Synthesis of Glue Logic, Transactors, Multiplexors and Serialisors from Protocol Specifications.

DJ Greaves
University of Cambridge, Computer Laboratory
Cambridge, UK.
David.Greaves@cl.cam.ac.uk

MJ Nam
University of Cambridge, Computer Laboratory
Cambridge, UK.
mjn31@cl.cam.ac.uk

***Abstract*—Today's system-on-chip (SoC) systems must be designed as quickly as possible by integrating IP blocks from diverse suppliers. In this paper, we present a new automata-based algorithm that automatically synthesizes glue logic for SoC fabrication and Transaction-level modelling (TLM) transactors for SoC modelling. Our approach introduces a new encoding for state variables which captures data conservation property and supports simple point-to-point connections as well as those the perform functions such as multiplexing, filtering and serialising.**

I. Introduction

The large scale and complexity of today's system-on-chip (SOC) demand inventive techniques and tools that simplify the design and verification process. There are several approaches to shorten time-to-market, and widely used approaches are IP reuse and transaction level modelling (TLM) [1]. IP Reuse simply means reusing some pieces of the existing designs. TLM is an electrical system level (ESL) modelling concept that allows designers to abstract hardware (HW) signals to abstract operations in a higher level. At the TLM, functional calls can be used to execute read/write operations, and also the functionality can be modelled using a higher level descrption and more abstract date objects. IP reuse enables the team to leverage the cost and verification across multiple designs and is proven to increase design productivity. TLM has been also increasingly adopted for advanced SOC design and verification to address the limitation of pure RTL modelling methodologies.

The key element of the reuse of IP blocks and TLM is to make the IP blocks as close to *plug-and-play* as technically possible. Manually adapting these interfaces is tedious and may cause human errors, hence, there is a need for automated interface synthesis both in the same level and the mixed-level communications. This paper is aimed at automatic interface synthesis adapting two incompatible interfaces in the same level or in the mixed-level. A common framework is presented that can be used both for automatic synthesis of glue logic between IP blocks in SoC design, between TLM models during modelling and for and of initiator and target TLM to net-level transactors during mixed mode simulation of a SoC. The framework extends the well-known product synthesis method by ranging over symbolic dead/live values. We also show how the technique can be applied to multi-way connections that implement multiplexing and/or serialisation.

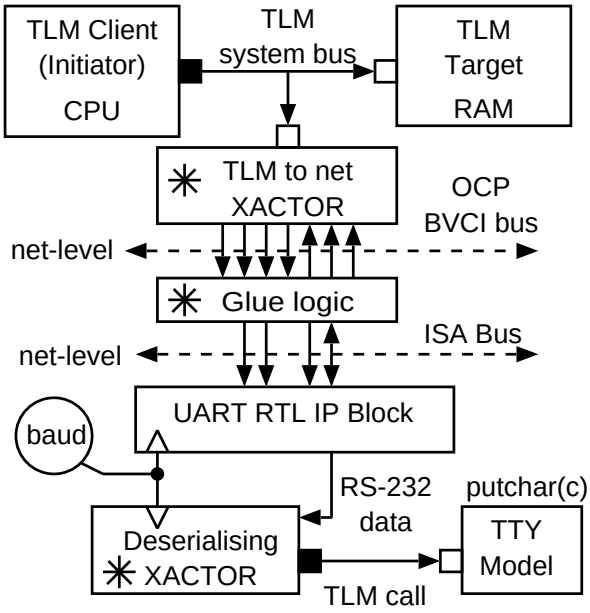

Fig. 1. Example using three (denoted with asterisks) applications of our method.

Our contributions are the symbolic extension to the product method (§IV) and its use for transactional-level models (§VI).

Figure 1 illustrates a typical situation where our technique can be used in three different places as part of a system model and for synthesis. A high-level model of a CPU connects to its memory and I/O sub-systems using TLM calls. While one I/O component of interest is a legacy UART with ISA connections, the SoC architecture uses the OCP BVCI ports on IP blocks, therefore, there must be a net-level glue logic interfacing between BVCI and ISA. Also, we need a transactor connecting the net-level BVCI port to the TLM processor model. This paper shows how to generate both of the glue-logic and the transactor. The third application of our technique is to generate a high-level model of a receiving UART that makes a software call to the workstation `putchar` method for each character deserialised. Our technique works with any mix of synchronous and asynchronous participants, provided all synchronous participants share the same clock

domain. The first two cases mentioned above deal with a communication bewteen one synchronous and one asynchronous participant (BVCI is a synchronous protocol.), and the third case is between two synchronous participants (both ends of the serial link were clocked from the same baud rate generator).

