Specification, Synthesis, and Simulation of Transactor Processes

Felice Balarin, *Member, IEEE*, and Roberto Passerone, *Member, IEEE*

***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 register-transfer-level (RTL) blocks through 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 with 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. Our synthesis algorithm avoids the state-explosion problems associated with certain features of the specification formalism, at the expense of a more sophisticated simulation algorithm. We describe three different code-generation techniques targeted at different verification languages: 1) C++; 2) Verilog; and 3) the combination of the two that is compliant with the Standard Co-Emulation Modeling Interface protocol. In addition, we present several case studies demonstrating that automatically generated transactors can indeed replace handcrafted ones in realistic designs.**

***Index Terms*—Code generation, finite-state machine (FSM) simulation, property specification language (PSL), standard co-emulation modeling interface (SCE-MI), state explosion, SystemC, transaction-level models (TLMs), transactor, verification, Verilog.**

I. Introduction

TRANSACTION-LEVEL MODELS (TLMs) are emerging as a key ingredient of system-level verification methodologies [1], [2]. They provide an accurate executable system model, which is still at a sufficiently high level of abstraction to enable full-chip simulation and early software development. Because they can be simulated orders of magnitude faster than register-transfer-level (RTL) models, TLMs can be used as an efficient executable specification and also as the golden reference for checking functional correctness and measuring transaction coverage. In addition, they can also serve as a verification environment for block-level verification and as the basis for architectural performance analysis.

Despite their advantages, TLMs have been traditionally only used for design requirement clarification and for early architectural performance estimations. The reason has to do with the difficulty of making TLMs interact directly with RTL blocks, due to the differences in their communication paradigm. Thus, once the design of RTL blocks started, verification engineers would put most of their efforts in developing block-level verification environments, and the TLMs would soon become obsolete. To amortize the cost of their development, and to improve verification productivity, there has recently been a growing trend toward establishing TLMs as the basis of the verification environment *for the entire design process* [3]. This is achieved by introducing an additional object between transaction-level and RTL blocks, called a *transactor*, which translates back and forth between RTL, signal-based communication, and transaction-level function-call-based communication.

We refer to a set of signals and to the protocol they use as an *interface*. Each transactor in a design is associated with a pair of interfaces, one at the RTL and one at the transaction level. However, each pair of interfaces may be associated at various times with *more than one* transactor, each taking a different role as the master, the slave, or the monitor of the transactions, depending on the kind of verification that needs to be performed (Fig. 3 is an example). Each of these transactors deals with different actions at the higher level (one sends the data, whereas the others receive the data) and have complementary views of the signals at the RTL (the inputs for one transactor are regarded as output by the other). In addition, most tests do not use the full capabilities of an interface. Thus, when designed manually, it is often more efficient to create specialized transactors with just the right capabilities, rather than to configure a complete, and therefore also complex and inefficient, transactor. For these reasons, the three transactors that are required for characterizing a single pair of interfaces are often described as three distinct entities, with limited code sharing, despite the fact that all three transactors implement the same basic protocol.

The matter is made worse by considering that transactors are written in a combination of different languages since they bridge the gap between TLMs, which are often written in C or C++, and RTL models, which are written in a hardware description language (HDL). In some cases, it may be efficient to use standard C/C++ interfaces to HDLs and to specify most of the transactors in C/C++. For example, several HDL simulators are now capable of simulating *SystemC* [1] models as well. In other cases, transactors are specified in an HDL. This may be useful, for instance, when the transactor is to be partially implemented on a hardware accelerator. This produces a proliferation of different implementations of the same transactor and of transactors corresponding to the same pair of interfaces. The consequence is an obvious increase in development cost and,

Manuscript received August 17, 2006; revised November 26, 2006. This paper was recommended by Associate Editor R. F. Damiano.

F. Balarin is with the Cadence Berkeley Laboratories, Cadence Design Systems, Inc., Berkeley, CA 94704 USA (e-mail: felice@cadence.com).

R. Passerone is with the Department of Information and Communication Technology, University of Trento, 38100 Trento, Italy (e-mail: roberto.passerone@unitn.it).

Digital Object Identifier 10.1109/TCAD.2007.895792

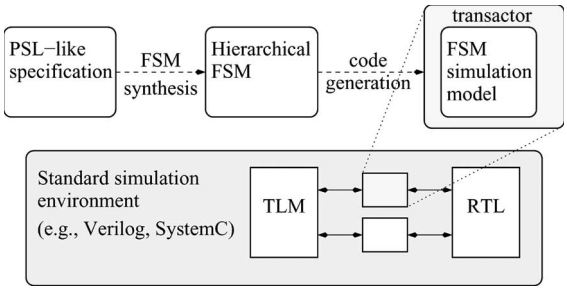

Fig. 1. Typical use case.

more importantly, the inability to maintain consistency among the implementations of the same protocol across all its different incarnations.

In this paper, we propose a methodology where both the interface protocols and their relation are captured by a single formal specification. Then, we show how all transactors associated with a pair of interfaces can be automatically generated from this specification. The transactors are generated either in some flavor of C/C++, in Verilog, or in the Standard Co-Emulation Modeling Interface (SCE-MI)-compliant combination of the two [4]. Each generator is tailored to a different verification environment. The choice of code generator is, however, independent of the particular protocol role of the transactor. Thus, by combining the two degrees of freedom (language and role), one can generate a full matrix of different transactors from a single protocol specification.

The specification formalism that we use is loosely based on the Property Specification Language (PSL) [5] and can be thought of as an extension of assertion-based verification. In its simplest form, generating transactors from such specifications is a two-step process. First, the specification is transformed into a finite-state machine (FSM), and then, the code simulating the FSM is generated. The generated code can then be used in any standard simulation environment, as shown in Fig. 1. Unfortunately, some PSL operators, and certain extensions that we introduce to manipulate data, may cause an exponential explosion in the number of states that are generated in this phase, making the translation impractical. Our solution avoids this problem by converting the specification into a nondeterministic hierarchical FSM, where states are derived from the PSL operators and labels (guards and actions) are derived from data operations. Our main contribution in this paper is an algorithm that can simulate the FSM (a simulation model) that can be used in several different simulation environments. We present a set of three case studies to show that automatically generated transactors can indeed replace handcrafted ones in realistic designs. In addition, we have employed a generic FSM model so that the simulation algorithm can be applied to complex finite-state systems that are not necessarily derived from our transactor specification.

The rest of this paper is organized as follows: Section II presents some related work. We then present our specification formalism in Section III and the overall architecture of our software tool in Section IV. Section V gives a detailed description of the generation and representation of the finite-state structure. The main simulation algorithm is presented in Section VI, whereas the specific code-generation strategy is presented in Section VII. Finally, Section VIII presents the results of our technique applied to three industrial case studies, followed in Section IX by an in-depth discussion of our design choices in the context of the most relevant related work.

II. Related Work

The problem of transactor specification and synthesis can be seen as a special case of automatic generation of adapters between incompatible protocols. There are several techniques that can be used to achieve the correct result, including the use of signal transition graphs [6], matching of constructs in HDLs [7], and approaches based on finite automata [8]–[10]. We follow the latter approach, which fits well with our specification mechanism and that can be easily supported by formal analysis tools.

There have been many proposals for formal interface specifications. The purpose of many early proposals was to specify interface properties that can then be formally verified. Therefore, the focus was on formalisms that can express complex relationships between control signals (which is a sweet spot for formal verification), whereas data were generally abstracted (because data can easily overwhelm formal verification tools). In particular, two formalisms have emerged as a foundation of most of the approaches: 1) regular expressions [11] and 2) temporal logic [12]. The formalisms are similar in that they can both be expressed with finite-state automata. The difference between the two is that for regular expressions, it suffices to consider only finite traces accepted by finite-state automata, whereas temporal logic requires also infinite traces. Infinite traces are convenient to talk about eventualities. For example, to refute the property "Every *request* is eventually *granted*," one needs to show an infinite trace in which there is a *request* that is never *granted*. While eventualities are clearly useful to specify individual properties that a system should satisfy, they are less useful for complete system specifications. Such a specification needs to precisely describe a finite sequence (or possibly sequences) of events that leads from a state where *request* is made to a state where it is *granted*.

