Modeling Asynchronous Communication at Different Levels of Abstraction Using SystemC

Shankar Mahadevan
Inst. for Informatics and Mathematical Modeling
Tech. Univ. of Denmark (DTU)
Lyngby, Denmark
sm@imm.dtu.dk

Tobias Bjerregaard
Inst. for Informatics and Mathematical Modeling
Tech. Univ. of Denmark (DTU)
Lyngby, Denmark
tob@imm.dtu.dk

ABSTRACT
There is a need for HDLs that operate across a wide range of abstraction levels. In asynchronous design, there is also a need for HDLs that are supported by standard EDA environments. This paper looks at the usefulness of SystemC for providing a means for these needs. We have used SystemC to implement basic communication channels, which can be used in a mixed mode environment, in which the individual modules might be at different abstraction levels. A range of channels has been designed, which implement specific asynchronous communication protocols, while simultaneously maintaining abstract communication function calls. The aim was to allow for the mode (abstract or protocol-specific) of each end of the channels to be independently chosen. This aim was attained, and a test environment demonstrating it, was developed.

Categories and Subject Descriptors
System modeling using high level language for asynchronous communication.

General Terms
Modeling, Standardization, Languages, Theory, EDA Tools.

Keywords
SystemC, asynchronous channels, system-level modeling, mixed-mode modeling, interface.

1. INTRODUCTION

SystemC is fast gaining prominence as a prime candidate for high-level system modeling and verification. It is based on *C++*, which allows to easily describing the models and algorithms while providing all the advantages of object oriented programming (OOP). On the other hand it has a synthesizable subset, which allows for its use in ASIC standard cell circuit design. In real life digital design environment the two activities are tightly bound and/or running in parallel and there is an increasing demand to converge on a single language. This will enable the ability to mix and match different blocks of design at intermediate step of the design process for testing and design iteration. SystemC provides constructs that allow modeling the system behaviour at a higher level and then unplugging selected blocks to be replaced with their real hardware (RTL) counterparts, while preserving the overall design approach.

Many other languages such as SpecCharts, SpecC et al exist for capturing system level behaviour. On the other hand the two industry standard digital design languages, VHDL and verilog lack high level primitives easily accessible for system modeling. These languages excel in implementation of blocks but at a much lower level which often doesn't provide the flexibility required for easy plug-and-play design iteration and testing at system level. In recent year many of the above language development groups have joined together in an attempt to support one language for all levels of abstraction and SystemC is emerging as a strong contender. SystemC has drawn on many of the lessons learnt during their development.

In this paper we have investigated and assessed the possibilities of using SystemC to model asynchronous inter-module communication at various levels of abstraction. Similar work has been done earlier by Michael Pedersen [1,2]. He used VHDL to model mixed-abstraction level communication channels. VHDL has some short comings however, in that there is no full support for data abstraction, it is cumbersome to switch between fully abstract channel and protocol-specific ones, and the simulation overhead is considerable.

In Section 2 the asynchronous background, the concept of mixed-mode modeling, and the basic SystemC constructs to be used are introduced. In Section 3 we present the structure of our design, and demonstrate the use of our channels in the various modes. Here the essential aspect is the possibility to simulate a mix of abstraction levels simultaneously, and to easily extend the provided channel package with new protocols. In Section 4 we present some results, mainly a more elaborate example, an asynchronous fifo, which is modeled using the channels, and show the results of some mixed-mode simulations. A conclusion is provided in section 5.

2. CHANNEL MODELING IN SYSTEMC

A channel is a means for communicating data from a source to its designated sink. When the source needs to *send* data, it commits to the communication and waits until the sink has performed the *receive* data operation. Thus a communication event also

© 2003 SoC Group, IMM-CSE, Tech. Univ. of Denmark

synchronizes the processes at either end of the channel. We have designed a number of different channels, which can be used to model the asynchronous communication between modules easily.

2.1 Asynchronous Channel Protocols

There are a number of well known protocols that can be used for asynchronous communication. Asynchronous channels use zero power when idle. In the work done for this paper, we have implemented 2 and 4-phase (*2ph* and *4ph*), push and pull versions of both bundled data (*bd*) and dualrail (*drl*) protocols.

