Conversing at Many Layers: Multi-layer System-On-chip Protocol Conversion

Roopak Sinha
School of Computer & Mathematical Sciences
Auckland University of Technology
Auckland, New Zealand
roopak.sinha@aut.ac.nz

***Abstract*—The numerous intellectual property blocks of a system-on-a-chip must be integrated so that they can meet system-specific requirements. However, such integration is not guaranteed due to mismatches between IP protocols. Protocol conversion algorithms can generate converters that can guarantee correct system behaviour, but the implementation of converters on-chip remains an open question. IPs can be modelled at several layers of the Open Systems Interconnection (OSI) model. Current protocol conversion algorithms either focus on a single layer or worse, blur the boundaries between these layers. We propose a formal framework that allows generating implementable converters for IP protocols modelled at different OSI layers using any existing converter generation algorithm. We apply the framework to an existing conversion algorithm and discuss how it can be as readily used with other algorithms.**

I. Introduction

System-on-a-chip (SoC) systems contain multiple interacting intellectual property blocks or IPs [1]. IPs interact at the seven layers of the OSI standard [2]. Each layer offers a more abstract set of communication *constructs*, which are sequences of constructs of the layer below it. Sometimes, IPs may suffer from *protocol mismatches* and fail to communicate and/or meet system-level requirements. We can manually modify IPs, or use layer-specific IPs like *bridges* at the data-link layer or *transducers* at the network layer [3]. Formal *protocol conversion* algorithms like [4]–[7] can automatically generate IPs called *converters* that guarantee that the system will satisfy implicit or user-specified requirements. Unfortunately, existing algorithms either target single OSI layers [5], or do not provide a ready path to implementing converters [4], [6], [7].

We focus on the problem of implementing converters at desired OSI layers when IP protocols and requirements are specified at different OSI layers. We assume that physical-layer connections are sound. We also constrain ourselves to *synchronous* on-chip interconnects like AMBA [8] which are the current de facto standards for SoCs.

We propose an elegant protocol conversion framework for IP protocols and requirements modelled at any OSI layer (Sec. III). Cross-layer models are *normalized* to a common target layer, enabled by a simple layer-based expansion of constructs. Normalized protocols and requirements are then processed by a chosen converter generation algorithm to generate a converter, if one exists. The generated converter can be implemented at the identified target layer or at a lower target layer. As a case study, we use the algorithm proposed in [7] as it can handle both control and data mismatches. We consider a video SoC (presented in Sec. II) case study and report experimental results in Sec. IV. Concluding remarks appear in Sec. V.

II. Video SoC Case Study

Fig. 1 shows an abstracted video SoC based on the AMBA AXI-4 interconnect. The SoC reads a composite A/V stream, splits it into video and audio streams via a splitter, and then processes the streams to produce progressive video- and audio-out streams. An ARM processor executes the main control program. We focus only on the processor and deinterlacer IPs in this paper, without losing the generality of the framework.

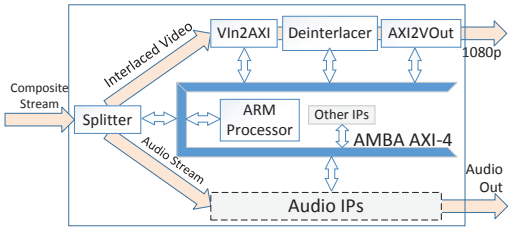
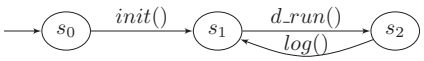


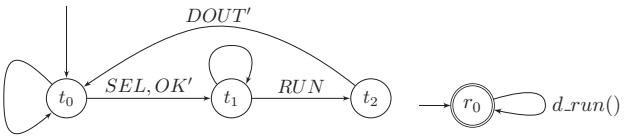
Fig. 1. Video SoC, an illustrative example

IP protocols are modelled using *labelled transition systems*, similar to existing protocol conversion approaches [3]–[5], [7].

Definition 1 (LTS): A LTS is a tuple $\langle Q, q^0, A, R\rangle$ where Q is a set of *states*, $q^0 \in Q$ is the *initial state*, A is a set of actions, and $R \subseteq Q \times 2^A \times Q$ is a *total* transition relation (all states have outgoing transitions). The language $\mathcal{L}$ is the set of all infinite words over A generated from state q^0.



