Functional Verification Methodology Based on Formal Interface Specification and Transactor Generation

Felice Balarin
Cadence Berkeley Laboratories
1995 University Ave Suite 460
Berkeley CA 94704
felice@cadence.com

Roberto Passerone
Dip.Informatica e Telecomunicazioni
Universit degli Studi di Trento
Via Sommarive 14 - Povo, Italia
roberto.passerone@unitn.it

Abstract

Transaction level models promise to be the basis of the verification environment for the whole design process. Realizing this promise requires connecting transaction level and RTL blocks through an object called a transactor*, which translates back and forth between RTL signal-based communication, and transaction level function-call based communication. Each transactor is associated with a pair of interfaces, one at RTL and one at transaction level. Typically, however, a pair of interfaces is associated to more than one transactor, each assuming a different role in the verification process. In this paper we propose a methodology in which both the interfaces* and *their relation are captured by a single formal specification. By using the specification, we show how the code for all the transactors associated with a pair of interfaces can be automatically generated.*

1. Introduction

Transaction level models have been traditionally used to clarify design requirements, and for early architectural performance estimations. Typically, these models could not interact with RTL blocks, so once the design of these blocks started, transaction level models would soon become obsolete. To improve verification productivity, there is a growing trend toward establishing transaction level models as the basis of the verification environment for the entire design process, which requires connecting transaction level and RTL blocks. However, they often communicate in a very different way, so they cannot be connected directly, but rather through an object called a *transactor*. The purpose of transactors is to translate back and forth between RTL, signal-based communication, and transaction-level, function-call based communication.

Each transactor is associated with a pair of interfaces, one at RTL and one at transaction level, but typically more than one transactor is associated with each pair of interfaces. Different transactors correspond to roles of different actors in a protocol, typical ones being *master*, *slave*, and *monitor*. The three transactors must deal with different actions at the higher level, such as sending data as opposed to receiving or monitoring the data. In addition, RTL signals which are inputs for one transactor may be outputs of another, and vice versa. For these reasons, the transactors required for characterizing a single interface are often described as three distinct entities, despite the fact that all three transactors implement the same protocol.

In this paper, we propose a methodology where both the interfaces and their relation are captured by a single formal specification. This specification is loosely based on the *Property Specification Language (PSL)* [16], and can be thought of as an extension of assertion based verification. We also describe a technique to automatically generate transactors from such a specification. The transactors are generated either in some flavor of C/C++, in Verilog, or in a combination of the two. We also describe the architecture of the transactor generation software. Because only a slim front-end depends on the input language, the tool can be easily extended to specification formalisms based on languages that are semantically similar to PSL, but syntactically quite different, such as *System Verilog Assertions (SVA)* [19]. Finally, we describe three case studies to show that automatically generated transactors can indeed replace hand-crafted ones in realistic designs.

1.1. Related work

There have been many proposals for formal interface specifications. In particular, two formalisms have emerged as a foundation of most of the approaches: regular expressions [8] and temporal logic [15]. The formalisms are similar in that they can both be expressed with finite-state automata. More recently, standard languages have been proposed to specify system properties. Two notable examples

3-9810801-0-6/DATE06 © 2006 EDAA

are PSL [16] and SVA [19]. They are both based on temporal logic logic, but both of them also include a capability to specify regular expressions. In PSL, such an extension is called *Sequential Extended Regular Expressions (SEREs)*. We have found this part of PSL, with some extensions, to be the most suitable for our purpose.

The properties specified in PSL or SVA can be used as a front-end to formal verification tools. In addition, many simulation environments are capable of generating simulation monitors from such properties, so that they can be verified by simulation, as well. Oliveira and Hu [12] have studied the suitability of regular expressions to specify complex interfaces for the purpose of generating simulation monitors. They have found that certain extensions to regular expressions to model pipelining and data can make interface specification significantly simpler and more compact. Our work is of similar nature, but our goals is complete interface specification and transactor generation.

