Towards A C++-based Design Methodology Facilitating Sequential Equivalence Checking

Philippe Georgelin
STMicroelectronics
Crolles, France
philippe.georgelin@st.com

Venkat Krishnaswamy
Calypto Design Systems, Inc.
Santa Clara, USA
venkat@calypto.com

ABSTRACT

It has long been the practice to create models in C or C++ for architectural studies, software prototyping and RTL verification in the design of Systems-on-Chip (SoC). These models are written in C++ primarily because it is possible to achieve very high simulation speeds, but also because it is productive to code at high levels of abstraction. In this paper we present a modeling methodology that continues to exploit the inherent advantages of writing models in C++ while ensuring that they are usable for formal verification of RTL through the use of sequential equivalence checking technology. An industrial case study is presented to show the validity of the approach.

Categories and Subject Descriptors

I.6.5 [**Simulation and Modeling**]: Model Development – *Modeling methodologies*

General Terms: Verification, Languages

Keywords

Modeling Methodology, Sequential Equivalence Checking

1 INTRODUCTION

C and C++ models are created by SoC design teams to serve different purposes during design including architectural exploration, software prototyping, supplying customers with executable prototypes, and verification of RTL models. These models generally call for some kind of simulation of the system being designed. Since the tasks to which the models are put differ widely in purpose, the C++ models created for each of these can vary in abstraction level across the axes of functional accuracy and detail in communication.

Traditionally, engineers responsible for writing these C/C++ models have paid little attention to the modeling methodology, in terms of maximizing code reuse for activities above and beyond the purpose to which their model is put. The guiding principle behind coming up with these models has been to achieve the maximum possible simulation speed. Simulation speed is largely a function of the level of abstraction of the model along the axes of behavioral accuracy and detail in inter-block communication.

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2006, July 24–28, 2006, San Francisco, California, USA.
Copyright 2006 ACM 1-59593-381-6/06/0007…$5.00.

New tools emerging in the EDA industry take C++ models as design inputs. These include both formal verification tools such as sequential equivalence checkers [1] as well as high-level synthesis offerings. The emergence of these tools makes the case for paying attention to modeling methodology. The object of such a methodology would be to enable the use of these tools, in particular sequential equivalence checking, while continuing to leverage the advantages of modeling in C++.

The fundamental underpinning of sequential equivalence checking technology is that it must build a hardware abstraction of the design described using C++. For this, it is important to use an unambiguous modeling style from which a tool can easily construct an abstract hardware model. In this paper, we describe how to construct C++ models that compose to form system level models at different levels of communication abstraction, achieving high simulation speeds while using an unambiguous, hardware friendly coding style.

The main facets of such a methodology are:

- separation of the modeling of functionality from that of inter-block communication
- use of language constructs and idioms from which hardware can unambiguously be extracted
- maintenance of a notion of correspondence between block level hierarchies and functions or classes in C++

This methodology is dependent on the ability to separate design functionality distinctly into computation and communication aspects. The approach scales well for algorithmic applications such as signal processing or image processing. It is also applicable for modeling some facets of networking designs. In applications where parallelism must be modeled at a very low level and is an integral part of the computation (e.g. an out of order instruction pipeline), this separation is not as easy or natural to obtain. This is true of highly control dominated structures commonly seen in high performance CPU’s.

The remainder of this paper goes into detail on these aspects of modeling. We also describe a signal processing use case where such a modeling methodology was employed, ultimately enabling the use of sequential equivalence checking.

2 ORTHOGONAL COMMUNICATION AND COMPUTATION

In this work, the system being modeled is assumed to be composed of multiple functional (computational) blocks that communicate with one another. Each functional block is modeled using a C/C++ function or class. Our methodology requires that

correspondences be maintained between these functional blocks and hierarchical boundaries in the RTL.

The level of detail in communication across functional blocks is a first order determinant of the simulation speed achieved by the system model. The level in communication detail increases from architectural models to software prototyping models to verification models. By separating the modeling of functionality and communication, it becomes possible to retarget the same functional code to differing levels of communication abstraction.

Thus a given piece of functional code can be leveraged in models that are un-timed, coarsely timed with API level interfaces, or timed with pin level interfaces. An example of the coarsely timed API level interface is that advocated by Ghenassia as part of his Transaction Level Modeling methodology using SystemC [2]. Each such model would achieve a level of simulation speed dependent on the level of communication detail.