(a) Processor protocol $proc$, application layer



(b) Deinterlacer model $deint$, data-link layer (c) Requirement req

Fig. 2. LTS models of IP protocols and requirements

The protocol $proc$ in Fig. 2(a) models the control flow of the program executing on the processor at the application layer. From initial state s_0 a transition to s_1 involves the action $init()$ signifying an initialization function. We write transition using the short-hand $s_0 \xrightarrow{init()} s_1$. States s_1 and s_2 have transitions to each other to signal the deinterlacer to run ($d\_run()$) and then to log data ($log()$). The data-link

level deinterlacer protocol $deint$ in Fig. 2(b) comes from the Xilinx LogiCORE IP Video Deinterlacer datasheet. Wait and error states are abstracted out. From initial state t_0, a transition to state t_1 involves the reception of slave-select signal SEL and emission of acknowledgement OK' (outputs are primed, e.g. OK'). Next, $deint$ awaits input signal RUN and moves to state t_2, and finally resets back to state t_0.

Existing protocol conversion algorithms expect user-provided formalized *requirements* to specify how IP protocols must interact. We use the requirements model of [7] that extends LTS to have marked states. Fig. 2(c) shows a single-state requirement req for the video SoC. The initial state r_0 is also a *marked* state. The language of req is the set of all infinite words that visit this marked state infinitely often. A system satisfies req if its *projected* language (with all non-relevant actions and transitions w.r.t. the requirement removed), is a subset of the language of the requirement. There are several issues with enforcing req on a system containing $proc$ and $deint$. The protocols are modelled at different layers, and a higher-layer action like $d\_run()$ would mean numerous data-link level transfers. Hence, we cannot *compose* the protocols to model their combined behaviour or reason about timing. Also, req constrains only the application layer behaviour and does not relate to the data-link layer.

III. Layer-based Protocol Conversion

Fig. 3 shows the stages in the proposed conversion process. Firstly, protocols and requirements modelled at different layers are *normalized* to a single target layer. Next, the normalized models are read by a conversion algorithm to generate a converter. The converter is then implemented using an implementation algorithm suitable for the target layer.

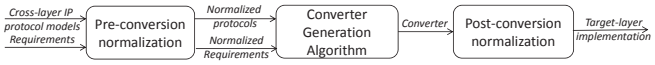


Fig. 3. Overall conversion process

A. Modelling layer-level expansions

The OSI stack contains seven layers. The lower-most physical layer involves driving and sampling pin and wire level voltages. The data-link layer uses the pin and wire constructs provided by the physical layer and implements key issues such as timing, arbitration, and synchronization. It provides logical links to be used by the network layer above it. The network layer implements routing of packets. The transport layer then enables streams of untyped messages by implementing packeting, flow control and error correction over the end-to-end packet streams of the network layer. The session layer provides the transmission of end-to-end untyped but named messages and carries out synchronization. The presentation layer provides end-to-end typed and named messages by using the constructs of the session layer. Finally, the application layer implements algorithms that use the presentation layer to activate and transfer data between IPs. A construct of a higher layer sequences constructs of a lower layer. We can capture this relationship as follows.

Definition 2 (Layer-based expansion): Given actions set A, the function $Deconstruct$ reads an action $a \in A$ and a target-layer $l \in [2, 7]$ and returns $null$ (no expansion available) or an **LTS** ***fragment*** $LTS_F \quad \langle Q_F, q_F^0, (A \backslash \{a\}), R_F, q_F^e, A_{In} \rangle$ where Q and q^0 are as per Def. 1, R is total except for the special *end*-state q^e that has no outgoing transitions. Finally, $A_{In} \subseteq A \backslash \{a\}$ is the set of *entry-actions* of LTS_F.

The $Deconstruct$ function decomposes a single action at a higher layer to a sequence of constructs (actions) of the next lower layer. Fig. 4 shows the fragment returned by $Deconstruct(d\_run(), 6)$.

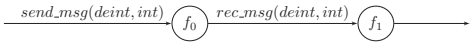


Fig. 4. Expansion of $d\_run()$ to the presentation-layer

A fragment may contain cycles; If we expand the presentation-layer construct $send\_msg(deint, int)$ to the transport layer, error-control strategies might introduce intermediate states that have transitions back to previously visited states in the fragment when messages require re-sending.

A key question in building the $Deconstruct$ function, essentially a look-up table, is, "Where does the expansion information come from?". There is no simple answer. We may find such information coded within software or network design layers (layers 3 to 7), or within timing diagrams and data-sheets (for layers 2-4). While it is desirable to extract this information automatically, this problem is beyond the scope of this paper. In many cases though, manual specification of such expansion information should be simpler than a complete manual normalization of a protocol. The testing of this hypothesis is an important future work. The restricted number of constructs at each layer helps ensure that the $Deconstruct$ function does not require too much space, however the amount of space will depend on the level of detail captured at each level, which in turn depends on the choice of conversion technique used.