Most of the illustrative examples in this paper are based on the *4ph_bd_push* protocol. A high *request* signal from the sender indicates that data is available to the receiver. The receiver responds with raising an *acknowledge* when it has accepted the data, after which a return-to-zero (RTZ) recover phase initiates during which first the sender lowers its request, then the receiver lowers its acknowledge.

In the *2ph_bd_push*, the RTZ recover phase is skipped. Toggling the request indicates that data is available, and toggling the acknowledge means that the data has been accepted. In the *4ph_drl_push*, the request is merged with the data, so that delay insensitive (DI) circuits can be implemented. Each bit of data is encoded onto two lines. Raising one line indicates that data is available and that it is a 0, while raising the other line indicates that data is available and that it is a 1. The rest of the protocol executes in a similar fashion to the 4ph_bd_push; the receiver acknowledges the receipt of data, and a RTZ recover phase is entered after which all signals are back to low. Details on the remaining protocols can be found in [1].

These varying forms of asynchronous communication need support for concurrency and handshake primitives for modeling. This support is inherent in communicating sequential processes (CSP) type languages. Our aim was to design and test communication channels supporting transactions through an abstract implementation of CSP-like *send* and *receive* commands as well as through one of many specific interfaces (e.g. 4ph_bd_push implementing actual *request*, *acknowledge* and *data* ports). It should be possible to make the choice of abstraction level independently for each end of the channel.

2.2 Mixed Mode Modeling

Considering a top-down design methodology; it is useful to make use of abstract communication methods, when starting the design process at the top level. In order to make the seamless transition from system-level specification to RTL implementation independently for different parts of the system, these communication methods should support mixed-mode communication. One module might still be communicating using abstract methods while another might have crystallized into RTL implementing a specific communication protocol. Figure 1 shows modeling at different levels of abstraction for communication. Here the producer is the source of data, the channel is the means of communication and the consumer is the sink. In our test setup

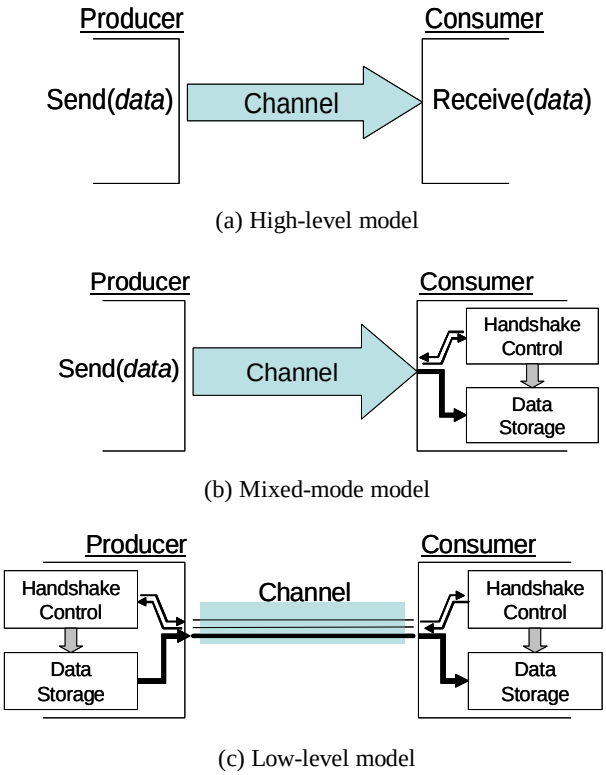


Figure 1: Channel usage at different levels of abstraction

we are using the same channel class for all three environments, the object being to show how a single channel can be used at all abstraction levels.

At a high level of abstraction (Figure 1(a)) the producer and consumer talk to the channel only via the abstract interface. The channel instance is bound to the producer's output port and the consumer's input port. Thus in order to send data, the producer calls the method *send(data)*. At the other end, the consumer calls the *receive(data)* method in order to acquire the data. The producer and the consumer are not aware of the actual channel implementation, but only the fact that they are bound to an object which implements the send and receive methods. The channel may implement any protocol, abstract or physical.