II. Related Work

Automated interface converter generation has been addressed in the literature from different perspectives. We focus on work done in the context of small subset of real hardware design based on Finite State Machine (FSM) models.

In the early work [2], protocols were presented as FSMs, and their cross product was used to construct a *converter*. This approach was later extended, and became the foundation of much work in interface synthesis problem. Sangiovanni-Vincentelli et al [3] synthesized a converter FSM, based on selecting the non-deadlocking paths through the cross product of a pair of FSMs. These machines can be composed synchronously, where they both move at once, or asynchronously, where they take it in turns. Interface synthesis using a SAT solver to populate a fictional FPGA was presented in [4].

There have been approaches to extend FSM-based converter problem with datapath issues. The early work by Gajski defined 'Finite State Machine with Datapath' [5] in a semi-formal way, and the paper [6] inspired by this work proposed more formalized definitions of the notions of assignments and statuses. In the paper [7], the work was extended with datapath width adaptation. The authors of [8] introduced the data path state machine which captures data path dependencies in the converter problem between two synchronous hardware modules with data communication protocols.

Transactor generation methods for cross-level communication (TLM-RTL) has been proposed in [**?**] [**?**] [**?**]. These approaches are also based on finite automata. [**?**] proposed a methodology where protocols manually described in a formal language, Property Specification Language (PSL) are transformed into a FSM, followd by the synthesis of simulation code, while Extended Finite State Machine (EFSM) are exploited in [**?**]. The methods proposed in both papers require designers to fully describe the formalism of the protocols. [**?**] presented a technique to automate the transactor generation for RTL IP components to be reused in TLM systems. The protocol information are extracted from testbenches by exploiting the EFSM models. This methodology assumes that RTL testbench is implemented with RTL IP components, that is a manual process.

III. Protocol Description

In this section, we show the structure of the system that we deal with. A system consists of a number of *components* that desire to communicate with each other. The basic unit of concurrency is a *finite-state automaton* (FSA). Each component is connected via a net-level or TLM interface that obeys an associated protocol specified by a *protocol FSA*, of which we have full knowledge. The protocol automata, in turn, receive input from unspecified circuitry elsewhere in their component (later called fv) and, from our point of view, these inputs can be changed at any time. According to context, we sometime use the term, 'participant' to denote one of the participating protocol automata and sometimes to also range over the joining automaton.

Protocol P = Loop `of` (ρ) where
ρ = Eq `of` $\alpha * \alpha$ `list` // Parallel assignment
| Seq `of` ρ `list` // Sequencing
| Disj `of` ρ `list` // Non-deterministic branching
| Next // Wait one clock (same as Eq `nil`.)
where α is an integer expression ranging over the interface nets (Table II).

TABLE I
Abstract Syntax for the Protocol Capture Language used in our experiments, given as an ML-like datastructure.

In general, many different input language constructs are useful for protocol and transactor specification. These include a wide range of commercial and experimental temporal logic and assertion languages. In our experiments, we used a combination of automatic and manual conversion from various sources to a common protocol representation with abstract syntax tree shown in Table I. Each protocol, P is represented as an infinite loop of a node ρ that is a recursively defined structure using three forms. An 'Eq' node defines a list of pairs of expressions which must be pairwise equal when the node is executed. For example, if one half of a pair is an output or local variable and the other half is an expression that is a function of inputs, then they are made equal with an assignment that copies the expression to the output or local. If one half is an input and the other is a constant, then the node can only be executed when the input has that value. This is a direction-agnostic style of participant description, in the style of IP-XACT [9]. Whether a particular net is an input or output varies according to the direction of instantiation of the associated interface.