More recently, standard languages that are based on temporal logic have been proposed to specify system properties. Two notable examples are PSL [5] and SystemVerilog Assertions (SVA) [13]. While regular expressions capture the full power of finite-state automata on finite traces [11], the common temporal logics can express a strict subset of finite-state automata on infinite traces [14]. Therefore, both PSL and SVA extend the temporal logic to include the power of regular expressions. In PSL, such an extension is called Sequential Extended Regular Expressions (SEREs). Since we are interested in complete interface specifications, rather than abstract properties, we base our formalism on SEREs and examine aspects in which they are, or they are not, suitable for our purposes.

The use of regular expressions for hardware description in general and for protocol description in particular is not new and has been already presented in the literature. Seawright and Brewer demonstrate how effective regular expression could be for protocol and control intensive specifications [15]. Oberg *et al.* use a similar grammar-based specification for the synthesis of hardware for data communication protocols,

although the problems of software and transactor generation are not addressed [16]. Shimizu *et al.* propose a monitor-based specification style for the verification of communication protocols that decomposes the specification into several automata, each specifying a particular property that the protocol should exhibit [17]. Their approach is, however, limited to the generation of monitors. Recently, Siegmund and Müller have proposed a similar approach where the regular expressions are embedded in the description language, in this case SystemC, using appropriate supporting classes [18]. The advantage is that the transactor description can be simulated directly with the existing application. However, in this approach, the user is required to describe each transactor individually, instead of having it be generated automatically from a description of the communicating protocols and their relation.

The properties specified in PSL or SVA can be used as a front end to formal verification tools. In addition, commercial simulation environments are capable of generating simulation monitors from such properties, so that they can be verified by simulation as well. Oliveira and Hu [19] 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 are complete interface specification and transactor generation.

As indicated by the aforementioned discussion, most of the previously published work is focused on generating simulation monitors, and little has been published on the synthesis of more general transactors. However, transactor generation has attracted some industrial interest, including TransactorWizard from Structured Design Verification [20], Bus Compiler from CoWare [21], and Cohesive from Spiratech [22]. Unfortunately, the proprietary nature of these tools prevents us from making a direct comparison with these tools.

Our approach to generating transactors is a two-step process, where a finite-state automaton equivalent to the regular expression is generated, followed by the synthesis of the simulation code. Converting a regular expression into a finite automaton is a fairly standard procedure [11]. Code generation has also been investigated, both for software implementation [23], [24] and for direct mapping to circuits [15], [25]. We will discuss some of these techniques in detail in Section IX.

PSL has been translated to finite automata not only for formal verification but also for simulation [26]. Still, our problem has several distinguishing features. First, as we will see in Section III, our approach extends PSL with data variables. Dealing with data variables may appear easy, but it actually gets quite cumbersome when nondeterminism and subsequence instantiations cause several versions of the same variable to coexist simultaneously. Second, existing simulation tools use finite automata generated from PSL as monitors, which are a special case of transactors that have only inputs and no outputs. The advantage of monitors is that parallel sequences can be monitored independently. This is no longer true for a transactor with outputs since all parallel sequences need to coordinate the choice of outputs. At the same time, keeping parallel sequences separate is essential to avoid an exponential increase in the number of states. Finally, PSL has been traditionally used for specifying simple properties. For these, a naive approach may still generate automata that are small enough to be handled. We aim at fully specifying complex protocols using a PSL-like specification and the naive approach is no longer feasible.

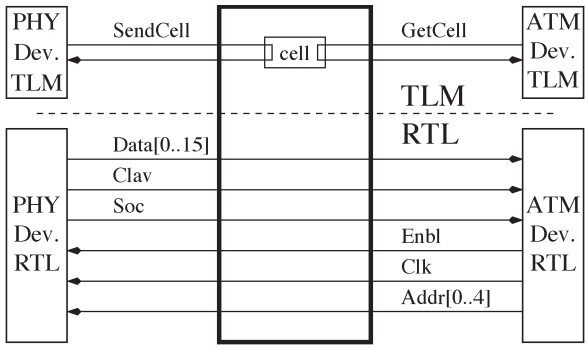

Fig. 2. Utopia Receive interface.

Another formalism that exhibits similar features is Esterel [27]. Esterel compilers deal with parallelism by compiling Esterel in either concurrent hardware or parallel software threads [28]. In these approaches, the size of generated code is small, although the size of state space that it represents may be very large. Unfortunately, it is hard to extend this approach to a formalism with nondeterminism, where multiple alternative states need to be maintained during simulation.

Statecharts [29] is also a specification mechanism that has all the problematic features of our formalism and that targets the modeling of complex systems. Statecharts is supported by simulation tools, but unfortunately, little has been published about simulation algorithms in these tools beyond the fact that "*calculating the effect of a step contains involved algorithmic procedures*" [30].

III. Transactor Specification

Our formalism is based on the SERE subset of PSL [5]. SEREs are regular expressions with multiple variants of the *sequencing*, *repetition*, OR, and AND operators. We also introduce certain extensions that we find necessary to more easily and completely specify transactors. We will discuss the extensions, and their implications, later in this section. We present our formalism by way of the transactor protocol used in the Utopia interface [31]. Our specification of the protocol is simplified and somewhat incomplete, but it nevertheless captures its essence, which is typical of many other similar protocols.

Utopia is a standard protocol that is used to connect devices implementing physical (PHY) and asynchronous transfer mode (ATM) layers [31]. 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 TLM are shown in Fig. 2. 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

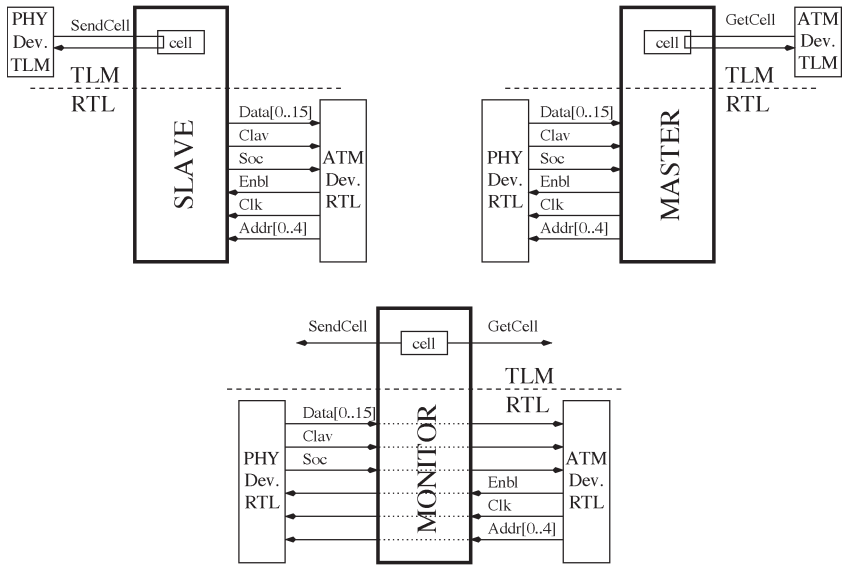

Fig. 3. Master, slave, and monitor transactors for the Utopia Receive Interface.

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 transmitting 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 deasserting the `Clav` signal. The process continues until all 53 bytes of the cell are transferred (the `Addr` signals are relevant only with multiple PHY devices, which is a case that we will not consider here).

Transactors deal with the signals that cross the thick line in Fig. 2. The three roles are shown in Fig. 3. The Master transactor is used to verify the RTL model of a PHY device, the Slave transactor is used to verify the RTL model of an ATM device, and the Monitor transactor is used with the RTL models of both PHY and ATM devices.

A portion of the specification of Utopia Receive is shown in Figs. 4 and 5. The specification is a conjunction of several SEREs. The largest one, dealing mostly with RTL signals, is shown in Fig. 4, and some of the others are shown in Fig. 5. The code in Figs. 4 and 5 shows our extensions, which are not part of the standard SERE formalism. The first extension is the addition of data variables. For instance, we use an array of 53 bytes called `cell` to represent the ATM cell. The addition of data variables is essential in protocol specification since the values that are presented at one end of the interface must match those that are received at the other end. In addition, the functionality may sometimes depend on the values that are exchanged by the communicating parties. However, `cell` is not part of the design itself but is rather an auxiliary object that is used to specify the interface. The second extension is the addition of loop counters such as the variable `i` (which ranges from 0 to 51) associated with the loop that repeats 52 times. This is particularly convenient as an index to arrays, such as `cell[i + 1]`, and therefore accesses a different value at each iteration. Both of these extensions (data variables and loop

```
// It starts with Clav asserted...
( !Soc && Clav && Enb )[*];
...and waiting for Enb asserted
( !Soc && Clav && !Enb );
{
  // transfer the first cell, Enb stays active ...
  ( Soc && Clav && Data == cell[0] && !Enb )
} | {
  // or, transfer the first cell, with Enb inactive for a while
  ( Soc && Clav && Data == cell[0] && Enb );
  ( !Soc && Enb )[*];
  ( !Soc && !Enb )
}; {
  // 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
    // (body omitted here)
  }
}[*52;;i] // repeat 52 times with index i
```

Fig. 4. Utopia Receive interface: RTL signals.

counters) add extra states to the interface specification and must be handled with care. Section V discusses our approach to avoid potential state-explosion problems.

