Incorporating Cores into System-Level Specification

Frank Vahid and Tony Givargis
Department of Computer Science and Engineering
University of California, Riverside, CA 92521
vahid/givargis@cs.ucr.edu, http://dalton.ucr.edu

Abstract

We describe an approach for incorporating cores into a system-level specification. The goal is to allow a designer to specify both custom behavior and pre-designed cores at the earliest design stages, and to refine both into implementations in a unified manner. The approach is based on experience with an actual application of a GPS-based navigation system. We use an object-oriented language for specification, representing each core as an object. We define three specification levels, and we evaluate the appropriateness of existing inter-object communication methods for cores. The approach forms the specification basis for the Dalton project.

1 Introduction

Increasing chip capacities has led to entire systems being implemented on a single chip. Pre-designed system components like bus controllers and data encoders, which previously took the form of integrated circuits (IC's), are now becoming available instead as intellectual property *cores* so that they may be incorporated onto a single chip with other components. Cores can be soft (synthesizable source code), firm (technology-independent netlists), or hard (technology-specific layouts).

Meanwhile, system synthesis has evolved to assist in converting a specification of desired system functionality into a collection of system components, some pre-designed and some custom designed. A key challenge that cores present to system specification relates to their *flexibile interfaces*. Specifically, before the advent of cores, pre-designed system components came as pre-packaged IC's with fixed interfaces. Thus, the approach for incorporating such a component into a system specification simply involved instantiating that component into a system-level netlist of components. A netlist requires fixed (or at best parameterized) interfaces. Cores, however, can have flexibile interfaces, meaning that we can vary the number, sizes and protocols of their external buses. Thus, continuing the past approach is too restrictive with regards to interfaces. Instead, we want to incorporate cores into a system specification in a manner that enables us to explore a variety of interfaces among those cores. Such an approach would complement existing industry efforts to build cores with flexible interfaces, e.g., the "bus wrapper" effort of the Virtual Socket Interface Alliance [1]. A first attempt to address the issues of system-level specification with cores, and interface exploration and synthesis for core-based systems, is being addressed in the *Dalton* project at UC Riverside.

In this paper, we describe an approach for incorporating cores into a system specification and for refining that specification towards an implementation. In Section 2, we define a three-level approach to system specification catering to cores. In Section 3, we describe the techniques necessary to refine the specification through the three levels. In Section 4, we illustrate the specification approach on a GPS navigation example. In Section 5, we provide conclusions.

2 System specification with cores

We have isolated three types of system specifications suitable for describing core-based systems, as illustrated in Figure 1. The first, referred to as method-calling objects, is the most abstract and best for early specification. The second, message-passing processes, represents an intermediate specification. It is commonly used as a system specification, and while adequate, does not provide as strong of core encapsulation as the first type. The third, structurally-interfaced components, represents the type of system specification commonly used today when structural components like cores are to be incorporated.

2.1 Method-calling objects

System designers commonly create an early system model using languages like C, C++ or Java. One of our goals was to define a system specification approach that made use of these existing languages rather than creating a new language or adding extensions to an existing language. The object-oriented model supported by C++ and Java seemed to be an excellent match for cores. Our system specification approach, however, can be implemented using any object-oriented language.

2.1.1 Active objects

In an object-oriented language, a *class* represents the definition of an object's data (state) and methods. A class can be *derived* from another class, meaning the former automatically includes the data and methods from the latter. A *method* is a procedure, declared as part of the class, that manipulates the object's data and serves as the interface to the object. An object must be *instantiated* by declaring a variable of the class type; multiple

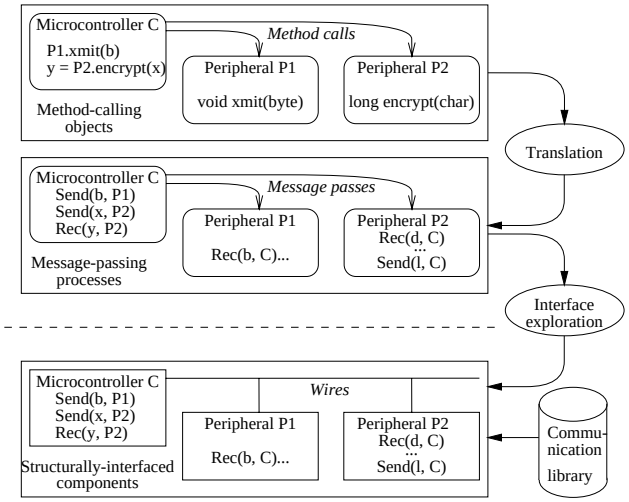

Fig. 1: Three levels of system-level specifications: Method-calling objects, message-passing processes, and structurally-interfaced components.

objects of the same class may be instantiated. A *passive object*, representing the traditional type of object, is a repository for data that is manipulated by calling the object's methods. On the other hand, an **active object** is an object that has or uses a thread of control as a part of its state. Once instantiated, an active object can be thought of as executing concurrently with other active objects.