In the next design flow step, some of the blocks may be in RTL format while others are still algorithmic in nature, thus the need for mixed mode simulation. For this stage, we have built an RTL asynchronous consumer block. The overall system model now looks as shown in Figure 1(b). The consumer block has a handshake controller (HC) which responds to channel communication. The handshake implemented in the consumer corresponds to that implemented by a translator function (which translates between the abstract interfaces and the physical interfaces) within the channel. For mixed mode, the ports of the consumer are bound to the physical signals in the channel rather than to the abstract interface. The channel thus in this case is accessed via the abstract methods at the producer end, while being accessed via the physical wires by the consumer. This form of system modeling is useful in intermediate state of design where some blocks are more refined than others.

The final implementation in our top-down approach is the fully signal true behavioural model of the producer, the channel and the RTL consumer (Figure 1(c)). Here the producer's ports are directly tied to the appropriate handshake control signals and the data lines in the channel. A similar port bounding is done at the receiving end between the channel and the consumer. The virtual methods implemented in the previous high-level and mixed-mode simulation still exists within the channel, but are never activated. Thus the channel is effectively a set of wires.

2.3 SystemC

SystemC 2.0 has introduced a new set of features for generalized modeling of communication and synchronization. These are *channels*, *interfaces*, and *events* [6].

A *channel* is an object that serves as a container for communication and synchronization. Channels implement one or more interfaces. An *interface* specifies a set of access methods to be implemented within a channel. The interface class is a *virtual* class that may be programmed to provide methods such as *send* and *receive*. The functions specified in the interface class are virtual as well, and as such are not actually implemented within the class. An *event* is a flexible, low-level synchronization primitive that allows for synchronization between different processes.

The channel class implements the functions specified in the interface. To perform a data transfer, the data source and sink, at either end of the channel, bind themselves to the abstract interfaces, then simply invoke the required method specified in the interface (which is implemented in the channel). This is very useful at system-level as it alleviates the module designers from concerns of channel implementation. We have completed successful simulations for a number of different protocol specific channels in the setup described. We have thus shown the flexibility of our design, how the same channel class can be used for high-level system investigation as well as RTL design and an arbitrary mix of these. This is possible because of the interface construct of SystemC [4,5].

3. CHANNEL DESIGN

In this Section, we will describe the structure of our channel. The main idea is having protocol-specific channels inherit the abstract communication methods from a base channel. Also the mechanisms for translation between abstraction levels will be dealt with.

3.1 Object Oriented Hierarchy

Figure 2 shows the object oriented hierarchy of our design. The base channel, *channel_abstract*, is protocol independent and inherits the abstract interface classes *send_if* and *receive_if*,

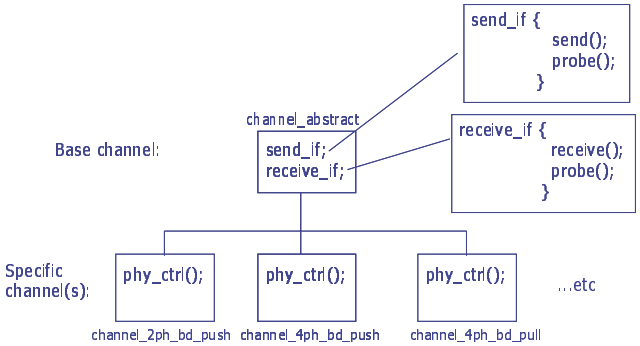


Figure 2: OOP hierarchy of channel classes

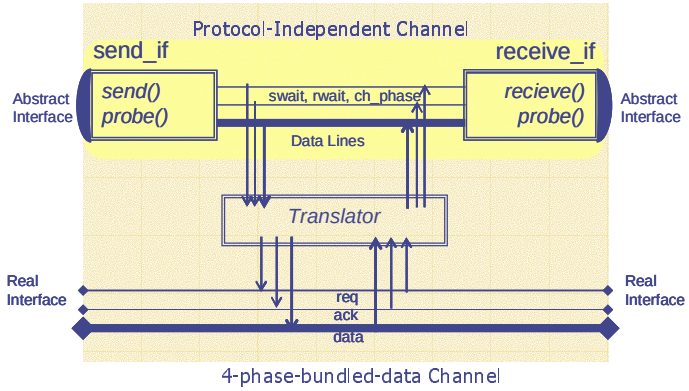


Figure 3: Protocol-specific channel example

which specify *send*, *receive* and *probe* commands. This completely abstract channel is one in which the signals of the channels would have no RTL meaning. The send_if specifies the *send* method for transmission of data and the receive_if specifies the *receive* method for reception of data. Each interface additionally specifies a *probe* method that detects pending transactions. SystemC allows flexible type-casting of the channel data [6] known as data templating. Thus the abstract channel model can transmit any type of data.