In the discussion of modeling computation and communication, it is important to understand what languages/features are most convenient or appropriate to use for each. It is important to remember that SystemC is a C++ library that provides not only a good set of hardware-centric data types (used in computation), but also notions of hierarchy, process concurrency and event-based communication (all of which govern how blocks communicate with one another). In many design teams, home grown libraries, written in both C and C++, have evolved to provide essentially the same communication functionality. In this work, we use SystemC constructs to illustrate how one might achieve inter-block communication at different levels of abstraction.

2.1 Modeling Computation

In order to isolate the modeling of computation, there are essentially two techniques that we can use: the first is to use global scoped functions and the second is to use classes that contain the computation involved. The difference between the two approaches is mainly in how state is modeled.

We will use a simple 8-tap FIR filter to illustrate these ideas. In this discussion, we use the sc_int<> data-type provided by the SystemC library. Note however, that no calls to any event based simulation kernel appear in the core of the code. There are no signal updates, no attempt to model process concurrency, etc. This is fundamentally the model characteristic that enables this piece of functional code to be reused across models at different levels of communication abstraction (and, as a result, simulation speed). The C++ code segment in Figure 1 shows how the design would be modeled as a globally scoped function.

```
typedef int16 sc_int<16>;

void fir_filter (int16 in, int16 coeffs[8], int16& out)
{
   static int16 regs[8];

   for (int i = 7; i > 0; --i)
      regs[i] = regs[i – 1];
   regs[0] = in;
   int tmp = 0;
   for (int i = 0; i < 8; ++i)
       tmp = coeffs[i] * regs[i];
   out = tmp >> 16;
```

Figure 1: A C function implementation of a FIR filter

The shift register regs has been defined as a static variable because it holds state across function invocations. Another way to model the same functionality is to do so as a C++ class. One possible way to do so is as shown in Figure 2.

```
typedef int16 sc_int<16>;
class fir_filter {
public:
        fir_filter () {}
        virtual ~fir_filter () {}
        int16 run (int16 in, int16 coeffs[8])
{
                        for (int i = 7; i > 0; --i)
                                regs[i] = regs[i – 1];
                        regs [0] = in;
                        int tmp = 0;
                        for (int i = 0; i < 8; ++i)
                                tmp = coeffs[i] * regs[i];
                        return (tmp >> 16 );
                }
private:
        int16 regs[8];
```

Figure 2: An object implementing a FIR filter in C++

In this implementation, the shift register regs persists for as long as the fir_filter object does. The state in the shift register is thus maintained across calls. The advantage of using an object over a function is that this style lends itself more easily to systems that contain multiple instantiations of a given module. This is due to the fact that static variables in function scope are uniquely allocated. The only way then, to have two copies of a FIR filter modeled by the function fir_filter, is to disambiguate the storage by changing the function name of each instance.

2.2 Using Wrappers to Model Communication

Untimed models achieve communication between blocks simply by calling functions one after another. There is no attempt made to reconcile how function arguments may be allocated to hardware resources, or the number of cycles expended in communicating across blocks in hardware. A more detailed level of communication may call for a coarse notion of clocking. However, blocks may continue to communicate through an API level interface that is still agnostic of the exact hardware implementation of the communication. At yet another level of communication, hardware resources may be allocated to function parameters, but the block needs to be clocked at an RTL level of accuracy. The details of communication are best modeled within wrappers, each encapsulating the functional or behavioral code. In the code fragment that follows, we illustrate a wrapper that implements a pin level interface, while using a coarse clock that does not model the latency or degree of pipelining in an RTL implementation.

Indeed, in Figure 3, where we use SystemC constructs to implement the wrapper, we use an idealized clock, one cycle of which is sufficient to complete the invocation of the function implementing the FIR filter. Sequential equivalence checking technology is capable of ascertaining whether functionality is consistent across models even if they are timed differently. This enables users to write coarsely timed functional models to verify RTL that is pipelined and resource shared.

```
extern void fir_filter (int16 in,
int16 coeffs[8], int16& out);
SC_MODULE(fir_filter_mod) {
  sc_in_clk clk;
  sc_in <bool> rst;
  sc_in <int16> in;
  sc_in <int16> coeffs0, coeffs1 ... coeffs7;
  sc_out <int16> out;
  SC_CTOR (fir_filter_mod) {
    SC_METHOD (doit);
    sensitive_pos << clk;
  }
  void readData ();
  void writeData ();
  void doit ();
  private:
  int16 inp, outp, coeffsp[8];
};

void fir_filter_mod::readData () {
  inp = in.read ();
  coeffsp[0] = coeffs0.read ();
  ...
  coeffsp[7] = coeffs7.read ();
}

void fir_filter_mod::doit () {
  if (rst.read ())
    reset ();
  else {
    readData();
    fir_filter (inp, coeffsp, outp);
    writeData ();
  }
}
void fir_filter_mod::writeData () {
  out.write (outp);
}
```