Adding data and loop counters to SERE is necessary for a complete and compact interface specification at the RTL. 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 Fig. 5. With each function call f, we associate three different Boolean expressions, called *action expressions*, which evaluate to true at different times during the protocol execution. The term `B(f)`, for *begin f*, is true at

```
// 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) && outCell == cell);
```

Fig. 5. Utopia Receive interface: other SEREs.

all cycles in which the function is invoked, whereas `E(f)`, for *end f*, is true whenever the function returns. The term `N(f)`, for *neither f*, is true at all other times. Function arguments are made available at function invocation and are syntactically attached to the `B` term, as for `inCell` in `B(SendCell, inCell)`. Similarly, the return value (if any) is associated with the `E` term, as for `outCell` in `E(getcell, outCell)`.

Function arguments and return values can be related to data objects. This is shown in Fig. 5 by the data literals `inCell == cell` and `outCell == cell`. In this way, we can separate the specification of the object being transferred over the interface (`cell`) from its source or destination (`inCell` or `outCell`). This is important because `cell` is an object that appears in all the transactors, whereas, for example, `SendCell` and `inCell` do not appear at all in the Master transactor. Notice also that a property like `inCell == cell` must be satisfied by the generated transactors, but there is considerable freedom in how to achieve that. For example, one can use pointer manipulation to ensure that `inCell` and `cell` refer to the same memory region, or one could copy the whole array. It is up to code generator to resolve these choices in an optimal way.

In addition to our extensions, we take advantage of other features of PSL that facilitate a compact description of the specification, such as the instantiation of subsequences. Sequence definitions may be parameterized with arguments that need to be specified when a sequence is instantiated. An example of sequence instantiation, and its implications on code generation, is discussed in Section V. We have found that our extensions, together with the facilities provided by PSL, are sufficient to express complex behaviors such as overlapped (or pipelined) transactions and burst transfers. While regular expressions are equivalent to state machines, and are therefore complete for finite-state models, other extensions might be necessary to easily express particularly complex cases. These additional extensions would have to be handled using techniques that are similar to those presented in this paper.

IV. 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. The overall software architecture is shown in Fig. 6.

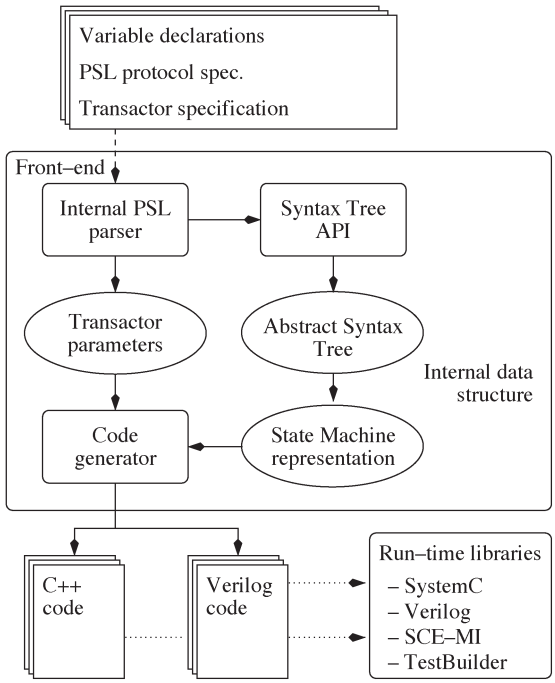

Fig. 6. Software architecture.

The input specification is divided into three sections. The first section, i.e., *variable declaration*, is used to declare the variables and functions that are used or exported by the transactor(s). This section extends the PSL language, which does not need declarations since these are inherited from the host language. The middle section is the PSL-like description of the translation protocol, as described in Section III. This section is independent of the particular role that is played by a transactor implementing the protocol. This is necessary, as we require that one protocol specification be used to construct several different transactors. In particular, the protocol specification is nondirectional, in the sense that data and control may flow in either direction. This is accomplished by declaring the legal sequences of data and control actions without specifying the initiator or the recipient of the action. The third section of the input, i.e., *transactor specification*, is used to add transactor-specific information. Here, for each transactor implementing the protocol, specific directions are assigned to the variables and functions. For regular data signals, the direction is classified as input or output. Transaction-level function calls (or actions), instead, are designated as *served* or *used*. Served actions are functions that are *implemented* by the transactor and called by other TLMs. Used actions are functions that are *called* but not implemented by the transactor. This information is used during synthesis to determine how to translate from the declarative specification to executable code, as described in Section VII. More customization options can be declared 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 an as-needed basis, thus

```
sequence transfer(num_bytes, start_addr) {
  {
    (addr = start_addr); <addr_cntrl>
  } && {
    < burst(num_bytes)> | < burst(num_bytes/2)>[*2]
  }
};
sequence addr_cntrl( ) {/* body omitted here */};
sequence burst( nb) {/* body omitted here */};
```

Fig. 7. Simple protocol specification.

reducing simulation time while maintaining compliance with the protocol.

The front end takes the input specification and builds an abstract syntax tree that represents the regular expression that describes the protocols. We support a simplified PSL parser (i.e., we only recognize the part of PSL that is related to SEREs) that has been extended with the required language features that were discussed in Section III. This has been developed primarily to have a quick way of implementing and evaluating different extensions to the language, without the burden of a large infrastructure. However, the set of functions that are used to construct the syntax tree (create and connect the nodes) is exposed to the user as an application programming interface so that other languages, such as SVA, could be easily integrated.

The nodes of the abstract syntax tree correspond to the sequential operators, whereas the leaves of the tree are the Boolean expressions, which are represented in sum-of-product form. The abstract syntax tree is further translated into an FSM representation that we use as the basis to generate the executable code. Section VII provides details of how this is accomplished. In addition, our parser interprets the transactor specification and creates the necessary transactor-specific information as a set of *transactor parameters* that include the direction of the signals and the constants. This, together with the state machine representation of the protocol, is passed to the code-generation back end, which can generate either C++ code or Verilog, as described in Section VII. The code that is common among all transactors, such as the simulation code that traverses the state machine, is collected in a set of runtime libraries that are used with the generated code to construct a complete executable transactor. The combination of the generated code and the runtime library is then instantiated within the preferred simulation environment, such as a Verilog and/or a SystemC simulator, to bridge the communication between the transaction-level and RTL codes, as shown in Fig. 1.

V. FSM Generation and State Representation

The first step of the synthesis process consists of translating the extended PSL specification of the transactor into an equivalent finite-state representation. We will use the example in Fig. 7, which is a simple protocol to transfer `num_bytes` bytes starting from address `start_addr`, to illustrate the procedure. The example also illustrates the use of sequence instantiation using the operator $\langle \ldots \rangle$. The transfer consists of two parallel subsequences. In the first, the `addr` is initialized with the starting address, followed by the execution of sequence `addr_cntrl`, which (presumably) defines how `addr` is updated. Concurrently, a data transfer is started, either as a single burst of `num_bytes` bytes or as two shorter bursts of `num_bytes`$/2$ bytes.

A. FSM Generation

To avoid constructing an overly large data structure, the size of the finite-state representation should not grow more than polynomially (ideally linearly) with the size of the input description. Unfortunately, there are several situations that could lead to a potential exponential explosion of the number of states. One is nondeterminism, which is shown in Fig. 7 by the choice of two different kinds of burst transfers. One approach to handling nondeterminism is to replace a nondeterministic FSM with an equivalent but deterministic one. Unfortunately, the deterministic equivalent may have exponentially many more states [11]. Our solution takes advantage of runtime determinization, where the simulator keeps track not only of the current state but also of *the set of all possible* current states consistent with the inputs [23]. The set of current states may still grow exponentially large. In practice, however, since we traverse only the portion of FSM that is excited by the input sequence, we have not observed this phenomenon when we applied the algorithm to specifications of realistic protocols. Section IX discusses our choice in more detail, in the context of other relevant related work.