The active object paradigm is an excellent match for a system-level specification. Not only does it provide a good means for managing system complexity, but it also provides an excellent means for incorporating cores early into system specifications. When incorporating a core, we have two main goals:

1. Encapsulate the core as a distinct component, providing access to the core's functions, while hiding the internal implementation details.
2. Keep the interface functional, not structural.

Active objects support both of these goals. Class definitions support the first goal directly. Methods support the second goal directly.

Describing active objects requires threads. Java has a Thread class as part of its API. For C++, we use Posix threads to implement a Thread class for C++. In either case, the Thread class has a *start()* method, which, in turn, invokes the *run()* method of the core class. Any class derived from the Thread class would result in an active object. We define a *method-based core* class as one describing a physical core but having functional internal behavior and a method-based interface with other objects.

For example, Figure 2 represents a method-based core class derived from Thread, written in Java. The

```
class UartFct extends Thread {

  public bit txd;
  private byte data;
  private boolean rdy = false;

  private synchronized void
    setRdy(boolean b) {rdy = b;}

  public void xmit(byte b)
  {
    setRdy(true);
    data = b;
  }

  public void run()
  {
    while (true) {
      if (rdy) {
        for (int i=0; i<8; i++) {
          txd = (data>>i)&1;}
      }
      setRdy(false);
    }
}
```

Fig. 2: A method-based core class for a UART.

run method, once activated, will loop continuously. It checks if data is ready to be sent, and if so, sends the data serially over an output port. Another object can interact with the UART by calling the *xmit* method, which receives a byte of data and sets a ready flag, so that the data would be sent by the executing *run* method. The *setRdy* method is synchronized, meaning that only one invocation of the method may occur at a time. The user of the core may instantiate and activate a core object as:

UartFct uartA = new UartFct();. uartA.start();.

The user may then send data x to the UART for serial transmission as:

uartA.xmit(x)

Note that a method-based core class may consist of a mix of both a functional interface (methods) and a structural interface (public data). Even in a system specification, there will often be the need for representing structural input and output, such as the bit output of the UART; the fact that the output is a bit is essential, and hiding this fact with a method is not beneficial. Also, a core may require multiple threads itself to support concurrency within itself.

2.1.2 Communication

A method call to an active object may be either synchronous or asynchronous. In a synchronous method call, the calling object blocks until the method returns. In an asynchronous method call, the calling object may continue execution as soon as the method parameters have been transferred.

Both types of communication are valid for cores and thus should be supported. An example of a synchronous call would be that of a core passing data to another core

that encrypts the data and returns it. The first core's object might call a method *encrypt* on the second core, waiting for the return value. In some cases there might not be any data returned by a method, but we might still want to block for synchronization purposes. On the other hand, an example of an asynchronous call would be that of a core passing data to a UART for serial transmission. The first core might simply pass the data to the UART and then proceed with other tasks while the UART sends the data serially.

While languages like Java and C++ support synchronous method calls directly, they typically do not support asynchronous method calls directly. Therefore, we use the technique shown in Figure 2 to achieve the same behavior. The method *xmit* is desired to be asynchronous, so instead of including the details of serial transmission in that method, we instead store the parameter and set a flag, where that flag is monitored by the *run* method and causes the actual serial transmission there. Therefore, *xmit* returns very quickly, thus making the communcation appear asynchronous.

We must also determine how the objects will be "connected." There are several alternative approaches for connecting and, hence, establishing communication between method-based core objects. We can describe two of those approaches as follows:

- *Hierarchical*: In this approach, a single master object transfers data to or from all the other objects, via method calls. The other objects, though executing concurrently, are all servants to the master.
- *Cooperating*: In this approach, each object transfers data to or from other objects; there is no obvious master.

The hierarchical approach matches well with many microcontroller-based systems we have examined. In such systems, an object is declared as the master object, usually a microcontroller. This object, in turn, instantiates servant peripheral objects, and can then call their methods. This approach is very simple but has the drawback that the servant objects cannot call methods of other objects.