As indicated by the discussion above, most of the previously published work is focused on generating simulation monitors, and almost nothing has been published on more general transactors (a monitor is a special case of a transactor). However, transactor generation has attracted some industrial interest, including TransactorWizard from Structured Design Verification [22], Bus Compiler from CoWare [10], and Cohesive from Spiratech [5].

Generating a simulation monitor in a C-like programming language from a regular expression is usually a two-step process. First, an equivalent finite-state automaton is generated for the regular expression. This is a fairly standard procedure, described in [8]. The second step is code generation for the finite state automaton, e.g. [3]. Generating a simulation monitor in HDL could in principle follow the same two-step procedure, but there are also direct ways to generate circuits from regular expressions [6, 17]. These technologies form a foundation of transactor generation as well, but they need to be adjusted in some details.

Our approach to transactor synthesis is based on the synthesis of converters between incompatible protocols. There are several techniques that can be used to achieve the correct result, including the use of signal transition graphs [4], matching of constructs in hardware description languages [11], and approaches based on finite automata [14, 1, 13]. We follow the latter approach, which fits well with our specification mechanism and that can be easily supported by formal analysis tools.

The rest of this paper is organized as follows. In Section 2 we introduce the formalism to specify interfaces and their relations. In Section 3 we present the architecture of the transactor generation tool. Code generation techniques are described in Section 4, and in Section 5 we present several case studies. Conclusions are given is Section 6.

2. Transactor specification

In this section we introduce our formalism by specifying the transactor protocol used in the *Utopia* interface [21]. Our presentation is simplified and somewhat incomplete, but it nevertheless captures the essence of the protocol, which is typical of many other similar protocols.

Utopia is a standard protocol used to connect devices implementing PHY and ATM layers [21]. Here we focus on the Receive part which covers a transfer of an ATM cell from a PHY to an ATM device. The latter is often referred to as Master, because it generates the clock that drives the transfer. For the same reason, the PHY device is referred to as Slave.

A possible design and its transaction level model are shown in Figure 1. At the transaction level, the PHY and ATM devices communicate through a simple mailbox. The transaction level PHY model calls the function `SendCell` to put a cell in the mailbox where it is picked up by the ATM device model, when it calls `GetCell`. At RTL, the PHY device asserts the `Clav` signal when it has a new cell available. After the ATM device confirms that it is ready to receive by asserting the `Enb` signal, the PHY device starts transmission by setting the `Data` signals and asserting the `Soc` signal to indicate the start of a cell. Thereafter, the PHY device may put fresh `Data` on each cycle that follows one in which `Enb` is asserted, or it may temporarily stop the transfer by de-asserting the `Clav` signal. The process continues until all 53 bytes of the cell are transferred.

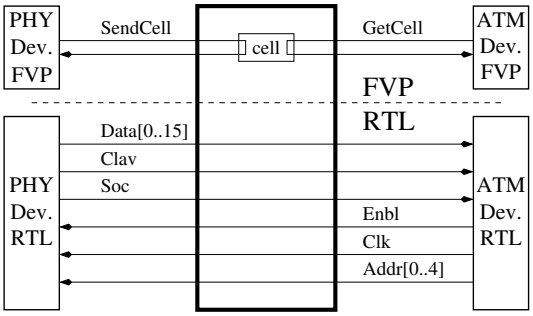

Figure 1. Utopia Receive interface

Transactors deal with the signals crossed by the thick line in Figure 1. Typically, there are at least three transactors associated with such a pair of interfaces, as shown in Figure 2. All three transactors essentially describe the same relationship over the same signals. The difference is that signals that are inputs to one transactor may be outputs to a different one. The relationship specifies which sequences of RTL signals correspond to which transaction-level calls. It is thus natural to call this relationship a protocol. It is clearly in-

efficient to repeat the specification of the same protocol for each transactor, so we explore an alternative, where all three transactors are generated from a single protocol specification.