Another source of potential state explosion is the extension to data variables, such as `addr` in the example. The explicit state representation of an n-bit variable requires 2^n states, which is infeasible. The obvious solution is simply to record the variable as an n-bit value. Symbolic state-space search (such as reachability) remains hard in this case. However, for simulation, we only record the current value of the variable along a specific input sequence. Nondeterminism may force us to consider several values for the variable, one for each concurrent nondeterministic state. As discussed, however, realistic protocols are, in practice, well behaved.

The process of deriving FSMs from PSL SEREs is syntax driven, where the FSM for each expression is constructed starting from the FSMs of each of its subexpression. Each FSM is defined by its *initial state*, its *final state*, and a *transition function* that assigns to each state a set of next states. For a given state s, we use the function $next(s)$ to denote this set. The construction for a representative of every class of operators, which is similar to the one proposed in [11] and [23], is shown in Fig. 8. The construction for other operators is also similar. In Fig. 8, s and q are SEREs, b is a Boolean expression, n is an integer expression, c is an integer variable, and SEQ stands for a name of a previously defined SERE with k parameters. The solid lines in the graph represent the next states.

The top two rows of Fig. 8 are essentially the same constructions that are presented in [11] and [23]. The states of the machine correspond to the nodes in the abstract syntax tree that are generated by the parser. In addition, the leaves of the abstract syntax tree, i.e., the Boolean expressions in the PSL specification, are used to label certain states. These states are called *labeled states* and are represented in the graph as solid

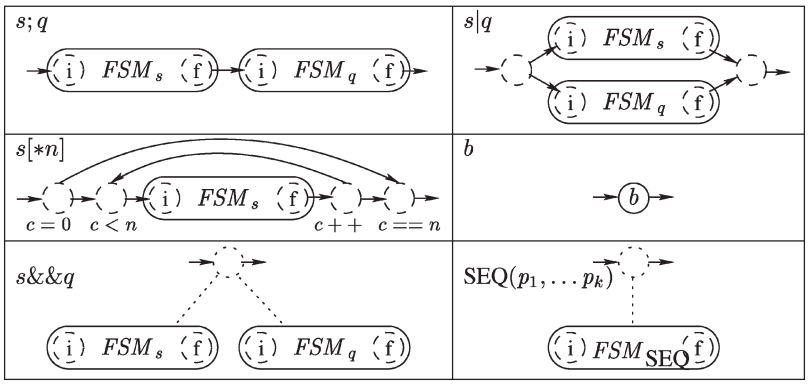

Fig. 8. Deriving FSMs from PSL SEREs.

circles. For example, the expression (`addr` = `start_addr`) in Fig. 7 generates a state with that same label.

Formally, labels are Boolean expressions over the set of input, output, and state variables. The atomic expressions include Boolean signals, data expressions of the form $e_1 == e_2$, where e_1 and e_2 are terms over the data variables, and action expressions of the form $B(f)$, $E(f)$, or $N(f)$, where f is a transaction-level function. Intuitively, the label describes: 1) conditions that must be satisfied for the system to transition into a state; 2) outputs that need to be generated when transitioning into a state; and 3) updates to state variables that need to be performed when transitioning into a state. In addition, expressions distinguish between *present-state* variables, which have already been assigned, and *next-state* variables, whose value must be computed. In each simulation step, the values of input and present-state variables are given. These values are then substituted into the label of the potential next states by partially evaluating the expression. If the label of a state becomes unsatisfiable, then the state cannot be entered under the given input condition. Otherwise, there exists an assignment to output and next-state variables that satisfies the label, and the state can be entered. Finding such an assignment is a crucial part of each simulation step.

The constructions for the sequencing and the choice operators are straightforward, and they simply require the generation and connection of the appropriate initial and final states, which are represented in the graph by dashed circles. The initial states of sub-FSMs are marked by "i," and the final states are marked by "f." These states are called *transient states* because they could, in principle, be eliminated and can never be observed as system states. They are retained in the representation because they simplify the simulation. During execution, transitions originating from transient states are traversed immediately upon reaching the transient state, without waiting for the next cycle. Thus, they behave like the ϵ transitions in [11] and [23].

In addition to the initial and final states, there may be transient states to which we associate variable updates or conditions that must be satisfied for the system to enter into that state. These are shown in the graph under the circle representing the state. One example is the translation of the repetition operator $[*n]$, which denotes repetition for n times. The straightforward implementation of the PSL semantics of $[*n]$ would be to unroll the loop n times. If repetitions are nested, this would clearly incur an exponential blowup. By introducing a *counter variable* c, the FSM can be represented as compactly as the original PSL expression. The exit condition and the update to the counter variables are executed in appropriate transient states. However, this complicates the task of simulating the system because: 1) variables need to be considered as part of the global state and 2) there could be variable updates and entrance conditions associated with transient states. The simulation algorithm that we propose addresses these two problems in general: 1) It is capable of handling state variables (including, but not limited, to counter variables); and 2) it is capable of handling variable updates and entrance conditions in transient states (including, but not limited to, those arising from the operator $[*n]$).

The FSM for the subexpressions in the top two rows of Fig. 8 are "in-lined"; that is, they will become part of a larger FSM. Effectively, this flattens the hierarchy by eliminating the node for the operator from the graph and by replacing it with its operands. Also, the subsequence and the parallel operator && could be eliminated this way. However, both of these operations may incur an exponential blowup. For example, in sequence `transfer` of Fig. 7, subsequence `burst` is instantiated twice, with different parameters. In-lining requires that we build *two copies* of the same FSM for `burst` and place them in the appropriate location within the FSM for `transfer`. However, if `burst`, in turn, instantiates some other sequence twice, and so on, up to some depth level n, then this approach would result in a state space that grows exponentially with n. The same problem happens with composition. In Fig. 7, there are two parallel subexpressions: one dealing with the address and the other with data. To in-line the composition, we need to create one FSM for each subexpression and then form their *product* [11]. Unfortunately, if there are n parallel subexpressions, then the state space of the product is exponential in n.

To avoid this problem, we retain the operator in the FSM as a *compound state*, which is shown as a dotted circle in the figure, and build a hierarchical FSM by recursively generating the FSM for the operands. For the subsequence operator, a *subsequence state* is generated and is associated (dotted lines in the figure) with the FSM that represents the instantiated sequence. The subsequence is represented only once. In other words, if two subsequence operators instantiate the *same* subsequence, then they will be associated with the *same* FSM, thereby avoiding static duplication. In the case of the parallel

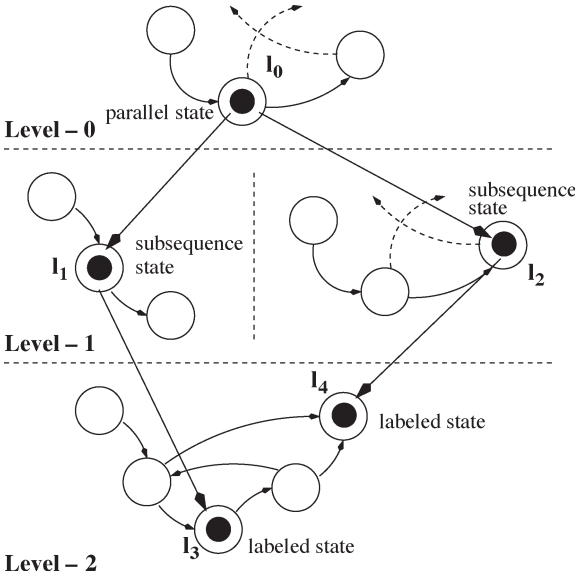

Fig. 9. Example of a hierarchical state.

operator, we introduce a *parallel state* that has references to the FSMs for the two subexpressions. Because we represent the two operands separately, we do not compute the product and avoid the potential complexity.

It is not hard to see that the number of FSMs generated by our procedure and the size of the generated FSMs both grow at most linearly with the size of the initial specification. Therefore, we have avoided the exponential explosion in FSM generation.