A 'Seq' node defines ordering of events and a 'Disj' node defines forking paths. A fourth node 'Next' is used for synchronous protocols where a clock cycle must be consumed while no part of the interface changes, but this is shorthand for 'Eq `nil`'. Expressions may be paired with user predicates that must be satisfied at the time the expression is evaluated. Each participating protocol is readily compiled into a protocol automaton whose state is the interface nets augmented with a program counter variable that ranges over the 'Seq' nodes and any extra local state that might be needed. For a synchronous product, transitions are taken on the active edge of the clock and for asynchronous product, at any time. Hence, a protocol FSA is defined as a tuple,
$\mathbb{M} =< \Sigma, V, S, S^{init}, S^{idle}, \Delta >$ where,

- Σ: a non-empty set of symbols (the input alphabet),
- V: a set of state variables that each range either over a concrete enumeration or a fixed, finite set of symbolic expressions,
- S: a finite set of states defined by the cross product of V,
- $S^{init} \in S$: a initial state
- $S^{idle} \subset S$: a set of idle states (includes initial state)

- $\Delta : S \times \Sigma \rightarrow S$: a state transition function

The initial predicate S^{init} holds for only one setting of V that corresponds to the start of day, reset state. Some protocols have more than one idle sate. An example is the two-phase handshake that attaches meaning to *changes* of net value and has two idle states: $P^{idle} =$ `Req==Ack`.

The conventional way to represent states and signals of FSM in RTL is with boolean vectors. In this paper, we introduce the concept of *symbolic* values. Each variable of our FSA has either a *symbolic* or a *concrete* value. Concrete variables range over a finite enumeration type. Symbolic variables are registers each of some width in terms of bits, but the bit values are run-time data. During our procedure (i.e. at compile time), symbolic variables are assigned either *dead* (denoted with $\perp$) or *live* with some symbolic expression α. Borrowing terminology from optimising compilers, a symbolic variable becomes live when a new value is stored in it and is killed to dead at its last read before the next write.

The input alphabets Σ, that are also Moore output functions of other connected FSAs are predicates over the concrete and symbolic values of the other state vectors. Certain predicates are routing and filtering conditions needed for certain connection patterns. The combinations of values out of our interests can be abstracted away by assigning predicates to them.

Once an FSA is defined, it is relatively trivial to map the FSA form to synthesisable RTL, a structural netlist, or net-level RTL. In generated SystemC TLM models, however, threads instead of signals pass between components. SystemC transactional modelling can be projected as a FSA quite easily if we make the restriction that every TLM method call is non-reentrant and called from only one point, in which case every return is just a jump. To make the projection, we introduce an *call active* flag into the converter state vector for every TLM interface. This boolean variable is initially clear, and it is set for a target entry point when the thread is logically 'inside' the joining FSA, and it is set for an initiating upcall when the thread is abroad. The formal parameters and return values to the calls are just additional symbolic or concrete variables that, for simplicity, are only used in one direction of the call and hence they are part of the state vector of the one FSA that writes to them (i.e. they are updated by its NSF, except for the death of symbolic variables, which as already stated, is an operation performed by the reader).

Our procedure automatically creates a certain amount of state for the converter in proportion to the product of the participants' states, however, some combinations require additional states such as holding registers in the converter. For instance, any converter that behaves like a mailbox or FIFO queue requires additional internal storage. In these cases, our approach requires the user to add sufficient state resources to fulfil these needs, but tends to avoid using excess such resources when not needed, and fails when insufficient resources were made available.

In order to generate a joining machine for the mixed-level communications, we also should consider asynchronous and synchronous issues. There are two styles of hardware glue logic: asynchronous and synchronous.

$\alpha =$	$\perp$	(dead)
|	D_n	(n-bit register)
|	$\alpha \mid \alpha'$	(bitwise OR)
|	α << N	(constant left shift)
|	kill(α)	(kill expression)
|	$(\alpha, P_{\text{user}}(\alpha'))$	(expression guarded by predicate)

TABLE II
ABSTRACT SYNTAX FOR EXPRESSIONS HELD IN SYMBOLIC VALUES AT COMPILE TIME.

Asynchronous systems do not use a shared clock between the participants whereas there is such a net in a synchronous solution. Asynchronous protocols include the Centronix parallel port, and other similar protocols based on a four-phase handshake, such as the VME bus. Synchronous protocols, such as AHB and OCP BVCI are commonly used in SoC design. The TLM style is asynchronous but commonly transactors for synchronous protocols are needed, hence requiring the product of an asynchronous and a synchronous participant.

IV. DATA-CONSERVING CONGRUENCE

For common transactors and pieces of glue logic, we simply require that the result be *data conserving*: i.e. that it does not drop or repeat any item of data. More-advanced joining patterns include the demultiplexer, the multiplexer, the filter, the serialiser and the deserialiser. We implement these as generalisations of the data conserving product.