Figure 3: A System C Wrapper around a fir filter

The advantage of separating computation from communication is clear. Models at differing levels of timing accuracy can essentially reuse the same core computational code. It is however, very important to adhere to this separation strictly because any bleeding of functionality into communication code can undermine all other models that use the core computational code.

3 HARDWARE INTENT IN C++

In order to unambiguously specify functionality in C++ such that a tool can statically create an abstract hardware model, it is necessary to follow certain rules. It is important to realize that these rules should be followed within that core piece of code that constitutes the functionality. Code within wrappers implementing communication in models not intended to be consumed by such a tool need not obey these rules. In the sections that follow, we touch upon the main aspects of these coding rules, and ways to mitigate their effect on the overall code.

The guidelines in the succeeding sections are not significantly different from those required by behavioral synthesis tools. The common aim of all these tools is to generate an efficient hardware representation that is ultimately restructured into an implementable piece of hardware subject to design constraints. Sequential equivalence checking technology too constructs hardware models of the designs it seeks to verify as equivalent. Given that sequential equivalence checking is a formal verification technology subject to runtime and capacity concerns, it is of paramount importance that a coding guideline enables extraction of efficient hardware models.

3.1 Dynamic Memory Allocation

Dynamic memory allocation has the property that it cannot, in general, be reasoned about statically in order to infer an efficient hardware representation. The success of sequential equivalence checking is dependent on the ability to generate efficient hardware structures in order to contain runtime and space requirements. Therefore, the guideline is that instead of using dynamically sized structures, it is necessary to size them so that they can be statically allocated. The call to malloc in the following code fragment must be removed:

```
int *a;
a = (int *) malloc (100 * sizeof (int) );
```

The hardware intent of this piece of code can more easily be understood if it is written as:

```
int a[100];
```

3.2 Pointer Aliasing

Pointer aliasing occurs when a single pointer can point to multiple locations in memory during its lifetime. Thus, the following indexing over elements of an array causes aliasing in the lifetime of the pointer x:

```
int *x;
int a[100], b[100];
for (x = a; x < (a +100); ++x)
  *a = 0xf;
for (x = b; x < (b + 100); ++x)
  *b = 0xf;
```

Some tools may provide support for a limited form of aliasing that applies specifically to indexing through elements of a single array. In general, however, it is impossible to extract a hardware abstraction from a more complex instance of aliasing such as the following linked list traversal:

```
my_struct *a, *first,
a = first;
while (a != NULL)
 a = a->next;
```

In general, hardware can be inferred from pointers that only point to a single object over their lifetime. Such pointers or references can be used to perform function calls by reference rather than by value for simulation efficiency.

3.3 Standard Libraries and Header Files

One of the advantages of programming in C++ is the ability to leverage the existence of stable libraries. An example of such a library is the C++ Standard Template Library (STL) that includes template containers such as lists, vectors, hashes, etc. The problem with these containers is that they dynamically resize themselves, and in order to do so, allocate memory at run-time. As seen in section 3.1, this is cannot be statically reasoned about. However, it is possible to build a statically sized library of template containers to provide model writers with similar features. Such statically

sized containers also provide more value in terms of hardware verification because resource over-runs or under-runs can be identified and seen to cause errors.

3.4 Variable loop bounds

Sequential equivalence technology generally reasons about designs whose latencies are fixed, or at least bounded. The latency of a block that is dependent upon input data frequently contains a loop whose bounds are data dependent e.g

```
for (int i = 0; i < x_range; ++i)
 a[i] = 0;
```

Here, x_range is an input to the functional block. The use of such data dependent loops, while technically synthesizable requires care. In order to successfully perform sequential equivalence checking, it may be necessary to supply a maximum loop unrolling limit, in addition to appropriately constraining the value of x_range during a given run of the SEC tool concerned.