The more elaborate, protocol-specific channels, inherit this base channel class. In addition, they implement the ability to access the channel by direct wire manipulation. The control and data wires specified by a chosen protocol are defined in the channel. Figure 3 illustrates the concept of how the *phy_ctrl* process, shown in Figure 2, functions as a translator between the physical protocol and the abstract protocol. If the abstract interfaces are not being used, the user may access the wires directly. Thus the channel supports both the abstract and the real interface of the protocol. The translator drives the control and data wires according to the protocol, and the channel may thus be used for communication across abstraction levels. It is illegal to use both abstract and real interfaces at the same end of the channel simultaneously.

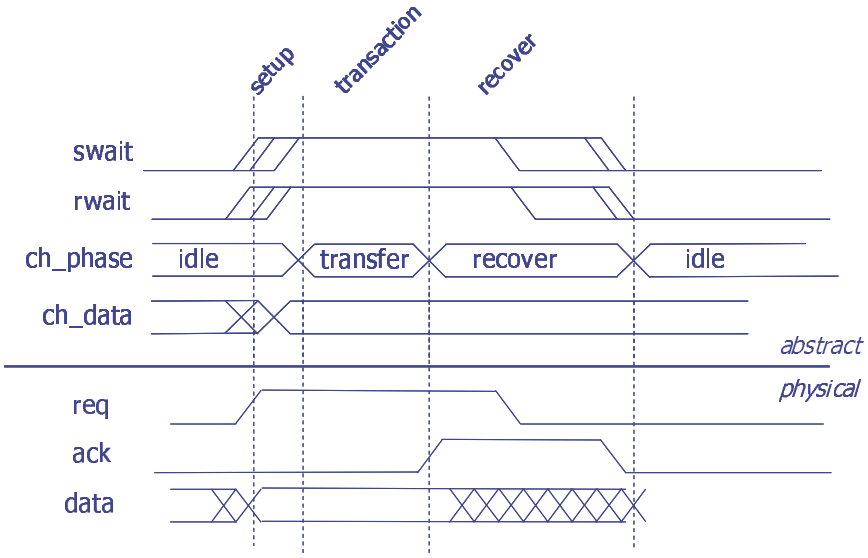


Figure 4: Abstract and physical protocol timing.

3.2 Implementing new protocols

The object oriented hierarchy makes the development of new protocol-specific channels easy and safe. All channels will share the same interface and abstract implementation. The abstract protocol only needs to be mapped to the specific protocol. Figure 4 shows the timing of the abstract protocol, and its mapping to a 4-phase-bundled-data-push protocol. The *swait* and *rwait* signals indicate that respectively the sender and receiver are ready and waiting for a data transaction. When both are high, the channel enters the *transfer* phase, during which the data transfer happens. When the receiver indicates that the transfer has happened, the channel enters the *recover* phase, during which the channel recovers, the swait and rwait signals return to low. Once this has happened, the channel re-enters the *idle* state.

The abstract protocol is designed with a mind for easy mapping to a wide variety of different physical protocols. The mapping to the specific protocol is a simple exercise in synchronizing handshake points in the timing of the two abstraction levels.

3.3 Channel usage

The flexible and seamless transition between abstraction levels is valuable during design iterations. The RTL implementation of one module may affect the function of another module, causing the need for the other to be re-implemented, starting at a high abstraction level. Thereby each system-level module in a design may move up and down the abstraction ladder a number of times, before the final design is at hand.

This further assists in code reuse. If a certain module is the bottleneck of a system, it might be feasible to limit a redesign to that module. A new algorithmic implementation of the module in question will be made. which can still be simulated in the RTL environment with the other modules, using the same channel.

4. RESULTS

A more practical example is the addition of data latching fifo elements between the producer and the consumer. The elements are fully synthesizable RTL blocks. This allows visualizing a pipelined process of data handling while using different modes of the channel among the intermediate blocks. Figure 5 shows one such configuration. Here the producer and the consumer are connected to the pipeline via the abstract interface while the elements are interconnected using physical signal ports. For each of the protocol-specific channel designs that we have made, we have implemented the pipeline, and shown working mixed-mode simulations.