Our main contribution is a unification algorithm that implements common data movement patterns. For concrete nets, there is a natural congruence between an arc of an FSA that drives the net with an arc of a receiving FSA that is guarded by that net being driven to that condition. For symbolic nets, we implement data movement, where a live symbolic value is reduced directly to *dead* or to a form with less live data that will then be further reduced. Each reduction may be associated with a user-provided predicate that ranges over the actual contents of the symbolic variable at run time. These guards enable common filtering, routing and multiplexing operations to be expressed.

For brevity, we present only a few forms for the congruence algorithm to range over (Table II), but a richer system should be provided for serious use.

Where a symbolic variable goes live in the input specification it takes on a user-provided value of α. For instance, for serialising or deserialising a 32 bit value over an 8 bit bus, the 8 bit bus would go live with D_8 and the 32 bit bus would go live with $((((E_8 << 8)|E_8) << 8)|E_8) << 8)|E_8$. Where a destination should only accept data that conforms to some predicate then it will go live with $(P_{\text{user\_condition}}(D_n), D_n)$.

The congruence procedure C (Figure 2) accepts input and output abstract syntax trees for the symbolic arguments, α and ω, where ω may receive some or all of the live data from α. C returns a triple containing actions to effect the transfer, a guard expression that must hold if the transfer is to be performed and a remainder α', that represents the left over contents of the input register after executing the commands. Actions are just assignments. In

```
let rec C = function
  | (D_n, D'_m) → ([D'_m := D_n], n=m, ⊥) // Width match
  | (α, P_u(ω)) →
        let (c, g, α') = C(α, ω)
        in (c, g ∧ P_u(α'), α') // Predicate
  | kill(α) →
        let (c, g, α') = C(α, ω)
        in (c, g, ⊥) // Kill
  | (α_l | α_r, ω) →
        let (c, g, α') = C(α_l, ω)
        in (c, g ∧ (α' = ⊥), α_r) // Serialise
  | (α_l, ω_l | ω_r) →
        let (c, g, ω') = C(α, ω_r)
        in (c, g ∧ (ω' = ⊥), ω_l) // Deserialise
  | (α<<N, ω) →
        let (c, g, α') = C(α>>N , ω)
        in ([(α>>N)/α]c, g ∧ (α' = ⊥), ⊥) // Shift out
  | (α, ω<<N) →
        let (c, g, α') = C(α, ω)
        in ([(ω<<N)/ω]c, g ∧ (α' = ⊥), ⊥) // Shift in
```

Fig. 2. Core algorithm of the data-conserving congruence/matching that generates guarded commands to move data from α to ω (ML-like pseudocode).

simple cases, $\alpha' = \bot$. If unification fails, then the returned guard is false.

The order of serialisation is syntax-directed in this simple version of the algorithm. The left-hand operand of every source disjunction is sent first and the right-hand operand is received first, allowing the same expression to denote both the sending and receiving end of a serialiser/deserialiser pair.

Some data is not conserved by the converter. It is locally consumed. This occurs in filters and where data has been tested with a predicate and is no longer needed (such as high-order address bits). The **kill**(α) construct is used in these cases. For convenience, it behaves as an identity function in terms of its return value, allowing us to write the address decoding predicates for the BVCI to ISA glue (32 to 20 bit for memory and 32 to 16 for I/O) as

$$P_{\text{ismem}}(A) = \text{kill}(A >> 12) == \texttt{0xFF0}$$
$$P_{\text{isio}}(A) = \text{kill}(A >> 16) == \texttt{0xFF10}$$

V. Overall Procedure

Our procedure (Figure 3) starts with a master XML file where the user lists the participants that need connecting. Our tool instantiates the interfaces with their associated protocols from a library held in the form of Table I along with user predicates. Net directions for a net-level interface are specialised according to whether the overall interface is an input or an output. The net-level inputs to a target are the outputs of an initiator, and vice versa, except for certain nets, such as reset and clock, that are always inputs and sourced from external third parties. A TLM port must be specialised to be either an invokable target (entry point) or an initiator that invokes a remote method (upcall). We expect that commonly the participants are selected from a