4 CASE STUDY

We present a case study where we have applied the ESL methodology described in this paper. The case study is a video pipe subsystem, shown in Figure 4, that is part of a larger system-on-chip (SoC). We discuss the overall model architecture and how it fits into a system level SoC model used for algorithm evaluation. We then present the equivalence checking methodology.

4.1 Modeling Style and Design

The video pipe comprises four blocks: dct, idct, quant and iquant. In RTL, there is a hierarchy corresponding to each of these blocks. The functional model for the video pipe comprises a function for each of the blocks. The model itself is untimed, and the video pipe functionality is achieved at the top level by calls to the functions in order. The block diagram is shown in Figure 3.

The coding style used for the C models follows the principals laid out in this paper. Once the algorithm was proven, the RTL was generated using a high-level synthesis tool, subject to area and timing performance constraints specified by the designer. Compliance to the algorithm was proven by running the C model on realistic vectors. Subsequently, the hardware was automatically synthesized into RTL.

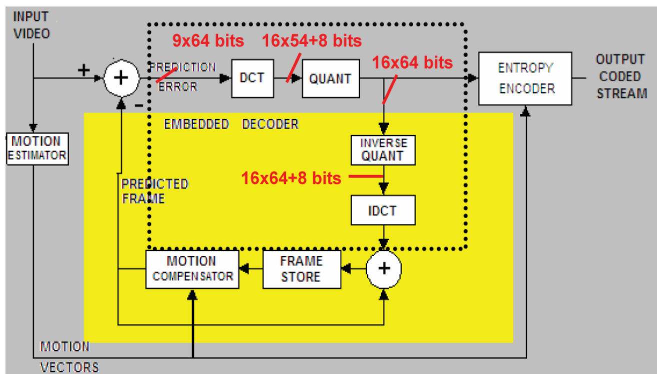

Figure 4: Video pipe block diagram

4.2 Equivalence Checking and Results

Equivalence checking was carried out at the block level, i.e. at the level of the dct, idct, quant and iquant. A pin accurate wrapper such as that shown in Figure 3 was created for each of these blocks. A coarse clock notion was employed whereby each function needed only one clock cycle to complete its operation. In the RTL implementation however, distinct latencies were allocated to each block.

Since a high-level synthesis tool was employed to build the RTL implementations, it was possible not only to automatically generate the wrappers, but also to create driver files into the sequential equivalence checking (SEC) tool. The driver files contain such information as clocks, resets, latency and throughput differences between the specification (C++) and implementation (RTL) models. Since the high-level synthesis tool built the RTL implementation, it knows the latency and throughput. As a result, it is possible to build a flow that automates such information transfer across tools.

During the running of the SEC tool, several counter-examples were generated. For the most part, these pointed to issues in the wrapper, or errors in specifying RTL design latencies and throughputs.

Once all issues were fixed, SEC was successfully performed on each of the blocks comprising the video pipe. The tool runtimes were under an hour for each of the four blocks, and full formal equivalence was proven.

5 CONCLUSION

By paying attention to the manner in which C++ code is written, we were able to show that it is possible to take a design from algorithm to verification using common design sources. In order to do so, it is necessary to define carefully how to write C++ code such that it leverages the simulation speed advantages inherent to that language, while enabling tools to infer hardware intent from the descriptions. We have been able to prove too, that it is possible to leverage the same code base over different C++ models ranging from architecture exploration to formal verification. The notion of wrappers was crucial to implementing this flexibility.

In this work, we have reused algorithmic C++ functions for RTL implementation via behavioral synthesis, as well as for sequential equivalence checking. In general, even if behavioral synthesis is not used to generate hardware, the technique of separating computation from communication can be used to verify manually written RTL. The advantage of sequential equivalence checking technology is that it is possible to reason about functional equivalence in the presence of latency and throughput differences. Therefore, the method of writing a simple, single cycle wrapper around a function and using that to verify a fully timed and scheduled RTL model is a fundamentally productive technique.

6 References

[1]David Maliniak, “Equivalence Checker Handles Sequential Logic”, *Electronic Design*, May 12, 2005

[2] Frank Ghenassia (ed), “Transaction Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems”, *Springer* 1st Edition, 2005