In some systems, a more distributed form of communication may be desired, requiring the cooperating approach. In these cases, we declare all objects within one parent testbench object. An object may call methods on other objects, and thus may require a handle to other objects; those handles are passed during creation of the object. A microcontroller object would require handles to all its peripherals. Likewise, a peripheral object may require a handle to another peripheral with which direct communication occurs. It may even require a handle back to the microcontroller, perhaps to call an interrupt method. This approach is more general, but results in increased specification complexity.

2.2 Message-passing processes

This specification level corresponds closely to the communicating process model common in many earlier proposed approaches to system specification. The specification is based on a communicating sequential process model [2]. The model consists of several concurrently executing processes, which communicate via message passing. Message passing consists of send and receive procedures, in which data is sent from one process to another, and received by that other process. In blocking message-passing, the sender or receiver blocks, or suspends, until the data is transferred. In non-blocking message-passing, the sender may proceed immediately after sending the data, requiring queueing of the sent data until the other process receives the data.

Like method-calling objects, all message-passing processes preserve the flexibility of both internal behavior and external interfaces by representing both functionally rather than structurally. However, in a message-passing process specification, the communication among processes does not describe the data transformation occuring on the transferred data. In contrast, method-calling objects may be designed to clearly indicate the data transformation. For example, rather than a process communicating with another process P as follows: *Send(x,P); Receive(y,P);*, an object can communicate with another object O as follows: $y = O.encrypt(x)$.

Note that message-passing processes can be implemented in an object-oriented language by defining a channel object, having send and receive methods. Then, each object, when declared, requires a handle to a channel object, with one channel being shared among two objects, and the object can then call the channel's send or receive method to communicate with the other object.

2.3 Structurally-interfaced components

This specification level corresponds to a netlist of system level components, equivalent to a traditional system-level block diagram. Here, the external interface of each component is fixed, and wires connect components together. Internally, each component may still be described behaviorally.

Each message-passing channel may be mapped to its own structural bus, or many channels may be mapped to the same bus. Some techniques related to synthesis of physical buses from behavioral channels are described in [3, 4].

We have investigated techniques for preserving code readability in the refined code within each component by maintaining the appearance of message passing communication even though a structural bus is being used. Towards this end, we developed OOCL (Object-Oriented Communication Library) [5]. OOCL is a library of routines written in C and VHDL that can be used for com-

munication among software and/or hardware components without any underlying operating system support (which is what differentiates OOCL from other common forms of process communication). A user instantiates a channel for a particular bus protocol using existing constructs in C or VHDL. The user can then call Send and Receive routines from the rest of the code. Unlike the case in pure message passing, these routines have detailed underlying implementations specific to the bus and protocol for which the channel has been declared. The user, therefore, can achieve many of the specification goals of message passing communication, while using existing language constructs and describing detailed protocol communication.

2.4 Related work

Many researchers have proposed using object-oriented (OO) specifications for digital systems design, but none have focused particulary on cores, to our knowledge. Cores require unique details to be worked out as described above. Some researchers have focused on describing hardware using OO languages, while others have focused on OO system specifications.

Kumar [6] discussed capturing a hardware design using an OO language, including capture of registers, comparators, etc. A processor would then take the form of a transformation function that declares components as objects, and accesses those objects through method calls. Some research has focused on extending VHDL for OO models; a good summary is found in [7].

Others have focused on using object-oriented modeling as an early specification of desired system functionality, rather than as a method for describing existing register-transfer components. Each object can be implemented on a software processor or a hardware processor, with the mapping of objects to processors being many to one or one to many (e.g., [8, 9]).

Our goal can be seen as a mix of the previous two. We too want to specify existing components, but more abstract components representing entire processors and peripherals, rather than register-transfer-level components. Likewise, we too want to specify desired system functionality, but we also want to be able to encapsulate cores in that specification.

3 System refinement

Given a system specification consisting of method-calling objects, we want to refine that specification into a set of structurally-interfaced components, in order to provide a link with existing tools.

3.1 Refinement from objects to processes

The first step is to refine the method-calling objects into message-passing processes. We must convert the method calls as well as the methods themselves. The only way that the processes will be able to communicate is by sending and receiving data.