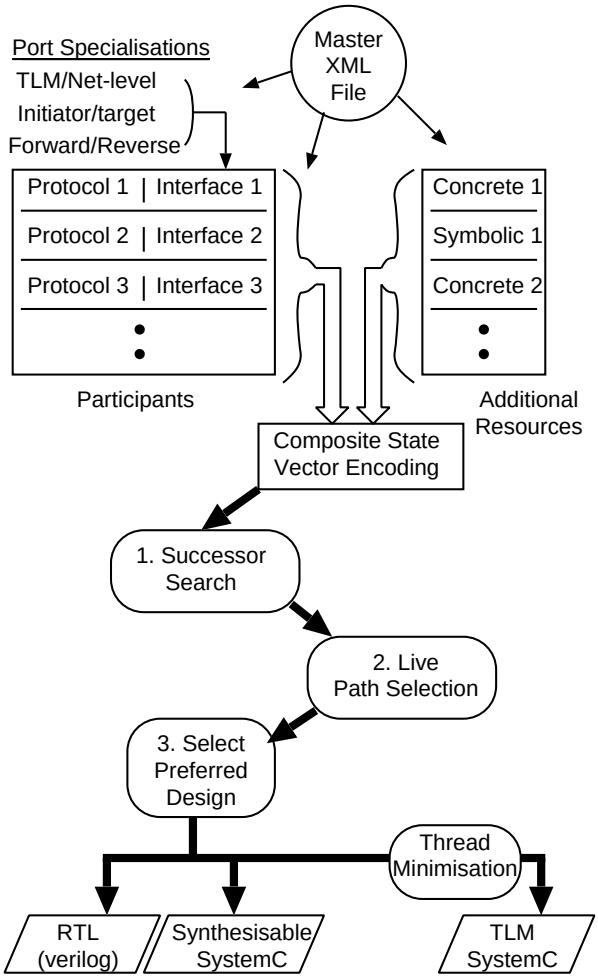

Fig. 3. Flow Diagram for our Method.

library (e.g. in IP-XACT style) of standard protocols and IP blocks.

As well as instantiating protocols and interfaces, the master XML file may invoke additional resources, including holding registers and the state bit. Additional resources are typically not provided with a protocol automaton that restricts their pattern of use, hence they can be freely used in the converter if needed.

A composite state vector is created that consists of the concatenation of the state vectors of the protocol automata of the participating components, the predicates of client FSAs and of the user-provided resources. An encoding converts the composite state vector to a single integer, n, so that it can be used to index an array, recording which states have been processed.

In phase one, starting from the value of n that represents the reset state of all machines, successors are explored recursively until the reachable state space is discovered. The product of automata can be formed in three basic ways: synchronous form, asynchronous turn taking form or stuttering synchronous form, where each participant non-deterministically moves or not, provided at least one does. A synchronous product is appropriate when all of the participants are synchronous. Asynchronous turn-taking form might seem appropriate for asynchronous participants, but cannot be used since our data conservation

1. Q = { Initial_state }; Result = $\emptyset$;
2. select $s \in$ Q-Result; Q := Q - { s };
3. $B := \{(s,s')|\forall \sigma \in \mathbb{P}(fv).\forall mc \in \mathcal{P}(\mathbb{M}).\delta = \bigcup_{m \in mc} E_m(s,\sigma)fv \wedge s' = \delta/s\}$
4. $T := \{s''|\forall n \in \mathbb{Z}^+.\forall (s,s') \in B \wedge \delta = C^n(s,s') \wedge s'' = \delta/s'\}$
5. Q := Q $\cup$ T-R; R := R $\cup$ T;
6. if Q=$\emptyset$ then return R else goto 2

Fig. 4. Successor product forming for phase 1.

```
while(1)
{ // Wait in next line only present when synchronous.
  wait (posedge clock);
  switch(pc)
   case 10:
       if (g1) { v1=e1; v2=e2; ... pc=20; }
       if (g2) { ... pc=34; }

       break;

   case 20:
        ...

   case ...:
 }
```

Fig. 5. Typical structure of raw transactor code before thread optimisation.

```
while (1)
{
   do { sc_wait(0, SC_NS); } while (callstate != active);
   RC = remote_port.call(ARGS);
   callstate = idle;
}
```

Fig. 6. Additional thread to make a TLM initiating port using its own dedicated thread (unoptimised).

rules require that, in a single transition of the product machine, a symbolic register in one participant goes live while the register the data was sourced from, generally in another participant, goes dead. This is not possible within the asynchronous product, so we use the stuttering synchronous product.