Figure 6 shows selected results for some simulations. The waveforms have captured the signals/states of a 4ph_bd_push channel, connected in three different setups. In Figure 6(a) transition through states of the channel accessed only through the abstract interfaces is seen. One can see that the physical control signals *req* and *ack* are idle. Observe the values of *ch_phase*. The value 1 represents *idle* phase, 2 represents *transfer* and 3 represents *recover*. As described above, the signals *swait* and *rwait* signify the sender and receiver waiting. Once both are high the transaction is free to occur.

In comparison Figure 6(b) shows the channel in a configuration where the producer is accessing it through the abstract interface, while the consumer is implementing access through the physical protocol-specific ports. Both the physical and the abstract protocol signals are seen to be active. Figure 6(c) shows the case

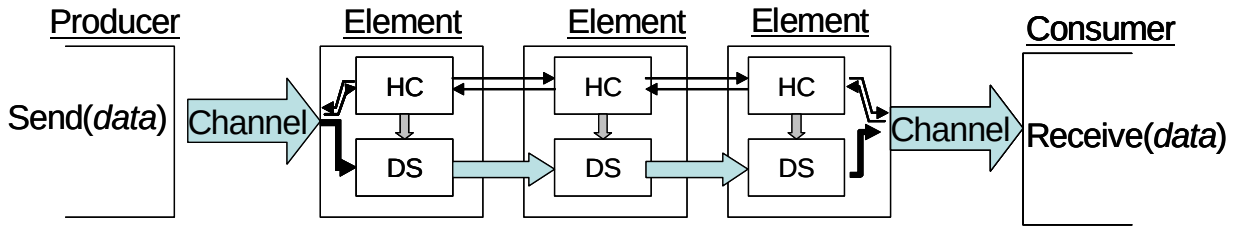


Figure 5: Model of pipeline.

where both producer and consumer are accessing the channel by means of the physical ports. The abstract protocol signals are idle. Please note that the delays between request and acknowledge signals in the physical interface seem to be zero. The reason for this is that the delay is in the delta step range. To our knowledge, SystemC v2.0 does not support non-blocking delays directly.

5. CONCLUSION

The availability of flexible and simple inter-process communication primitives is essential to the design of asynchronous circuits. It is also important that these communication primitives are seamlessly usable at all levels of abstraction and that they are available as part of a language which has standard EDA industry support. SystemC has emerged as robust tool for interoperable system-level design. It contains all the basic primitives to synchronous and handshake mechanisms. We have shown that it is possible to use the build-in constructs of SystemC to design communication channels that can be used to model asynchronous transactions at all levels of abstraction and in mixed mode environments.

We have simulated high level modules together with RTL implementations, and succeeded in bridging the traditional gap between high-level abstract functional models and low-level RTL code. The object oriented structure of our channel design makes it possible with minimum effort to extend the channel package with new protocol-specific channels. Thus, our channels are accessible through CSP-like function calls, as well as direct wire manipulation. Our test systems performed well at all layers of abstractions and for different modes of the channel. We have thus joined the capabilities of SystemC with modeling requirements of asynchronous channels.

SystemC provides an ideal platform for mixed-mode design and verification, where different modules may be at various stages of implementation. In the future, we hope to elaborate on our channel package, i.e. by exploring cross-protocol channels, bidirectional channels, and the use of channels as a tool in investigating on-chip networks. As for the SystemC language itself, the language standardization body is considering adding the capability of fork and join threads to the basic set of primitives. This would be welcome addition from an asynchronous design point of view and would supplement our current endeavourers in full system design.

6. ACKNOWLEDGMENTS

Our thanks to Professor Jens Sparsø for introducing to us issues in asynchronous channel modeling, Professor Jan Madsen for introducing SystemC and Michael Pedersen whose work on mixed-mode channels in VHDL has been an inspiration.

7. REFERENCES

[1] J. Sparsø, S. Furber, “Principles of Asynchronous Circuit Design” chap 8, Kluwer Academic Pub. 2001.

[2] M. Pedersen, “Asynchronous Design Using Plain VHDL in a Standard CAD-tool Framework”, ACiD-WG Workshop, Newcastle upon Tyne, 1999.