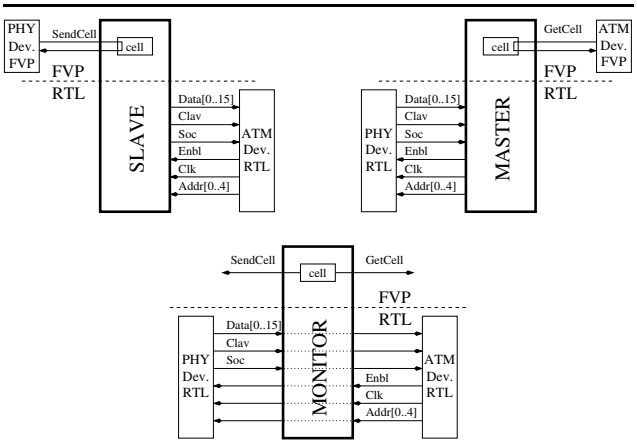

Figure 2. Master, Slave and Monitor transactors for Utopia Receive interface

A portion of a possible specification of Utopia Receive is shown in Figures 3 and 4. The specification is a conjunction of four "SEREs". The largest one, dealing mostly with RTL signals, is shown in Figure 3, and some of the others in Figure 4. We have put SEREs in quotation marks, because the code in Figures 3 and 4 contains extensions that we find necessary to completely specify interfaces, but which are not a part of the standard SERE formalism. In the following paragraphs, we motivate these extensions. The first extension is the addition of data, e.g. in Figure 3 we use an array of 53 bytes called `cell`. The second addition is loop counters, e.g. in Figure 3 we associate counter `i` (which ranges from 0 to 51) with the loop that repeats 52 times. We also use `i` to index the `cell` array. Both of these extensions (data and loop counters) make it very easy to add many extra state bits to the interfaces specification, which is likely to cause a state-space explosion in formal verification. However, for transactor generation, neither one represents a major problem, as the generated code represents them as compactly as the original specification.

```
// it starts with Clav asserted, waiting for Enb asserted
(!Soc && Clav && Enb)[*];
(!Soc && Clav && !Enb);
{ // transfer the first cell, Enb stays active ...
   ( Soc && Clav && Data==cell[0] && !Enb)
} | { // ... or Enb inactive for a while
   ...
};
{ // repeat 52 times, for cell[1] ... cell[52]
   // first, Clav can be inactive for a while
   {
      (!Soc && !Clav && Enb)[*];
      (!Soc && !Clav && !Enb)
   }[*];
   // then, transfer cell ...
   { // ... with Enb staying active ...
      (!Soc && Clav && Data==cell[i+1] && !Enb)
   } | { // ... or Enb inactive for a while
      ...
   }
}[*52;;i]
```

Figure 3. Receive interface RTL signals.

```
// Sendcell is followed by asserting Clav
(!Clav && B(SendCell, inCell) && inCell==cell);
(!Clav && N(SendCell))[*];
( Clav && N(SendCell));
(N(SendCell))[*]; (E(SendCell))

// GetCell is preceeded by a complete cell transfer
{ N(GetCell)[*]; B(GetCell); N(GetCell)[*] } &&
{ [*];
   { (Clav && !prev(Enb)); [*] }[*53];
};
(E(Getcell, outCell) && outCell==cell);

// SendCell and GetCell come in non-overlapping pairs,
// GetCell finishes before the coresponding SendCell
{
   (N(SendCell))[*]; (B(SendCell, inCell));
   (N(SendCell))[*]; (E(SendCell))
} && {
   (N(GetCell))[*]; (B(GetCell); (N(GetCell))[*]
};
E(Getcell, outCell)
```

Figure 4. Receive interface, other SEREs.

In addition to data and loop countersi, another class of extensions must be included to deal with function-call based transactions at the higher level. An example of this extension is shown in Figure 4. To each function call we associate a three-valued variable. The variable takes value `B` (for *begin*) when the function begins to execute, value `E` (for *end*) when the execution ends, and value `N` (for *neither*) at all other times. In addition, if a function has arguments, we associate them to the `B` value of the variable, e.g. `inCell` in `B(SendCell, inCell)`. Similarly, if the function returns some value, we associate it to the `E` value of the variable, e.g. `outCell` in `E(getcell, outCell)`.