B. Global State Representation

The result of the FSM generation process is a nondeterministic hierarchical state machine. Because the representation is generic, we will refer to the state machine for the transactor simply as a *system*. Starting from the initial state, for a given level of the hierarchy, the system may transition to a labeled state (or a transient state) or to a compound state. Whenever a compound state is reached, the state machine at the lower level is also activated by entering its initial state. The compound state is abandoned as soon as the lower level FSM reaches its final state. Therefore, at any point in time, the state machine is at some labeled or transient state at a certain level of the hierarchy and at a compound state at each of the higher levels of the hierarchy. The combination of the labeled/transient state and the higher level compound states forms the *global state* of the FSM. By contrast, we will refer to the states of the individual FSMs in the hierarchy as *local states*. Because the FSM is hierarchical, the global state can be represented by a tree, where the internal nodes are the compound local states and the leaves are the labeled or transient local states, as shown in Fig. 9. Equivalently, we can represent the global state g as an expression. We define a hierarchical state h as an expression in the grammar

$$h := (l) \,|(s, h_1)|\, (p, h_1, h_2)$$

where l is a labeled or transient state, s is a subsequence compound state, and p is a parallel compound state. Hierarchical states corresponding to subsequence and parallel states contain references to the hierarchical states h_1 and h_2 (called *substates*) rooted at the associated subsequences. A global state g is then a hierarchical state rooted at the top-level FSM and is therefore a tree that spans the entire system hierarchy. The example in Fig. 9 is represented as

$$(l_0, (l_1, (l_3)), (l_2, (l_4))).$$

In addition to the explicit state information, the state comprises the evaluation of the variables that are present in the specification. Variables are classified as *input*, *output*, or *state* variables. Input and output variables are used to communicate with the environment, and they include both the simple signals carrying the data and the extra variables used to denote the value of action expressions (such as `B(func)`) corresponding to the activation of transaction-level function calls. State variables record internal information, and they include the data and the counter variables of the extended PSL specification. With each local state, we therefore associate a data *interpretation*, i.e., a function I that assigns values to the variables in scope, or visible, at that state. The values assigned by the interpretation are computed during simulation according to the given input sequence and the transitions of the hierarchical FSM. A local state l (labeled, transient, or compound) with interpretation I is denoted as the pair $[l, I]$. The example in Fig. 9, together with interpretations, will then be denoted with the expression

$$([l_0, I_0], ([l_1, I_1], ([l_3, I_3])), ([l_2, I_2], ([l_4, I_4]))).$$

This expression can efficiently be stored in the program by recording the values of the variables that are given by the interpretation functions and by maintaining the appropriate references to the local states in each of the FSM that make up the system.

Global variables have global scope, whereas other variables have limited scope. For example, the counter variable c of Fig. 8 goes in scope at the state labeled $c = 0$ and goes out of scope at the state labeled $c == n$. The scope is well defined since there are no transitions going in or out of the portion of the FSM enclosed between the initial and final states. The scope of a variable also extends to all the substates in the lower hierarchy levels that are reachable while the variable is in scope at a certain level. Thus, when two distinct states refer to a state variable x, which is already in scope at a common state at a higher level of the hierarchy, then they actually refer to the *same* variable. In this case, we say that the two states *share* variable x. Otherwise, the two states refer to two *distinct* variables that simply happen to have the same name.

For a labeled local state l, we say that the pair $[l, I]$ is *consistent* whenever the label of l is satisfied by the values of the variables that are assigned by I. During simulation, we always construct pairs of states and interpretations that are consistent. We say that a hierarchical state h is *compatible* whenever all labeled local states in h are consistent with the associated interpretations, and all interpretations agree on the values of the variables that are shared by their associated states (including all global variables). Thus, a noncompatible hierarchical state either has interpretations that are inconsistent with the labels of their local states or has at least two interpretations that

require that different values be assigned to the same variable. For example, consider a state $h = ([l_0, I_0], h_1, h_2)$ (a parallel composition), where $h_1 = [l_1, I_1]$ and $h_2 = [l_2, I_2]$. Assume that states l_1 and l_2 are labeled with the expressions "$x == 3$" and "$x == 5$," respectively, where x is a state variable. If x is in scope at l_0, then h_1 and h_2 share x. The condition of compatibility requires that both I_1 and I_2 assign the same value to x. It follows that h_1 and h_2 cannot be made simultaneously consistent (since, then, x would be equal to 3 and 5 at the same time), and thus, state h is not compatible. Otherwise, if x is a state variable that is *not* in scope at l_0, i.e., it enters in scope in l_1 and l_2 separately, then x is not shared, and the compatibility condition can be satisfied by assigning x to 3 in I_1 and to 5 in I_2 to ensure consistency.

An interpretation may be incomplete; that is, not all the variables that are in scope have to be assigned a value. Typically, we start with an interpretation that assigns values only to input and present-state variables. Later, when the system transitions to the next state, we extend it to a compatible interpretation that assigns values also to output and next-state variables. Compatibility is therefore the criterion used during simulation to select a set of next states that are consistent with the chosen interpretation. Different choices for output variables will therefore result in different sets of compatible next states. This process will be described in more detail later in this paper.

The transitions between global states are computed bottom-up starting from the leaves. From a leaf l, we first determine the set of next states $next(l)$ (obtained from the construction in Fig. 8) found at the same level of the hierarchy. If no final or compound state is reached, then the structure of the tree is unchanged, except that we update the leaf node. Otherwise, if we reach a final state, then we trim the branch of the tree at the parent compound state and compute the next states from there. If, instead, we reach a compound state, then we extend the branch of the tree to a new level, which is initialized to the initial state. In addition, every time a transient state is reached, we immediately continue to its next states until a labeled or compound state is reached. This process is implemented by a recursive procedure, as described in the next section.

VI. Simulation Algorithm

The simulation of a system evolves through a series of transitions between the global states of the finite-state structure. Given a global state and a set of input values, the possible destination states include all those next states whose label can be satisfied by an appropriate assignment to the output and state variables. In general, there may be one or more possible destination states; that is, the FSM may be nondeterministic. We distinguish between two types of nondeterminism: 1) *state nondeterminism*, where the choice has to do only with which state to transition to; and 2) *output nondeterminism*, where the choice involves selecting among several output values that are consistent with the execution. We do not resolve state nondeterminism, but rather maintain a record of all the possible current states that the system may transition to, thus performing at runtime the subsets construction [11], [23]. We cannot follow the same approach to solve the output nondeterminism, as the transactor needs to generate a unique output at every point in time. Our approach is to offer a simple heuristic for resolving output nondeterminism and to keep it isolated so that the user can easily replace it with a more sophisticated strategy.

```
    function Simulate
1       Curr = {(g_0, I_0)};
2       forever
3           Wait();
4           UpdateInputs(Curr);
5           Temp = ⋃_{g∈Curr} Next(g);
6           Curr = UpdateOutputs(Temp);
        end forever
    end Simulate
```

Fig. 10. Simulation algorithm.

The overall simulation algorithm is shown in Fig. 10. The procedure is similar to [23, Algorithm 3.4] (the two-stack algorithm), with the additional computation of output variables and handling of the hierarchy. Variable *Curr* is used to keep track of the set of current states. It is initialized in step 1 to the initial state g_0 of the top FSM, and to I_0, which is the interpretation that assigns initial values to global variables. The algorithm then enters an infinite loop in step 2, and it immediately stops to wait in step 3 for the next triggering event, indicating that inputs have arrived and that outputs need to be generated. The triggering event is usually a clock edge, but our approach is not limited to this case.

The following three steps execute a transition: First, in step 4, the function *UpdateInputs* updates the interpretation of all states in the set *Curr* to reflect the current value of the inputs. Next, in step 5, the set of next states for all states in *Curr* are combined to compute the set of provisional next states that are consistent with the input assignment. The set is provisional because only those states that are consistent with the (yet to be) chosen assignment to the output and next-state variables will actually be retained. We will discuss this step in more detail later in this section.

The output assignment is determined in step 6. To understand the process of output selection, we need to extend the notion of state compatibility to sets of global states under incomplete interpretations. Intuitively, two global states are compatible if their labels can be simultaneously satisfied by interpretations that agree on the variables shared by the two states. We say that a set of global states is compatible if:

1) leaf nodes of all the states in the set are labeled local states (i.e., none of the leaf states are transient);
2) the interpretations associated with all the leaf states can be extended to output and next-state variables so that all leaf labels are satisfied (consistency);
3) if two states share variable x, then the extensions in condition 2 assign the same value to x (compatibility).