Figure 4 outlines the product search algorithm, where fv is all possible settings of the client inputs (including autonomous go dead/live changes), and δ/s denotes updating state s with changes δ. The algorithm maintains a queue of states to explore, seeded from the initial state. For each state, for every possible change of external inputs, for every possible stuttering combination (mc) of participants, the successors are found and added to the results and queue if not already considered. Note that T includes s because no input changes and no execution of any participants are parts of the respective powersets. The intermediate potential transitions set, B, is processed by the symbolic congruence function C, to produce the final set of successors. For efficiency, in our implementation, those with manifestly invalid guards are deleted at this stage, rather than later on. We also save the commands from the algorithm, rather than running it again in a subsequent phase, but for clarity of presentation, this is not shown. Instead $C^n(s,s')$ denotes all changes to symbolic variables needed for a data conserving transition from state s to s' achieved with n successive applications of the congruence algorithm (Fig. 2) with their commands composed and conjunction of guards. The search over increasing n is terminated as soon as invalid guards are generated.

$E_m(s,\sigma)$ denotes the changes produced by stepping participant $m \in \mathbb{M}$ in state s with external inputs σ.

In phase two, we find the live states of the product machine by eliminating all those that lead only to dead ends. The method used is to create successive iterations of the product machine where each state is only retained if any of its immediate successors were present in the previous generation. When two iterations are the same, only infinite paths remain. Then, we form the intersection over each setting of fv of the result of eliminating sub loops that do not satisfy the idle state predicate of at least one participant. This uses a depth-first search from the initial state that records what idle states have been encountered at what level and discards any back arc to a state that records the same pattern of idle states.

In phase three we generate a *basic-block* machine by collapsing successive product states where outputs are changed but no input is tested. A basic block is a sequence of assignments to outputs with a conditional branch to successor basic blocks as the last stage. The basic block machine will typically have a number of branches to different successors that cannot be distinguished by their branch condition. Indeed, a number of them may be unconditional. If there are any unconditional ones then all of the conditional ones are discarded. If there are only conditional branches, they are collated according to equivalent branch conditions. In each group of arcs that share the same guards, any of the members would result in a correct design and we are free to select the most *desirable* of them. A rank function $M = 3*C-10*G+D$ generates a figure of merit for each arc, where G is the number of clauses in the guard expressions, D is the number of differences in state variables and C is the number of data movement operations. A higher value of M loosely denotes an arc that performs more useful work. Hence, for each guarding condition shared by a number of arcs, we retain only the highest ranked behaviour.

The generated converter is a finite-state machine that can be readily output as an RTL or SystemC infinite loop containing a case statement that dispatches over a variable, PC, which ranges over the integer codings of the utilised product states. For a synchronous converter, the loop is made to wait for a clock edge at the start of each iteration by inserting the appropriate target language construct. For medium to large converters, the range of values and hence number of bits in the PC may become excessive even though it is sparsely used, so it must be re-encoded for hardware or SystemC implementation. The general form is illustrated in Figure 5. Of course, we also output the appropriately handed (input, output or local) declarations for the participant nets so the converter can be installed directly as glue logic in a system on chip implementation.

VI. TLM Output Generation

Our second contribution explains how we modify one or more of the interfaces of our net-level converter to be a TLM transactor.

```
RC tlm_target(ARGS)
 {
   args = ARGS;
   callstate = active;
   do { sc_wait(0, SC_NS); } while (callstate != idle);
   return RC;
 }
```

Fig. 7. A stub to make a blocking TLM entry point (unoptimised).

```
while(1)
{
  switch (pc)
    {
     case 10:
         if (callstate == active) { C10cmds; }
         if (g1) ...
         break;

     case 90:
         if (g2) { C90cmds; callstate = idle; pc = 10; }
         break;
   }
}
```

Fig. 8. Typical structure of raw transactor code before thread optimisation.

The raw form from phase 3 consists of one thread that communicates using shared variables for all I/O. We convert certain ports so that they invoke or can invoke TLM-style methods, where the method calls can optionally be conveyed over TLM2.0 convenience sockets. For each of the participants that was a TLM protocol, there is a corresponding call state variable assigned or tested in the machine. For initiator participants the call state will be assigned active by the converter (in one of its '`v=e`' assignments) and tested to see whether it has returned idle (in one of its '`if (g)`' tests). On the other hand, for a target participant, the call state variable is tested for being active and assigned back to idle by the glue machine.

