industrial settings demonstrated the following benefits:

- *Ease of selecting the implementation media*. Two key advantages of having a unified model for describing hardware and software are the ability to divide up a system description into different hardware/software partitions to explore performance trade-offs and the ability to analyze and verify a complete system that comprises hardware and software.
- *Correctness*. Esterel generated RTL passed all required validation tests. Embedded tools for formal property verification allows to check properties very early in the design cycle. Embedded sequential verification tools allows us to check correctness of the synthesis algorithms and is a foundation for checking correctness of late manual changes.
- *Abstraction*. Esterel specification enables bug avoidance by construction due to the use of high-level primitives. The compiler takes on the burden of generating many of the control signals, which would be explicit in RTL. On an actual design, the Esterel specification was 5 to 10 times smaller in code size compared to the original VHDL. Esterel helps to manage the complexity of the design by hierarchical partitioning.
- *Quality*. We could often achieve some area and/or delay reduction for the control-dominated examples exploiting the advantages of the group-hot state encoding.

This paper reports promising initial results for the later advantage showing proofs of hardware/software systems which would have been very difficult to perform using conventional techniques. As technologies like platform FPGAs becomes more common and more designers face system level and embedded systems and concurrent programming challenges the utility of more powerful and abstract design and analysis tools like Esterel will also increase. In particular, these systems seem suited for control-based operations, protocol implementation and co-verification.

Raising the level of abstraction in the design is not coming for free. As potential disadvantages we should mention the following. An RTL code compiled automatically from the higher level Esterel has less controllability (and less readable) than manually written RTL. It may become harder to control timing violations or to do manual changes to RTL (in late ECO). Esterel capabilities for describing and synthesizing a data path can be further improved.

We believe that the most interesting future research and development directions are as follows:

- Modular compilation for better scaling.
- Efficient verification supporting mixed control and data.
- Multi-clock support.
- Support for late Engineering Change Orders (ECOs) in silicon compilation.
- Support for flexible data and control encoding to improve synthesis quality.
- Improved readability and traceability of automatically generated RTL.
- Specific compiling techniques for fast software simulation.

"Virtex-II" and "MicroBlaze" are trademarks of Xilinx Inc.

8. REFERENCES

[1] *SystemC*, 2002, website at http://www.systemc.org.
[2] UCI, *SpecC*, 2002, website at http://ics.uci.edu/ specc.
[3] G. Berry, "The Foundations of Esterel," in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, ser. Foundations of Computing Series, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press, Aug. 2000.
[4] C. André, "Representation and analysis of reactive behaviors: A synchronous approach," in *Proc. CESA'96, Lille, France*, July 1996.
[5] G. Berry, *The Esterel v5_91 Primer*. Draft book, available at http://www.esterel-technologies.com, version 3, August 2000.
[6] *Esterel Studio 5.0 reference manual*, Esterel Technologies, 2003.
[7] G. Berry and M. Kishinevsky, "Hardware Esterel language extension proposal," Tech. Rep., August 2000, available at http://www.esterel-technologies.com.
[8] K. Schneider, "Embedding imperative synchronous languages in interactive theorem provers," in *Proc. Int. Conference Application of Concurrency in System Design*, June 2001.
[9] *Virtex^{TM}-II Platform FPGA Handbook*, Xilinx Inc., December 2000.
[10] Xilinx, *Aurora Technology Overview*, 2003.
[11] *CoWare*, 2003, website at http://www.coware.com.
[12] S. Singh, "Design and verification of CoreConnect$^{\text{TM}}$IP using Esterel," *the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods, L'Aquila, Italy*, 2003.
[13] IBM, *The CoreConnect^{TM}Bus Architecture*, 1999.