It can be shown from the definition of *Next* (as discussed later in this section and shown in Fig. 11) that each state of *Temp* in step 6, taken individually, satisfies the aforementioned conditions and is therefore compatible. The set *Temp* taken as a whole, however, may not. The task of *UpdateOutputs* is to find a compatible subset of *Temp* and the interpretation extensions

```
    function Next(s)
1       if (s == [l, I])
            // l is a labeled/transient state
2           Res = SimpleNext(l, I);
3       else if (s == ([l, I], q))
            // l is a subsequence state
4           Res = {([l, I], q') | q' ∈ Next(q) and q' not final };
5           if (there are final states in Next(q))
6               Res = Res ∪ SimpleNext(l, I);
8       else if (s == ([l, I], q, r))
            // l is a parallel state
9           Res = {([l, I], q', r') | q' ∈ Next(q), r' ∈ Next(r) and
                                      {q', r'} compatible};
10      if (there are final states in Next(q) and Next(r))
11          Res = Res ∪ SimpleNext(l, I);
12  return Res;
    end Next
```

Fig. 11. Computation of next states.

```
    function SimpleNext(l, I)
1       Res = ∅;
2       forall (n ∈ next(l))
3           if (n is consistent with I)
5               if (n is labeled or n is final
6                   Res = Res ∪ {Global (n,I)};
7               else
                    // n is a compond or a non-final transient state
8                   Res = Res ∪ Next{Global (n,I)};
    return Res;
    end SimpleNext
```

Fig. 12. Helper function for the computation of next states.

satisfying the compatibility conditions 2 and 3. The set depends on the particular choice of values for output variables. Once a compatible set is found, *UpdateOutputs* updates *all* interpretations in the selected subset of *Temp*, making sure that all states satisfy condition 3. Finally, it updates the values of current-state variables to reflect the selected next-state values, removes the next-state variables from all the interpretations (making them undefined), and returns the selected subset, which now becomes the new set of current states.

In our implementation, the compatibility conditions are efficiently enforced by storing in the interpretations only a reference to actual variables and making sure that interpretations associated with all states that share a variable have a reference to the same actual variable. Note also that *UpdateOutputs* has some flexibility in choosing outputs if the FSMs are *output nondeterministic*, i.e., if in a given state for given inputs, output variables may be assigned several values. Different choices may affect the quality of verification, as defined, for example, by a coverage metric. Therefore, in our implementation of the simulation algorithm, we provide a customization interface so that the user (or a verification tool) may make a choice that is smarter than our default one.

The function *Next*, as shown in Fig. 11, computes the set of next states of a state g. Since the FSM is hierarchical, the procedure must traverse the levels of the hierarchy each time a compound state is reached or when an FSM at some level reaches its final state. In addition, the machine must transition out of transient states immediately, without waiting for the next simulation step, since transient states are used only for bookkeeping and are not part of the real state of the system. Thus, transient states correspond to the ϵ moves in [11] and [23], with the addition of data operations.

The function *Next* differs depending on the type of the root local state associated with g. If g is a labeled or transient state $[l, I]$, then the algorithm computes and returns (step 2) the states reachable from l in one step by calling *SimpleNext*, as discussed in the following paragraph. For compound states, the procedure explores next states at the current and at the lower level of the hierarchy. For a subsequence state $g = ([l, I], q)$, we first compute all states reachable from q in the lower level by recursively calling *Next* on q (step 4). If this search reaches the final state of the subsequence, in addition to the already computed set, we also compute and return the states reachable from l at the current level (step 6). Similarly, if g is a parallel state $g = ([l, I], q, r)$, we first compute the set of compatible pairs of states reachable in the associated FSMs (step 9). If both of the parallel subsequences reach their final states, then we continue the search at the current level (step 11).

SimpleNext, as shown in Fig. 12, actually executes the transitions. It first iterates over all local next states $next(l)$ of l (step 2) and removes from the computation those states that are not consistent with the current interpretation, i.e., transient states with an entering condition that is not satisfied by I, or labeled states where the label is not satisfiable in the context of I. Then, for labeled and final states, it simply constructs a new global state (step 6). Transient nonfinal states (the ϵ moves) are instead processed immediately by recursively calling the function *Next* (step 8). The same occurs for compound states, so that the appropriate substates of the FSM can be activated.

The function $Global(n, I)$ creates a new global state with n at its root. If n is a labeled or transient state, then $Global(n, I) = [n, I']$, where I' is the same as I except that I' reflects the variable updates associated with n (if there are any). Also, the variables that are no longer in scope at n are not assigned a value by I', and the variables that are just entering in scope at n are assigned their initial values by I'. If n is a subsequence state, then $Global(n, I) = ([n, I], ([s, I']))$, where s is the initial state of the FSM associated with n, and I' is the same as I except that actual parameters are exchanged for formal parameters. If n is a parallel state, then $Global(n, I) = ([n, I], ([s, I]), ([q, I]))$, where s and q are initial states of the FSMs associated with n.

For example, if n is the state corresponding to the instantiation $\langle$`burst(num_bytes/2)`$\rangle$ in Fig. 7, then $Global(n, I) = ([n, I], ([s, I']))$, where s is the initial state of the FSM for `burst`, and I' is the same as I except that I' assigns to `nb` half of the value that I assigns to `num_bytes`. Similarly, if n is the state corresponding to the && operator in Fig. 7, then $Global(n, I) = ([n, I], ([s, I]), ([q, I]))$, where s is the initial state of the FSM for the subexpression

$$\{(\texttt{addr} = \texttt{start\_addr}); \langle \texttt{addr\_cntrl} \rangle\}$$

and q is the initial state of the FSM for the subexpression

$$\{\langle \texttt{burst(num\_bytes)} \rangle \mid \langle \texttt{burst(num\_bytes/2)} \rangle [*2]\}\,.$$

VII. Code Generation

We have implemented the FSM generation and the simulation algorithms for several target verification environments. In all cases, the tool generates code to build the hierarchical FSM data structure and to instantiate the appropriate ports and transaction-level functions to connect the transactor to the rest of the system. Additional code is generated to evaluate input constraints for consistency checking, and it executes assignments to output and state variables. Specific hooks in the code have been reserved to let the user modify the strategy to solve output nondeterminism. In addition to the transactor code, the tool generates a procedure to implement each of the protocol functions served by the transactor. This code intercepts function calls issued by the transaction-level testbench; that is, it computes the value of action expressions such as `B(func)` and then returns when the machine reaches a state labeled with `E(func)`. The data handling associated with served functions is instead obtained by manipulating the interpretations.

The rest of the code, i.e., the algorithm that traverses the FSM data structure to compute one transition and the set of next states (the execution engine), and the code to interface to the simulation environment are not transaction specific. These have been developed manually and are collected in a precompiled runtime library that is simply linked during program execution. For C++, the library consists of approximately 4000 lines of code for the execution engine and another 300 lines to interface to the simulation environment.

A. C++ Code Generation

In general, a transactor needs to interface both to transaction-level modules through served and used functions and to RTL modules through ports. Function calls are natively supported in C++. However, communication through ports 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 runtime wrappers to specialize ports for the particular verification environment used. For example, we have developed a runtime wrapper that creates SystemC ports [1] out of generic ports. We have also developed alternative runtime wrappers that can be used with the TestBuilder (TB) environment [32]. If any other similar C++-based environments became relevant, we can easily develop new wrappers without the need to change the code-generation part of our system.

B. Verilog Code Generation

The algorithm in Section VI is not directly suitable for Verilog implementation because data structures representing both global states and sets of such states are dynamic. While it is in many cases possible to tightly bound the size of a representation of an individual global state, it is usually much harder to find a reasonable *a priori* bound on the number of such states that need to be stored at the same time. Our approach is to make this bound a user-controlled parameter. A warning is reported if this bound is exceeded during an execution. The simulation results represent correct system behaviors even if this happens. However, some of the correct behaviors may be impossible to exhibit (see also Section IX for a discussion of these choices).

As mentioned earlier, transactors need to offer both function-calling interfaces to TLMs and port interfaces to RTL models. For Verilog code, port communication is natively supported, whereas there is no clear best way to do function calls. 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.

C. SCE-MI-Compliant Transactor

SCE-MI is a protocol supporting *transaction-based acceleration*, where transactors consist of a software (SW) part and a hardware (HW) part communicating through SCE-MI-defined ports [4]. The HW part, which is 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.

The generated code consists of two parts. The SW part, which is written in C++, implements functions served by the transactor. The implementation of the served functions 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 the 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. All the detailed RTL signal manipulation, which would usually require a much higher bandwidth, takes place instead on the accelerated HW side. This is important because the HW–SW bandwidth may limit the gains obtained by emulating the HW part.