To render the converter as a SystemC initiator style transactor a thread must make the TLM call. Initially, we consider providing a separate thread for this and then explain how the original thread could be used instead, in some cases, as an optimisation.

As shown in Figure 6, the new thread for the initiator executes code consisting of an outer infinite loop that waits for the original thread to set the call active state and then makes the call. On return from the call it clears the active state flag. The code for the new thread is completely boilerplate, except for the name of the TLM method it calls and the arguments passed, which can be configured in a variety of ways (e.g. from IP-XACT) and in our experiments these were taken from the XML interface description of the TLM participant. Note, the zero wait in the unoptimised version could be replaced with a longer wait that would improve efficiency when our subsequent optimisation fails or, better, as kindly pointed out by a reviewer, we could usefully make `callstate` a SystemC event that would then be visible to the scheduller.

Similarly (Figure 7), for a TLM target entry point, we first off let the initiator invoke a boilerplate stub that sets the call active flag and then spinlocks, waiting for it be set idle again by the main thread.

Now we optimise where possible, so that the main thread for a target is eliminated with its work being performed by the initiator's thread when it would otherwise be

```
RC tlm_target(ARGS)
{
  callstate = active;
  pc = 10;
  switch (pc)
    {
     case 10:
         C10cmds;
         if (g1) ...
         break;

     case 90:
         if (g2) { C90cmds; return RC; }
         break;
   }
}
```

Fig. 9. Using a TLM target's thread to execute the converter automaton.

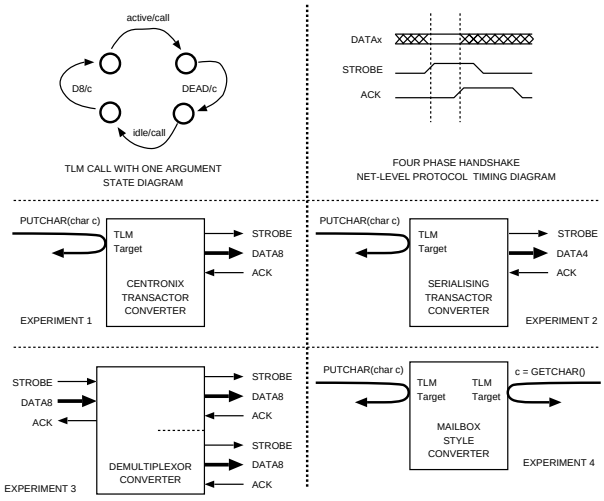

TABLE III
PARTICIPATING PROTOCOLS AND EXPERIMENTAL CONFIGURATIONS.

spinning. Additionally, in some cases, where a transactor is both a TLM initiator and a client, our optimisation may enable all of the internal work as well as the initiating upcall to be performed on the thread provided by the client. Using pattern matching on the abstract syntax tree of the converter code (i.e. before it is emitted as SystemC) we detect where one thread is spinning doing nothing, while another thread is working and the other thread needs do nothing when the first thread is not spinning. Figure 8 can be optimised to become Figure 9 where the pattern matching has detected that only one state waits for the callstate to be active (state 10) and that there is only one state that sets it back to idle (state 90), and that the idle setting state transfers control only back to the active waiting state.

Where multiple TLM ports exist, these peephole operations can be applied in some order, but one might preclude another.

Although our code fragments show the optimisations for the blocking style of transactional modelling, generating the non-blocking style follows a similar pattern, with `false` being returned instead of spinning at a target and making repeated calls until success for an initiator.

VII. EXPERIMENTAL RESULTS

Experiments with protocols and interfaces are summarised in illustrated in Table III.

Exp	Participants concrete states no.	Product no. states explored	Converter no. states live paths	SystemC no. lines
1	$4 \times 6 = 24$	72	71	1070
2	$4 \times 6 = 24$	123	123	1848
3	$6 \times 6 \times 4 = 144$	575	575	14198
4	$4 \times 4 \times 2 = 32$	325	324	7534

TABLE IV
EXPERIMENTAL RESULTS

Experiment 1 took the product of a TLM client with signature `putchar(char c)` with a net-level output port using the centronics-style four-phase handshake.

Experiment 2 was a serialising version of experiment 1, where the net-level port was only four bits wide and hence two transfers are needed for each TLM call.

Experiment 3 was a net-level demultiplexor, where a four-phase input port connected to a pair of four-phase output ports and traffic was routed according to a user predicate that examined the least significant bit of the data.