A method call takes the form of:

o1 = P.method1(i1, ..., in, o2, ..., om)

We can convert the method call by a sequence of sends and receives as follows:

Send(i1, P); Send(..., P); Send(in, P); Rec(o1, P); Rec(o2, P); Rec(..., P); Rec(on, P);

To be more efficient, we can send all input parameters at once:

Send(i1, ..., in, P);

and, likewise, receive all output parameters at once, requiring specialized but straightforward send and receive routines. This stage of the refinement process can be automated. A parser for the language can generate the send and receive routines as it encounters method calls.

A conservative approach to converting an object's methods themselves is to create a process for every one of its methods, in addition to a main process for the object's *run* method. Thus, an object would be replaced by a group of processes, where those processes share the object's data. In this case, the above sends and receives would specify the particular method's process (e.g., *Send(i1, P_method1)*), rather than the main process P as above. However, such a high degree of concurrency may not be necessary if the methods are called sequentially. In this case, several methods may be implemented with one process. If the sequence of method calls can be determined statically, then the process can simply consist of a sequence:

Rec(i1, ..., in);
o1 = method1(i1, ..., in, o2, ..., on);
Send(o1, o2, ..., on);
Rec(inputs for another method)
Call another method

Otherwise, the process would consist of a decode stage in which the address of the called method is received, and a branch stage in which the appropriate method is called. Note that recursion among methods in different processes would not be supported in such an approach. Again, given a parser for the language in use, this stage of the refinement process can be automated as well.

4 A GPS navigation example

We performed a case study of a real application in order to define and verify the above approach. The application is a navigation system based on GPS (Global Positioning System). The system is an Autonomous Vehicle Navigation System (AVNS). The AVNS uses Differential GPS coupled with a high speed Inertial Navigation System to accurately estimate its exact position

within centimeter-level of accuracy. This estimated position, along with a pre-recorded path, is fed into a control system that commands a steering-wheel stepper motor, as well as a speed control unit of a vehicle to maintain a steady speed along that pre-recorded path. We fully implemented the system using discrete system components. An Intel 486SX processor runs a real-time application program written in approximately 15,000 lines of C code. The system has been used to control a driver-less Ford Escort.

We also created a object-oriented specification of this same system, as illustrated in Figure 3. A microcontroller is connected to 10 peripheral soft cores. Six analog-to-digital converters (ADC's) feed the microcontroller with acceleration and rotational rates, sensed by the INS (Intertial Navigation System) unit, 100 times per second. The microcontroller integrates these rates to obtain high-speed position and velocity estimates of the vehicle. The GPS receiver and the radio modem units, connected via two UARTs to the computer, feed the microcontroller with very accurate positioning data once every second. The microcontroller computes vehicle position using this data with centimeter-level accuracy. The two estimates, one from INS and one from GPS are meshed together in a Kalman Filter, whose output feeds the controller. The controller, after computing a steering angle and throttle position, outputs commands to the steering-wheel stepper motor and the speed control units of the vehicle ten times per second.

The object-oriented specification consisted of implementing the UART, AD, PWM and Micro-controller cores, on the three different abstraction levels, method-calling, message-passing, and structural-interfaced, in Java. The method-calling implementation consisted of 640 lines of Java code, with an average of 100 lines per core. On the other extreme, the structural-interfaced specification consisted of 1700 lines of Java code, with an average of 250 lines per core. In the case of the Micro-controller cores, the Run method consisted of the actual navigation code. Instead, a Micro-controller core could have been implemented to emulate the actual 486SX processor.

Stepping through our three specification levels would result in the specification being refined into that shown in Figure 4. In this case, an interface configuration is chosen using a single system bus. All the cores now have structural interfaces. This specification can directly feed into existing synthesis tools, which would synthesize (for soft cores) or retrieve (for hard cores) implementations for each core.

There are two advantages to specifying this system using our three-level approach, rather than starting with the structurally-interfaced component description. First, the method-calling object specification is easier to comprehend and fits in with the common approach designers

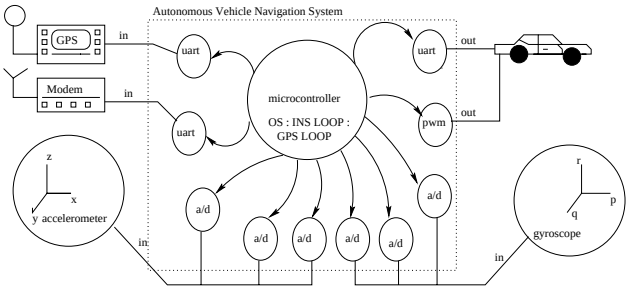

Fig. 3: The navigation system as an object-oriented specification having soft cores with soft interfaces (methods). The cores are represented as abstract models.