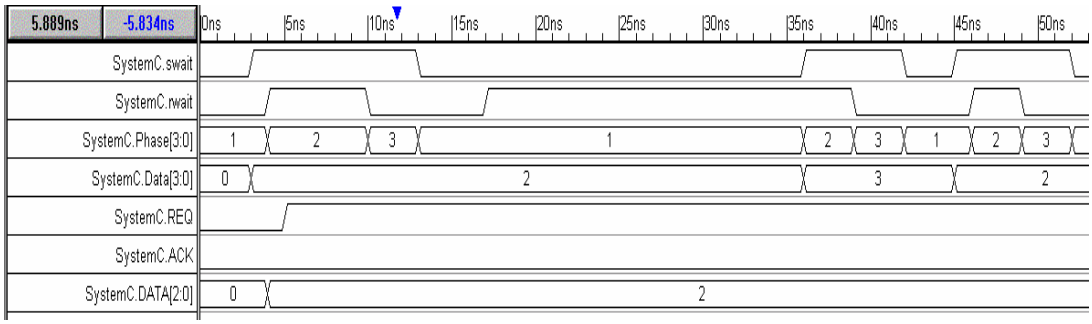
[3] SystemC Workgroup, http://www.systemc.org

[4] T. Grötker, S. Liao, G. Martin, and S. Swan, “System Design with SystemC,” Kluwer Academic Pub. 2002.

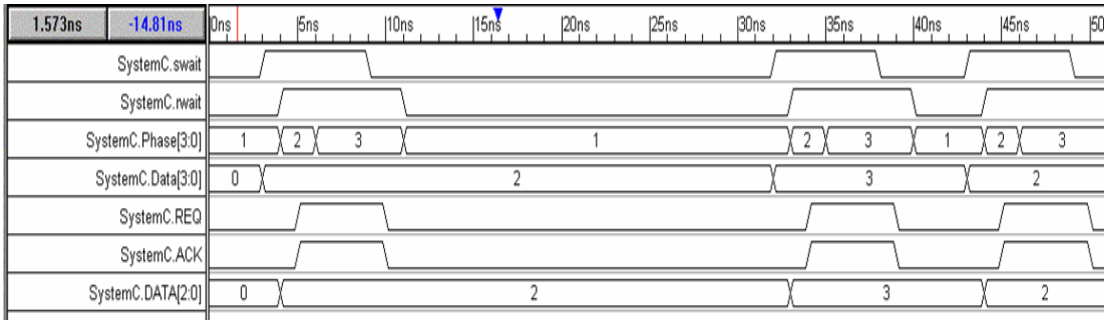
[5] S. Swan, “An Introduction to System Level Modeling in SystemC 2.0”, Cadence Design Systems Inc. 2001.

[6] “Version 2.0 User’s Guide”, SystemC.

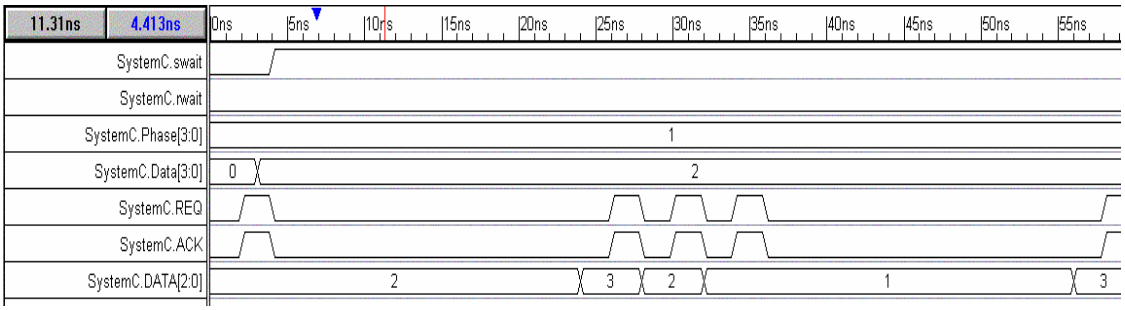
[7] “SystemC Golden Reference Guide”, Doulos 2002.



(a) Abstract to abstract



(b) Abstract to physical



(c) Physical to physical

Figure 6: 4-phase-bundled-data-push channel simulation waveforms