B. Normalizing heterogeneous protocols

Algorithm 1: Algorithm `NormalizeToLayer`

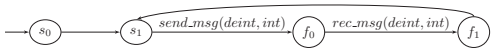
Input: LTS $\langle Q, q^0, A, R \rangle$, **integer** cl, **integer** tl
Output: Normalized LTS $\langle Q_1, q_1^0, A_1, R_1 \rangle$ (tl)

1 Make copies Q_1, q_1^0, R_1 of Q, q^0, R; $cl = cl - 1$;
2 **while** $tl < cl$ **do**
3 **foreach** $q \xrightarrow{\{a,\ldots\}} q' \in R_1$ *s.t.* $Deconstruct(a, cl) \neq null$ **do**
4 Remove transition $q \xrightarrow{\{a,\ldots\}} q'$ from R_1;
5 $LTS_F \ \langle Q_F, q_F^0, (A \backslash \{a\}), R_F, q_F^e, A_{In} \rangle$ = $Deconstruct(a, cl)$;
6 Add copies of states in Q_F to Q_1, add transitions from R_F to R_1;
7 Add transition $q \xrightarrow{\{a,\ldots\} \cap \{a\} \cup A_{In}} q_F^0$ to R_1;
8 **foreach** $q' \xrightarrow{\{a_1,\ldots,a_n\}} q'' \in R_1$ **do**
9 Remove transition $q' \xrightarrow{\{a_1,\ldots,a_n\}} q''$ from R_1;
10 Add transition $q_F^e \xrightarrow{\{a_1,\ldots,a_n\}} q''$ to R_1;
11 **end**
12 **end**
13 $cl = cl - 1$;
14 **end**
15 **return** $\langle Q_1, q_1^0, A_1, R_1 \rangle$;

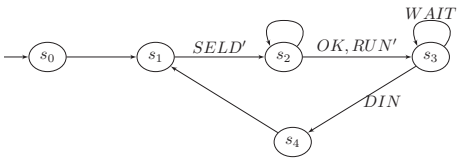
Layer-based expansion allows transforming higher-layer protocols and requirements to a lower layer. For the Video SoC, we must *normalize* the protocol $proc$ and requirement req to the data-link layer at which the deinterlacer protocol is modelled. We introduce the algorithm `NormalizeToLayer` for this purpose. Its inputs are a LTS, and integers cl and tl

indicating the current and target layers respectively. Values $1 \ldots 7$ for these variables refer to the physical $\ldots$ application OSI layers. The LTS is iteratively normalized layer-by-layer (lines 2–14). For each layer, the algorithm replaces a transition in the LTS with an equivalent LTS fragment at the next lower layer obtained using the call $Deconstruct(a, cl)$ (lines 5–11). Fig. 5(a) shows how the transition $s_2 \xrightarrow{\{d\_run()\}} s_3$ of the processor protocol shown in Fig. 2(a) is replaced by the LTS fragment shown in Fig. 4.

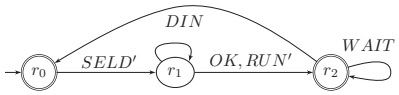
Fig. 5(b) presents the processor model normalized to the data-link layer. At this level, each transition in the model synchronizes with the bus clock, like the deinterlacer model in Fig. 2(b). We have renumbered the states of the normalized protocol as $s_0 \ldots s_4$, and have removed any unreachable states after normalizing. Note that Alg. 1 does not explicitly remove any state from a LTS. However, most protocol conversion techniques construct converters by traversing only reachable states, and hence the removal of unreachable states can be seen as a secondary activity. During the normalization of the processor protocol, we assume that local application-layer actions of the processor, such as $init()$ and $log()$ from Fig. 2(a), take only one bus clock because processors tend to be many times faster than buses. This assumption, and the fact that the processor does not use the bus to carry local operations, lead to the outgoing transitions from s_0 and s_4 to be unlabelled. Operations, like $d\_run()$ use the bus and are progressively expanded into LTS fragments. The fragment consisting of states s_1, s_2 and s_3 and their outgoing transitions are obtained after expanding the action $d\_run()$.



(a) Expanding $proc$ to the presentation layer



(b) `NormalizeToLayer`$(proc, 7, 2)$



(c) `NormalizeToLayer`$(req, 7, 2)$

Fig. 5. Normalizing protocols and requirements

Fig. 5(c) shows how the requirement shown in Fig. 2(c) is normalized. The application-layer construct $d\_run()$ is expanded to a sequence of data-link layer constructs. This sequence is identical to the boxed sequence shown in Fig. 5(b). While processing requirements, Alg. 1 needs to be slightly extended to correctly handle marked or acceptance states. In Alg. 1, the transition from state $q \xrightarrow{\cdots} q'$ is removed and replaced by an LTS fragment LTS_F (lines 4–11). If q' is a marked state, the algorithm identifies the end-state q_F^e of the fragment (lines 5, 10) as a marked state.