VIII. 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 TLM of at least the testbench, if not the whole design. Every case included several handwritten transactors based on standard protocols. In every case, we formally specified the protocol and used our system to generate transactors. We then replaced handwritten transactors with the ones we generated and verified that the overall behavior did not change.

In developing our code generator, we have been primarily concerned with the verification methodology and with avoiding state explosion, rather than with optimizing simulation speed. Nonetheless, for all cases, there was no observable change in performance between the simulation of the system with

TABLE I
TRANSACTOR GENERATION CASE STUDIES

design	States	lines of code					
		hand	PSL	C++	Ver	SCE-MI C++	SCE-MI Veril
ATM (TB)	59	1012	60	634	299	72	313
UART (Ver)	50	55	53	450		126	169
SoC (C++)	279		513	5600		100	11000

the handwritten transactor and with the generated transactor. Indeed, the bulk of the simulation time is likely to be spent in the (RTL) code for the device under verification, thus severely limiting the impact of any optimization in the transactors. We have therefore simply focused on avoiding obvious inefficiencies. For designs that are transactor dominated, however, optimization techniques might be required to achieve acceptable performance [33], [34].

The first test case is an ATM switch design that follows the Utopia protocol to transfer ATM cells. It was verified by a C++ transaction-level testbench generating ATM packets that were then converted to RTL signals by a handwritten transactor using TB utilities. We replaced this transactor with the one we generated in C++, supported by the runtime wrappers we have developed for TB.

The second test case is a universal asynchronous receiver/transmitter (UART) design that communicates to the outside world through the on-chip peripheral bus protocol [35]. It was originally verified by a transaction-level testbench and transactors, both of which were written in Verilog. Therefore, we have used this case study to test our Verilog code generation. We have also rewritten the testbench in C++ to test our C++ code generation with SystemC runtime wrappers. Finally, we have tested our SCE-MI-compliant code generation using the same testbench.

The third test case is a complex system-on-chip (SoC) design consisting of two processors, four direct memory access channels, and a number of application-specific integrated circuit (ASIC) engines targeted at multimedia application. A partial TLM in SystemC has been created for this design. This model communicates with the rest of the system (written in RTL Verilog) through the advanced microcontroller bus architecture (AMBA) [36], and handwritten transactors in C++ were used to convert AMBA transactions into sequences of RTL signals. We have replaced these handwritten transactors in two experiments: once by transactors we generated in C++ with SystemC runtime wrappers and once by the generated SCE-MI-compliant transactors. The generated transactors did not support arbitration and split transfers since the system did not make use of these features. However, they did support both single and burst transfers, as well as overlapped (pipelined) transactions.

Table I shows the experimental results. The table lists the number of generated states, the size of the handwritten transactors, the size of the formal protocol specifications in PSL, and the size of the generated C++, Verilog, and SCE-MI-compliant codes (both the C++ and Verilog part). The UART design did not use general-purpose transactors, but rather some code specific to this testbench that acted as a transactor. Thus, in this cases, the original transactors were rather small and very similar in size to the formal protocol specification. The size of the generated transactor is considerably larger than that of the handwritten one, but it is still relatively small and, as we explained earlier, grows 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 the 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, our protocol specification did not cover the complete protocol, but only the features exercised by the testbenches. Hence, the handwritten transactor also has much larger functionality than the transactor generated from our protocol specification.

Our largest case study is the SoC design. In this case, the original handwritten transactors were not available in source code but were part of a dynamic library. We are therefore unable to report the number of lines of code. This study, however, shows that a relatively complex transactor can be specified compactly in our extended PSL.

The full comparison of our approach versus the handwritten one must also take into account the size of the runtime library, which consists of approximately 4300 lines of code. However, developing this library is a one-time effort that can be amortized over many designs. In addition, the execution engine and the hierarchical FSM representation are generic. The runtime library could therefore be used for other purposes where a compact nondeterministic representation is required. Exploring other uses of our technique, such as in architecture analysis, is part of our future work.

IX. DESIGN CHOICES AND RELATED WORK

Our approach to code generation is inspired by the production based specification (PBS) framework and its evolutions, such as Clairvoyant and Protocol Compiler [15], [37], but it differs significantly in objectives and focus. As for PBS, our specification mechanism is based on regular expressions. In PBS, besides the operators, the specification is grammar based and partitioned into conditions, which depend on the input variables, and actions, which set the value of the output variables. This distinction is useful during synthesis. However, this is contrary to our methodology, where the transactor specification is shared between all protocol roles, and where the direction (input/output) of the signals must be initially abstracted away. Some of our extensions to PSL are, in fact, motivated by the requirement of representing "actions" in an abstract form within the context of a Boolean expression. These considerations also justify our use of the sum-of-product form for representing Boolean and data expressions, which is easier to handle when extracting the information about output assignment than the binary decision diagram (BDD) representation used in PBS. Indeed, actions in PBS are represented procedurally, which is a choice that makes it hard to represent the output nondeterminism that may arise as a consequence of a particular partition of the signals into inputs and outputs. In addition,

our extensions are conceived to directly support function-call-based communication, which is an area that is not covered by PBS. Besides these modeling aspects, PBS also faces the same issues regarding state explosion during synthesis. Their implementation makes use of an encoding of the global state that effectively keeps track of all the possible nondeterministic states simultaneously by asserting a set of bits, each of which corresponds to one of the currently active states. This is therefore equivalent to executing the subsets construction at runtime [23]. This scheme, which is typical also of Esterel compilers [28], is particularly indicated for a hardware implementation of the protocol since hardware is good at working with bits and the representation is very compact. Our focus, however, is not on the synthesis of hardware as in PBS, but rather on the *software simulation* of the transactor protocol. Indeed, transactors are regarded as part of the testbench and are not generally intended for inclusion in the final system. As such, we have opted for a simpler synthesis strategy, which is inspired by parser generators (see next paragraph), that is easy to implement in software and that facilitates the creation and simulation of a hierarchical state machine. For this reason, and to avoid potential state explosion connected with one-hot encoding of data or counter variables, we have maintained the same approach also for the Verilog code generation. Similarly to our hierarchical representation, Seawright and Meyer also consider the problem of partitioning the synthesized hardware by taking advantage of the hierarchical nature of the specification [38]. The process involves the application of reachability analysis to compute the set of sequential don't cares required for Boolean simplification and optimization. Using this technique, the authors demonstrate a substantial reduction in the size of the generated circuit and an increase in performance. Such complex procedure is justified in the case of hardware synthesis to achieve higher performance or lower power and, more importantly, where the area of the circuit is directly proportional to the production cost. These problems are far less relevant in simulation, where program memory is relatively cheap, and simulation speed is not affected by code size the way hardware is. Therefore, we were satisfied with avoiding state and code-size explosion and did not pursue further opportunities for optimization. That notwithstanding, advanced optimization techniques could be applied in the case of transactor generation for hardware-based acceleration. The evaluation of the tradeoffs that are involved in this choice is, however, part of our future work.

Similarly to PBS, our framework resembles the specification style and implementation strategy employed in parser generators such as Lex and Yacc [39], [40]. Our modeling approach is somewhat simplified since it is based on regular expressions and is therefore not concerned with lookahead. On the other hand, as discussed before, our model is distinguished by the implicit (as opposed to explicit) representation of actions that we use to obtain transparency with respect to protocol role. In addition, we must handle the parallel composition operator and output nondeterminism, which are not present in parser specifications. Several techniques have been developed for avoiding state explosion. As shown, we rely on the runtime subsets construction [23, Algorithm 3.4]). Other techniques include *lazy evaluation* [3, p. 128] (where the explicit transition structure itself is built only at runtime for the part of the state machine that is actually traversed) and alternative ways of constructing, partitioning, and/or encoding the finite automata [34], [41]. While it would be possible to use these methods in our case, we have again preferred to use a simple translation algorithm and to focus on the aspects of the translation that are characteristic to our problem. In particular, for partitioning, we rely on the natural decomposition denoted by the hierarchy building operators of our source language.

X. Conclusion

The development of transactors connecting TLMs to RTL models 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 versus acceleration). In addition, formal interface specifications can be used as assertions and verified either dynamically or statically (in which case, some abstraction may be necessary).

The specification formalism supports data variables, nondeterminism, parallelism, and submodule instantiation. Our scheme avoids state-space explosion associated with a naive treatment of these features by postponing dealing with them until simulation time. The burden is thus shifted to the simulation algorithm, which we have also proposed. In the worst case, the number of states visited in simulation may be exponential in the size of original specification, but, in practice, it is typically proportional to the size of the input sequence. Our approach can be applied to FSM generation from other formalisms that have some or all of the critical features.