3. Software architecture

Our transactor generation methodology is supported by a prototype synthesis software that accepts the declarative description of the RTL and transaction-level protocols, and generates appropriate code to support different protocol roles. As discussed in the introduction, only the front-end layer is language dependent. We support a simplified PSL parser that has been extended with the required language features discussed in the previous section.

The input specification is divided in three sections. The first section is used to declare the variables and the functions that are used or exported by the transactor(s). This section extends the PSL language, since PSL simply inherits the declarations from the host language. The middle section is the PSL-like description of the translation protocol, as described in Section 2. This section is independent of the particular role played by a transactor implementing the protocol. This is necessary, as we require that one protocol specification be used to construct several different transactors. The third and final section of the input is used to add transactor-specific information. Here, for each transactor implementing the protocol, specific directions are assigned to the variables and functions. More customization options can be made available in this section. For instance, a transactor may declare certain quantities to be constant, or implement only a subset of the available functionality. This could allow our synthesis technique to greatly simplify the transactors on a as-needed basis, thus reducing simulation time while maintaining compliance with the protocol.

The front-end builds an abstract syntax tree that represents the regular expression that describes the protocols. The abstract syntax tree is further translated into a finite state machine representation that we use as the basis to generate the executable code. Section 4 below provides details of how this is accomplished and discusses the complexity of the translation in terms of both time and space. In addition, our parser interprets the special transactor directives and creates the necessary transactor-specific information. This, together with the state machine representation of the protocol, is passed to the code generation back-end.

4. Code generation

The starting point of code generation is the finite state machine (FSM) derived from the protocol specification by the procedure described in [8]. The states of the machine correspond to nodes in the abstract syntax tree generated by the parser. Therefore, the number of states grows only linearly with the size of the input specification. The states are labeled with expressions in sum-of-product form, where each literal is either a *boolean literal*, i.e. a boolean variable or its negation, a *data literal*, i.e. an expression of the form $e_1 == e_2$, where e_1 and e_2 are expressions over data variables, or an *action literal*, i.e. an expression of the form $B(f)$, $E(f)$, or $N(f)$, where f is a transaction-level function, indicating that the execution of f is beginning, ending, or neither, respectively.

For a given transactor, actions are designated as *served* or *used*. The former designation indicates that the function is implemented by the transactor and called by other transaction level models, and the latter indicates that the function is called, but not implemented by the transactor. In addition, boolean and data variables are designated as input, output, or state variables. This leads to the following classification of data literals of the form $e_1 == e_2$. A data literal in which only input and state variables appear (but no output variables) is called an *input constraint*. A data literal in which either e_1 is just an output variable and e_2 contains no output variables, or vice versa, is called an *output assignment*. All other data literals are called *output constraints*.

To account for possible non-determinism in the specification, we maintain a set of possible current states for the FSM, rather than a single current state. This corresponds to performing the subsets construction [8] on-the-fly. However, in our case, the procedure is performed only partially for the specific input sequence, thus avoiding the associated exponential explosion.

No matter what the target language is, the generated code must implement the served functions, and execute the transactor correctly. Overall, the transactor execution engine must perform the following tasks in each execution step:

1. for the current inputs, find enabled transitions for all of the current states,
2. choose values of boolean and data output variables, served functions to return from, and used functions to call, consistent with at least some of the enabled transitions,
3. disable transitions inconsistent with choices in step 2,
4. update the set of current states by executing enabled transition, or report a failure if no transitions are enabled.

4.1. C++ code generation

In our approach to C++ code generation, the code for the transactor execution engine is not transactor specific, and thus it can be stored in a run-time library. The user can customize the module implementing step 2 to implement alternative non-determinism resolution approaches.

The code generated for each transactor contains only implementations of served functions, code constructing a data structure representing the FSM which is then traversed by the execution engine, and code to evaluate input constraints,

and execute assignments to output and state variables. The size of the generated code is proportional to the size of the FSM, which in turn is proportional to the size of the original specification. The run-time memory is also proportional to the size of the FSM, because the set of states are represented as binary bit-vectors whose size is equal to the number of states.