While we only propose a down-stream normalization, it is possible to have an up-stream normalization `NormalizeToLayer`$^{-1}$. However, this is difficult because protocols modelled at lower levels may be optimised such that construct sequences may not relate to any single construct on an upper layer. Our framework also allows us to directly convert a protocol from a higher layer to a lower layer without going through intermediate layers, saving time if a lower target layer has been chosen. A normalized LTS does not have any standard correspondence to the original, higher layer LTS it is obtained from. The lack of strong bi-simulation or simulation relationships can be overcome by using projections, and by ensuring that the function $Deconstruct$ accurately models the inter-layer relationships between constructs. We can show that in the presence of consistent down-stream and up-stream normalization, protocols at different layers are consistent.

Theorem 1: Given a labelled transition system LTS $\langle Q, q^0, A, R\rangle$, layer-based normalization function `NormalizeToLayer` and a consistent inverse normalization algorithm `InverseNormalize`, then `InverseNormalize`(`NormalizeToLayer`(LTS, cl, tl)) is **strongly bi-similar** to LTS.

The translation of upper-layer constructs into sequences of lower-layer constructs results in a progressively larger model. However, we can control this growth in state space by abstraction. E.g., data values can be replaced by control signals. Similarly, special address bus values such as for decoding slave-select signals can be modelled as control signals too.

C. Conversion and Implementation

The conversion stage applies the chosen protocol conversion algorithm to the normalized protocols and requirements. A conversion algorithm might require additional meta-data. E.g., the protocol conversion technique presented in [7] requires a classification of control signals, signal mappings, and data linkages. The proposed generic approach allows user-provided meta-data to be included and passed to the conversion algorithm along with protocol and requirement models. If required, the generated converter can be further normalized to a lower target-layer using Alg. 1. Finally, the converter can then be integrated into the system by implementing it at the target layer using an appropriate implementation algorithm, such as the one presented in [2].

IV. Results

We have created a Java-based tool that allows users to specify layer-based expansions by relating a layer-specific construct to a sequence of constructs at the next lower layer. Our tool implements the `NormalizeToLayer` and layer-based conversion algorithms to normalize multi-layer IP and requirements and then carry out converter generation. Unlike existing approaches, our tool always generates converters that are specific to the target layer. The generated converter can be further normalized to a lower layer if required. We assume that the use of soft-IP cores allows us to exclude modelling the physical layer constructs. Hence, our tool can generate converters until the data-link layer.

Once a layer-specific converter has been obtained, we can use a layer-specific implementation technique to synthesize a converter implementation. Tab. I shows the implementation options for the various layers [2]. As noted earlier, we do not consider the physical layer due to the use of soft IPs and automatic interconnect generation supported by SoC/NoC tools. Note that for layers 3–7, protocols are asynchronously

composed, and we may extend existing asynchronous conversion or transducer generation techniques like [3] for these layers. Our chosen technique [7] can work directly with synchronous compositions at layers 2 and 3. We mitigate this restriction by using *wait*-states or states with self-loops when expanding models at layers 3–7. The wait states may progress to other states only when specific actions happen, allowing us to de-synchronize protocol execution and yet use synchronous composition supported by the chosen approach [7].

TABLE I
LAYER-BASED IMPLEMENTATION

Layer	Composition	Translated into	Implementation
Application	Asynchronous	send and receive typed and named messages	Application code
Presentation	Asynchronous	Type-dependent unpacking of messages	Code/system calls
Session	Asynchronous	multiplex and synchronize streams	OS kernel
Transport	Asynchronous	unpacking streams to packets	OS kernel
Network	both	transfer packets over logical connections	OS kernel, driver
Data-link	Synchronous	synchronization, multiplexing, arbitration, timing, etc.	Driver, HAL hardware
Physical	–	–	–

Fig. 6 shows how the size of video SoC increases as the processor protocol's size grows from three states at the application layer to 40 states at the data-link layer. We report two system sizes. $Ex1$ refers to a system comprising the processor and the deinterlacer. $Ex2$ refers to an extended system that also contains a stream splitter and an abstracted audio converter, modelled at the data-link layer based on Xilinx data-sheets. System size (the number of states in the composition of all constituent protocols) grows proportionally to the size of the processor protocol as other protocol sizes are kept constant, and directly affects protocol conversion times.