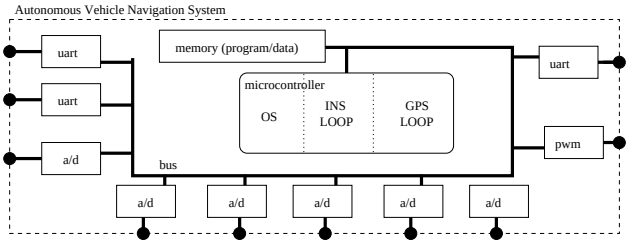

Fig. 4: The navigation system refined to have soft cores with hard interfaces, connected to a single system bus. The cores are represented as VHDL synthesizable models.

use today of building early system models in C++ or even Java. Second, the fact that the cores have functional, not structural, interfaces enables us to automatically explore a wide variety of bus interface configurations among those cores. Such exploration can significantly optimize important design metrics, such as power consumption, wiring size, and performance. In fact, we are developing a tool within the Dalton project to automatically explore a variety of bus configurations, and have found tremendous power optimization potential even for a single-bus configuration [10]. The configuration can be varied by the data bus size, by the way that large data is multiplexed over the bus, by the way that unused bus lines are padded, and by applying bus-invert. Early results are summarized in Figure 5; notice the large variation in power for different bus configurations. We have found that the optimal bus configurations varies per example, since each example has a unique combination of data being transferred over the bus (e.g., the AVNS has a particular scheduling of 12 bit and 8 bit data being transferred; other systems may have different bit sizes).

5 Conclusions

We have presented a three-level system specification approach supporting cores. The first level of method-calling objects encapsulates cores, while keeping inter-

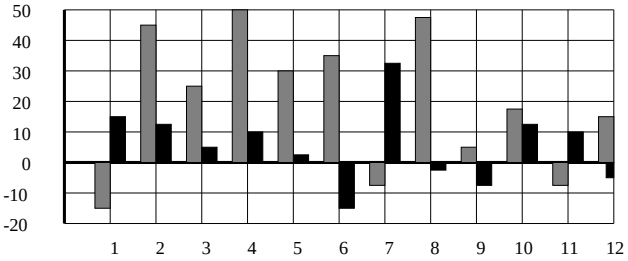

Fig. 5: Percent power improvement using standard (black) and bus-invert (gray) configurations and different bus widths (1..12) for the AVNS system.

faces functional and thus supporting interface exploration. The approach forms the basis of the specification approach being used in the Dalton project. Future work will include developing interface exploration techniques, made possible by cores and their functional interfaces.

6 Acknowledgement

A Design Automation Conference Graduate Scholarship and a NSF grant No CCR9811164 supported this research. We are grateful for their support.

References

[1] A. Cataldo, "VSI abandons plans for system-chip bus," *EE Times*, October 6, 1997.

[2] C. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[3] S. Narayan and D. Gajski, "Synthesis of system-level bus interfaces," in *Proceedings of the European Conference on Design Automation (EDAC)*, 1994.

[4] M. Gasteier and M. Glesner, "Bus-based communication synthesis on system-level," in *International Symposium on System Synthesis*, pp. 65–70, 1996.

[5] F. Vahid and L. Tauro, "An object-oriented communication library for hardware-software co-design," in *International Workshop on Hardware-Software Co-Design*, pp. 81–86, 1997.

[6] S. Kumar, J. Aylor, B. Johnson, and W. Wulf, "Object-oriented techniques in hardware design," *IEEE Computer*, vol. 27, pp. 64–70, June 1994.

[7] G. Schumacher and W. Nebel, "Inheritance concept for signals in object-oriented extensions to VHDL," in *Proceedings of the European Design Automation Conference (EuroDAC)*, pp. 428–435, 1995.

[8] S. Vercauteren, B. Lin, and H. D. Man, "Constructing application-specific heterogeneous embedded architectures from custom HW/SW applications," in *Proceedings of the Design Automation Conference*, 1996.

[9] N. Woo, A. Dunlop, and W. Wolf, "Codesign from cospecification," *IEEE Computer*, vol. 27, pp. 42–47, January 1994.

[10] T. Givargis and F. Vahid, "Interface Exploration for Reduced Power in Core-Based Systems" *International Symposium on System Synthesis*, 1998.