In general, a transactor needs to interface both to transaction level modules through served and used functions, and to RTL modules through ports. The former is natural for C++ code, but the latter can be done in different ways, supported by different verification environments. Our approach is to generate code based on a generic notion of a port, and to use run-time wrappers to specialize ports for the particular verification environment used. For example, several simulators support connecting SystemC [7] ports directly to RTL ports. So, we have developed a run-time wrapper which creates SystemC ports out of generic ports. We have also developed alternative run-time wrappers that can be used with the TestBuilder environment [20].

4.2. Verilog code generation

Our generated Verilog code implements the four steps of the transactor execution engine as a sequence of four combinational blocks. In other words, an FSM transition is executed in a single cycle. Step 2 of the algorithm is isolated in a separate module for possible customization by the user. As for C++ code, the size of the generated Verilog code is proportional to the size of the FSM.

As mentioned earlier, transactors need to offer both function calling interfaces to transaction level models, and port interfaces to RTL models. For Verilog code, the latter is natural, while there is no clear best way to do the former. Our generated code uses a simple handshaking protocol to indicate the beginning and the end of a function call. Alternatively, we could use Verilog tasks for the same purpose.

4.3. SCE-MI compliant transactor

Standard Co-emulation Modeling Interface (SCE-MI) is a protocol supporting *transaction-based acceleration (TBA)*, where transactors consist of a SW part and a HW part communicating through SCE-MI defined ports [18]. The HW part (responsible for most of the computation) can then transparently be simulated by a simulator, or emulated by an accelerator supporting SCE-MI. This clear advantage of SCE-MI is traded-off with the additional burden on the designer of creating SCE-MI compliant transactors. We have alleviated this burden by implementing a generator of SCE-MI compliant transactors from formal protocol specifications.

design	hand	spec.	C++	Verilog	SCE-MI C++	SCE-MI Verilog
ATM	1012	60	634	299	72	313
UART	55	53	450	181	126	169
SoC	64	94	564	206	164	248

Table 1. Transactor generation case studies.

The generated code consists of two parts. The SW part, written in C++, implements functions served by the transactor. The implementation of the served function does no processing, but it simply forwards function arguments to the HW part through SCE-MI defined ports, and then waits for the HW part to indicate that the function must return (possibly with a return value also communicated through SCE-MI defined ports). The HW part is written in Verilog and it is similar to Verilog-only code, except that it uses SCE-MI defined ports to communicate function arguments and return values to the SW part.

In this scheme, only the transaction level arguments and return values cross the HW-SW boundary, and all the detailed and usually much higher bandwidth RTL signal manipulation takes place on the HW side. This is important because the HW-SW bandwidth may limit the gains obtained by emulating the HW part.

5. Case studies

We have applied our transactor generation techniques to three case studies. In each of the three cases we started from an existing design coupled with a transaction level model of at least test-benches, if not the whole design. Every case included several hand written transactors based on standard protocols. In every case, we formally specified the protocol, and used our system to generate transactors. We then replaced hand written transactors with the ones we generated, and verified that the overall behavior did not change. Also, there was no observable change in simulation speed in any of the cases.

The first test case is an ATM switch design, which at its interfaces follows the Utopia protocol to transfer ATM cells. The second test case is a UART design that communicates to the outside world through the On-Chip Peripheral Bus (OPB) protocol [9]. Finally, the third test case is a complex SoC design, for which there exists a partial transaction level model in SystemC. This model communicates with the rest of the system (written in RTL Verilog) through the AMBA bus [2].

In Table 1 we show the number of lines of code of the hand written transactors (in column 1), formal protocol specifications (column 2), and the size of generated C++, Verilog and SCE-MI compliant code (columns 2-7, re-

spectively). The ATM switch and the UART design did not use general purpose transactors, but rather some code specific to this test-bench which acted as a transactor. Thus, in these cases the transactors were rather small, and very similar in sizes to formal protocol specification. It is important to notice that our protocol specification did not cover complete protocols, but only the features exercised by the test-benches. The size of the generated transactors is considerably larger than that of hand written ones, but still relatively small, and, as we explained earlier, growing only linearly with the size of the formal specification. Even in this case, our approach has the advantage that the formal specification can be shared between transactors for different protocol roles and in different languages. In the case of ATM switch, a full fledged UTOPIA transactor was used. It is much larger not only than the protocol specification, but also than the generated code. However, it also has much larger functionality than the transactor generated from our protocol specification which is limited to features exercised by this particular test-bench.