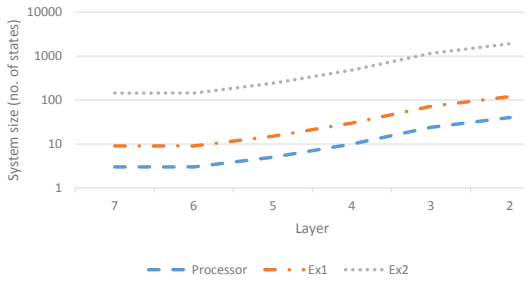


Fig. 6. Layer-based system sizes

V. CONCLUSIONS

We present a generic framework that allows converter generation techniques to work with IP implementation methods. We normalize multi-layer models of IP protocols and system-level requirements, allowing the generation of layer-specific converters that can be implemented using layer-specific methods. The normalization depends on user-provided expansions of layer-specific constructs into sequences of constructs at lower layers. We show how our technique works with a recent protocol conversion technique and discuss important issues such as choosing the right target layer, extending to other approaches, and automating layer-based expansions. Experimental results show that our approach can help translate converter logic into implementable converters.

Cross-layer modelling of IP protocols is common as IPs are customised at different levels. Converter generation techniques can generate "correct" converter logic, but overlook timing and relationships between cross-layer signals. Normalized protocols and converters as proposed in this paper can be implemented in hardware (layers 2, 3) or software (layers 3–7). We can utilize the implementation techniques presented in [2], [9], [10] to implement IPs. Our approach can work with other protocol conversion approaches, as well as other related approaches for generating bridging logic [3], [11], [12]. All existing approaches use finite state machines to model IP protocols and can therefore readily use layer-level expansions and normalization. Additional information needed by a specific technique can be modelled as additional meta-data. Capturing requirements is more tricky; Some approaches have implicit requirements [5], [12] while others support models in temporal logic [6] or different finite-state machine variants [4]. Hence, normalizing requirements might require additional formalization. We can avoid this problem by normalizing protocols to a specific target layer and then writing requirements on these.

Future directions include looking at distributed layer-based converter generation and creating a robust tool-chain to provide end-to-end on-chip protocol conversion.

REFERENCES

[1] L. Benini and G. De Micheli, "Networks on chip: a new paradigm for systems on chip design," in *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*. IEEE, 2002, pp. 418–419.

[2] A. Gerstlauer, D. Shin, J. Peng, R. Domer, and D. D. Gajski, "Automatic layer-based generation of system-on-chip bus communication models," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 9, pp. 1676–1687, 2007.

[3] M. Fujita, H. Tanida, F. Gao, T. Nishihara, and T. Matsumoto, "Synthesis and formal verification of on-chip protocol transducers through decomposed specification," in *Quality Electronic Design (ISQED), 2010 11th International Symposium on*. IEEE, 2010, pp. 515–523.

[4] R. Passerone, L. de Alfaro, T. Henzinger, and A. Sangiovanni-Vincentelli, "Convertibility verification and converter synthesis: Two faces of the same coin," in *International Conference on Computer Aided Design ICCAD*, 2002.

[5] K. Avnit, V. D'silva, A. Sowmya, S. Ramesh, and S. Parameswaran, "Provably correct on-chip communication: A formal approach to automatic protocol converter synthesis," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14, no. 2, p. 19, 2009.

[6] R. Sinha, P. S. Roop, Z. Salcic, and S. Basu, "Correct-by-construction multi-component soc design," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2012, pp. 647–652.

[7] R. Sinha, A. Girault, G. Goessler, and P. S. Roop, "Formal system-on-chip design using incremental converter synthesis," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14.

[8] ARM, "AMBA 4 Specifications," www.arm.com, 2011.

[9] D. Shin, A. Gerstlauer, J. Peng, R. Dömer, and D. D. Gajski, "Automatic generation of transaction level models for rapid design space exploration," in *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*. ACM, 2006, pp. 64–69.

[10] D. Shin, A. Gerstlauer, R. Dömer, and D. D. Gajski, "Automatic network generation for system-on-chip communication design," in *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2005, pp. 255–260.

[11] S. Yoo, G. Nicolescu, D. Lyonnard, A. Baghdadi, and A. A. Jerraya, "A generic wrapper architecture for multi-processor soc cosimulation and design," in *Proceedings of the ninth international symposium on Hardware/software codesign*. ACM, 2001, pp. 195–200.

[12] M. Tivoli, P. Fradet, A. Girault, and G. Goessler, "Adaptor synthesis for real-time components," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2007, pp. 185–200.