Experiment 4 was a mailbox component with two blocking TLM entry points, one for writing a character and the other for reading back. This experiment requires an additional resource, a holding register, but an additional state bit resource to record whether the holding register is live or dead was not needed, since such a bit is intrinsic to the encoding of symbolic variables that take on two symbolic values.

The columns of the results table (Table IV) show the number of concrete states of each participant, the number of states explored, which is larger owing to symbolic values, the number of states retained owing to being on live paths that loop through idle states and the number of lines of SystemC generated. The results indicate that no deadlocking paths were deleted in phase 2 for these participants considered. The relatively large output files arise owing to all possible interleavings of external events being explicitly represented.

The results table presented at the conference will include lines for the three components of Figure 1 and perhaps report on some real-world tests.

VIII. CONCLUSION

We have extended the product technique for glue logic generation so that it builds data paths for run-time data values, including multiplexors, filters, serialisers and deserialisers. (The filter is a demultiplexor with an internal port that just invokes **kill**(α) on the data it receives.) We have demonstrated automatic synthesis of TLM modelling components within the same framework.

Future work is to perfect input from IP-XACT and integrate our tool as an Eclipse '*tightly-coupled generator*'. This would enable, for instance, the interface net names and the numerical constants in the address decoder predicates to be sourced from other generators during SoC compilation. Also, optimisations to reduce the output complexity are required. These can be based on any technique that combines converter states that are observably equivalent.

Another requirement in practice is some form of 'branding' because currently there is nothing to stop the glue from crossing over the address and data busses in a write operation where each have the same width.

Instead of implementing the thread optimisations as pattern matching peepholes on the AST for the converter, they might better be implemented by compiling the converter to an assembly-like language and inserting the relevant transfers of control (entry labels, subroutines calls and return statements) in the assembly code. Additionally, the phase 3 heuristic that selects between the suitable converter machines can be enhanced to make these optimisations more readily applicable. For instance, on a TLM server, the call may go idle at various points without altering the correct behaviour, but if it goes idle at an early point, while there is still 'work to do' then the optimisation cannot be applied.

A useful improvement to the congruence algorithm that could assist with hardware timing closure would be to prohibit certain data paths. For instance, the solution to Experiment 4 transfers data directly between the input and outputs in the case that both are active at once, whereas a simpler design with shorter critical path would always pass the data through the holding register.

REFERENCES

[1] M. Burton, J. Aldis, R. Günzel, and W. Klingauf, "Transaction level modelling: A reflection on what tlm is and how tlms may be classified," in *FDL*, 2007, pp. 92–97.

[2] J. Akella and K. L. McMillan, "Synthesizing converters between finite state protocols," in *ICCD '91: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer Processors*. Washington, DC, USA: IEEE Computer Society, 1991, pp. 410–413.

[3] A. L. Sangiovanni-Vincentelli, T. A. Henzinger, L. de Alfaro, and R. Passerone, "Convertibility verification and converter synthesis: two faces of the same coin," *iccad*, vol. 00, pp. 132–139, 2002.

[4] D. Greaves, "Automated hardware synthesis from formal specification using sat solvers," *RSP*, vol. 00, pp. 15–20, 2004.

[5] D. D. Gajski and L. Ramachandran, "Introduction to high-level synthesis," *IEEE Des. Test*, vol. 11, no. 4, pp. 44–54, 1994.

[6] D. Borrione, J. Dushina, and L. Pierre, "Formalization of finite state machines with data path for the verification of high-level synthesis," in *SBCCI '98: Proceedings of the 11th Brazilian Symposium on Integrated circuit design*. Washington, DC, USA: IEEE Computer Society, 1998, p. 99.

[7] V. D'silva, A. Sowmya, S. Parameswaran, and S. Ramesh, "A formal approach to interface synthesis for system-on-chip design," University of New South Wales, Sydney, Australia, Tech. Rep. UNSW-CSE-TR-304, 2003. [Online]. Available: citeseer.ist.psu.edu/582933.html

[8] "Synthesis and optimization of interface hardware between ip's operating at different clock frequencies," in *ICCD '00: Proceedings of the 2000 IEEE International Conference on Computer Design*. Washington, DC, USA: IEEE Computer Society, 2000, p. 519.

[9] S. C. (www.spiritconsortium.org), "IP-XACT version 2.0," 2006. [Online]. Available: www.spiritconsortium.org