6. Conclusions

The value of transaction level models is fully realized only if they can be connected to RTL models. The development of transactors connecting the two levels is complex, costly and error-prone. We have proposed a methodology where interface protocols are specified formally only once in a way that is very similar to assertions used in verification. Transactors are then automatically generated from such a specification. Many transactors may be generated from a single interface specification, depending on which part of the design is being verified, what modes of the interface are being exercised, and which verification technology is being used (e.g. simulation vs. acceleration). In addition, formal interface specifications can be used as assertions and verified either statically or dynamically. We believe that formal interface specification and automatic transactor generation reduce development effort, foster transaction based acceleration, enable more reuse, and allow designers to explore more design options.

References

[1] J. Akella and K. McMillan. Synthesizing converters between finite state protocols. In *Proceedings of the International Conference on Conputer Design*, pages 410–413, Cambridge, MA, October 14 - 15 1991.

[2] AMBA Home Page. http://www.arm.com/products/-solutions/AMBAHomePage.html.

[3] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. M. Sentovich, and K. Suzuki. Synthesis of software programs for embedded control applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):834–49, June 1999.

[4] G. Borriello and R. H. Katz. Synthesis and optimization of interface transducer logic. In *Proceedings of the International Conference on Conputer Aided Design*, November 1987.

[5] Cohesive. http://www.spiratech.com.

[6] R. W. Floyd and J. D. Ullman. The compilation of regular expressions into integrated circuits. *J. ACM*, 29(3):603–622, 1982.

[7] T. Grotker, S. Liao, G. Martin, and S. Swan. *System design with SystemC*. Kluwer Academic Publishers, 2002.

[8] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, languages and Computation*. Addison Wesley, 1979.

[9] IBM. On-chip peripheral bus. available at http://www-306.ibm.com/chips/techlib/techlib.nsf/-productfamilies/CoreConnect_Bus_Architecture.

[10] T. Michiels. Generating TLM bus models from formal protocol specifications, Feb. 2004. presented at 9^{th} *European SystemC Users Group Meeting*, slides available at http://www-ti.informatik.uni-tuebingen.de/ systemc/ninth_escugm.html.

[11] S. Narayan and D. D. Gajski. Interfacing incompatible protocols using interface process generation. In *Proceedings of the* 32^{nd} *Design Automation Conference*, pages 468–473, San Francisco, CA, June 12 - 16 1995.

[12] M. T. Oliveira and A. J. Hu. High-level specification and automatic generation of IP interface monitors. In *Proceedings of the 39th ACM/IEEE Design Automation Conference*, pages 129–134, June 2002.

[13] R. Passerone, L. de Alfaro, T. A. Henzinger, and A. L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *Proceedings of ICCAD'02*, November 2002.

[14] R. Passerone, J. A. Rowson, and A. L. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *DAC*, San Francisco, CA, June 1998.

[15] A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, Oct. 1977.

[16] Property Specification Language: Reference Manual. available at http://www.accellera.org/pslv101.pdf.

[17] A. Seawright and F. Brewer. Clairvoyant: a synthesis system for production-based specification. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(2):172 – 185, June 1994.

[18] Standard Co-Emulation Modelling Interface (SCE-MI): Reference Manual (DRAFT), May 2003. available at http://www.eda.org/itc/scemi.pdf.

[19] System Verilog 3.1: Accellera's Extensions to Verilog. http://www.eda.org/sv/SystemVerilog_3.1_final.pdf.

[20] TestBuilder. http://www.testbuilder.net.

[21] The ATM Forum Techncal Committee. Utopia Level 2, Version 1.0, June 1995. available at http://www.atmforum.com/standards/approved.html.

[22] TransactorWizard. http://www.sdvinc.com.