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] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Norwell, MA: Kluwer, 2002.

[2] L. Cai and D. Gajski, "Transaction level modeling: An overview," in *Proc. Int. Conf. Hardware/Software Codesign and Syst. Synthesis*, Newport Beach, CA, Oct. 2003, pp. 19–24.

[3] A. Bruce, A. Nightingale, N. Romdhane, K. Hashmi, S. Beavis, and C. Lennard, "Maintaining consistency between SystemC and RTL system designs," in *Proc. Des. Autom. Conf.*, San Francisco, CA, Jul. 24–28, 2006, pp. 85–89.

[4] *Standard Co-emulation Modelling Interface (SCE-MI): Reference Manual (DRAFT)*, May 2003. [Online]. Available: http://www.eda.org/itc.

[5] *Property Specification Language: Reference Manual, Version 1.1*, Jun. 9, 2004. [Online]. Available: http://www.eda.org/ieee-1850

[6] G. Borriello and R. H. Katz, "Synthesis and optimization of interface transducer logic," in *Proc. Int. Conf. Comput.-Aided Des.*, Nov. 1987, pp. 274–277.

[7] S. Narayan and D. D. Gajski, "Interfacing incompatible protocols using interface process generation," in *Proc. 32nd Des. Autom. Conf.*, San Francisco, CA, Jun. 12–16, 1995, pp. 468–473.

[8] R. Passerone, J. A. Rowson, and A. L. Sangiovanni-Vincentelli, "Automatic synthesis of interfaces between incompatible protocols," in *Proc. 35th Des. Autom. Conf.*, San Francisco, CA, Jun. 1998, pp. 8–13.
[9] J. Akella and K. McMillan, "Synthesizing converters between finite state protocols," in *Proc. Int. Conf. Comput. Des.*, Cambridge, MA, Oct. 14/15, 1991, pp. 410–413.
[10] 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 *Proc. IEEE/ACM ICCAD*, Nov. 2002, pp. 132–139.
[11] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley, 1986.
[12] A. Pnueli, "The temporal logic of programs," in *Proc. 18th IEEE Symp. Foundations Comput. Sci.*, Oct. 1977, pp. 46–57.
[13] *System Verilog 3.1: Accellera's Extensions to Verilog*, 2003. [Online]. Available: http://www.eda.org/sv
[14] P. Wolper, "Temporal logic can be more expressive," *Inf. Control*, vol. 56, no. 1/2, pp. 72–99, 1983.
[15] A. Seawright and F. Brewer, "Clairvoyant: A synthesis system for production-based specification," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 2, no. 2, pp. 172–185, Jun. 1994.
[16] J. Oberg, A. Kumar, and A. Hemani, "Grammar-based hardware synthesis of data communication protocols," in *Proc. 9th Int. Symp. Syst. Synthesis*, La Jolla, CA, Nov. 6–8, 1996, pp. 14–19.
[17] K. Shimizu, D. L. Dill, and A. J. Hu, "Monitor-based formal specification of PCI," in *Proc. 3rd Int. Conf. Formal Methods Comput.-Aided Des.*, Austin, TX, Nov. 2000, pp. 335–353.
[18] R. Siegmund and D. Müller, "A novel synthesis technique for communication controller hardware from declarative data communication protocol specifications," in *Proc. 39th Conf. Des. Autom.*, New Orleans, LA, Jun. 2002, pp. 602–607.
[19] M. T. Oliveira and A. J. Hu, "High-level specification and automatic generation of IP interface monitors," in *Proc. 39th Des. Autom. Conf.*, New Orleans, LA, Jun. 2002, pp. 129–134.
[20] *TransactorWizard*. [Online]. Available: http://www.sdvinc.com
[21] T. Michiels, "Generating TLM bus models from formal protocol specifications," presented at the 9th Eur. SystemC Users Group Meeting, Feb. 2004. [Online]. Available: http://www-ti.informatik.uni-tuebingen.de/~systemc/ninth_escugm.html
[22] *Cohesive*. [Online]. Available: http://www.spiratech.com
[23] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.
[24] 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 Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 18, no. 6, pp. 834–849, Jun. 1999.
[25] R. W. Floyd and J. D. Ullman, "The compilation of regular expressions into integrated circuits," *J. ACM*, vol. 29, no. 3, pp. 603–622, Jul. 1982.
[26] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal, "FoCs: Automatic generation of simulation checkers from formal specifications," in *Proc. Comput. Aided Verification*, 2000, pp. 538–542.
[27] G. Berry and G. Gonthier, "The ESTEREL synchronous programming language: Design, semantics, implementation," *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, Nov. 1992.
[28] S. Edwards, "An Esterel compiler for large control-dominated systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 21, no. 2, pp. 169–183, Feb. 2002.
[29] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, Jun. 1987.
[30] D. Har'el, H. Lachover, A. Naamad, A. Pnueli *et al.*, "STATEMATE: A working environment for the development of complex reactive systems," *IEEE Trans. Softw. Eng.*, vol. 16, no. 4, pp. 403–414, Apr. 1990.
[31] *The ATM Forum Technical Committee*, Jun. 1995. Utopia Level 2, Version 1.0. [Online]. Available: http://www.atmforum.com/standards/approved.html
[32] *TestBuilder*. [Online]. Available: http://www.testbuilder.net
[33] W. M. Waite, "The cost of lexical analysis," *Softw. Pract. Exp.*, vol. 16, no. 5, pp. 437–488, May 1986.
[34] G. Navarro and M. Raffinot, "Compact DFA representation for fast regular expression search," in *Proc. 5th Workshop Algorithm Eng.*, 2001, vol. 2141, pp. 1–12.
[35] IBM, *On-Chip Peripheral Bus*. [Online]. Available: http://www.ibm.com/chips/products/coreconnect
[36] *AMBA Home Page*. [Online]. Available: http://www.arm.com/products/solutions/AMBAHomePage.html
[37] A. Seawright, U. Holtmann, W. Meyer, B. Pangrle, R. Verbrugghe, and J. Buck, "A system for compiling and debugging structured data processing controllers," in *Proc. Eur. Des. Autom. Conf.*, Geneva, Switzerland, Sep. 1996, pp. 86–91.
[38] A. Seawright and W. Meyer, "Partitioning and optimizing controllers synthesized from hierarchical high-level descriptions," in *Proc. 35th Des. Autom. Conf.*, San Francisco, CA, Jun. 1998, pp. 770–775.
[39] M. E. Lesk, "Lex—A lexical analyzer generator," AT&T Bell Lab., Murray Hill, NJ, Computing Science Tech. Rep. 39, 1975.
[40] S. C. Johnson, "Yacc—Yet another compiler compiler," AT&T Bell Laboratories, Murray Hill, NJ, Computing Science Technical Report 32, 1975.
[41] G. Myers, "A four Russian algorithm for regular expression pattern matching," *J. ACM*, vol. 39, no. 2, pp. 432–448, 1992.

Felice Balarin (S'90–M'95) received the Ph.D. degree in electrical engineering and computer sciences from the University of California, Berkeley, in 1994.

Since 1994, he has been a Research Scientist with the Cadence Berkeley Laboratories, Cadence Design Systems, Inc., Berkeley. His research is focused on the development and application of formal methods to design, verification, specification, and analysis of embedded systems implemented by both hardware and software. He is the author of numerous papers and coauthor of two books on these topics.

Roberto Passerone (S'96–M'05) received the Laurea degree (*summa cum laude*) in electrical engineering from the Politecnico di Torino, Torino, Italy, in 1994 and the Master's and Ph.D. degrees in electrical engineering and computer sciences from the University of California, Berkeley, in 1997 and 2004, respectively.

From 1998 to 2005, he was with Cadence Design Systems, Inc., Berkeley, where he held various positions from Senior Member of Technical Staff in the System Level Design Product Group, to Research Scientist in the Cadence Berkeley Laboratories. Since 2006, he has been an Assistant Professor with the Department of Information and Communication Technology, University of Trento, Trento, Italy. His research interests include the design and implementation of high-performance microprocessors, system-level design, communication design, and formal methods. In particular, his research has focused on the development of methods for the automatic synthesis of protocol converters and transactors and for the analysis of the semantic foundations of heterogeneous systems.