Correct-by-Construction Multi-Component SoC Design

Roopak Sinha
POP ART Team
INRIA Rhône Alpes
France
roopak.sinha@inria.fr

Partha S Roop, Zoran Salcic
Electrical and Computer Engineering
University of Auckland
Auckland, New Zealand
p.roop, z.salcic@auckland.ac.nz

Samik Basu
Department of Computer Science
Iowa State University
Ames, IA
sbasu@cs.iastate.edu

***Abstract*—Systems-on-chip (SoCs) contain multiple interconnected and interacting components. In this paper, we present a compositional approach for the integration of multiple components with a wide range of protocol mismatches into a single SoC. We show how SoC construction can be done in single-step when all components are integrated at once or it can also be performed incrementally by adding components to an already integrated design. Using a number of AMBA IPs, we show that the proposed framework is able to perform protocol conversion in many cases where existing approaches fail.**

I. Introduction

A system-on-a-chip (SoC) contains multiple intellectual property blocks (IPs) composed together to achieve system-level functionality. It is desirable that IPs are integrated into a SoC automatically because manual integration is a time consuming and error prone process for even small SoCs.

A few techniques for compositional reasoning for multi-component systems have been proposed [1], [2], [3]. In Alfaro et al [1], *interface automata* are introduced as a framework to capture the temporal aspects of software component interfaces. Here, two components are deemed compatible if an environment under which they can work together exists. This framework is extended by Tripakis et al [2] to *relational interfaces* where input-output relations can be captured. Bozga et al [3] provide a modelling framework for the compositional nature of synchronous multi-component systems. Although SoCs are compositional systems, existing compositional frameworks are not directly applicable to their construction. Generally, IPs cannot be composed without the resolution of protocol mismatches, which may lead to undesirable composite behaviour such as deadlocks. Existing compositional techniques can check whether two or more IPs are *compatible* [1], but cannot *enforce* compatibility.

Formal approaches have been used in generation of protocol *converters* which resolve incompatibilities between IPs [4], [5], [6], [7]. Avnit et al [4] present a framework for the precise modelling and conversion of AMBA bus protocols. Passerone et al [6] present a game-theoretic formulation for protocol conversion between two protocols. Kumar et al [5] present a technique to bridge mismatches using supervisory control theory, while Tivoli et al [7] present adaptor synthesis, where an adaptor is generated to control a multi-component real-time system such that given timing as well as functional requirements (expressed as automata) are met. However, these approaches are also inapplicable to SoC construction. Some (like [4]) generate converters for only two IPs, and only address control mismatches. Existing approaches cannot address data-width mismatches (caused by different word sizes), nor clock mismatches (that occur when different clocks drive the IPs). In addition, no existing protocol conversion approach can generate converters to ensure the satisfaction of multiple constraints (specifications) on the behaviour of the SoC.

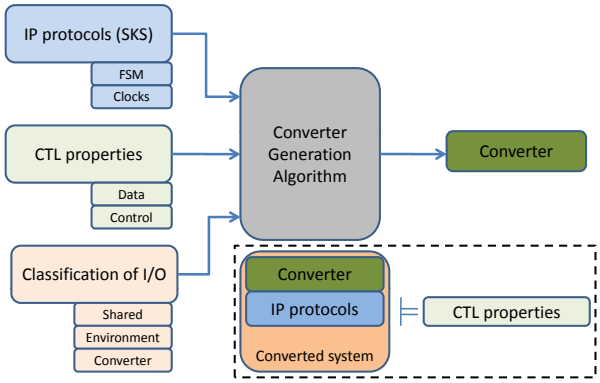
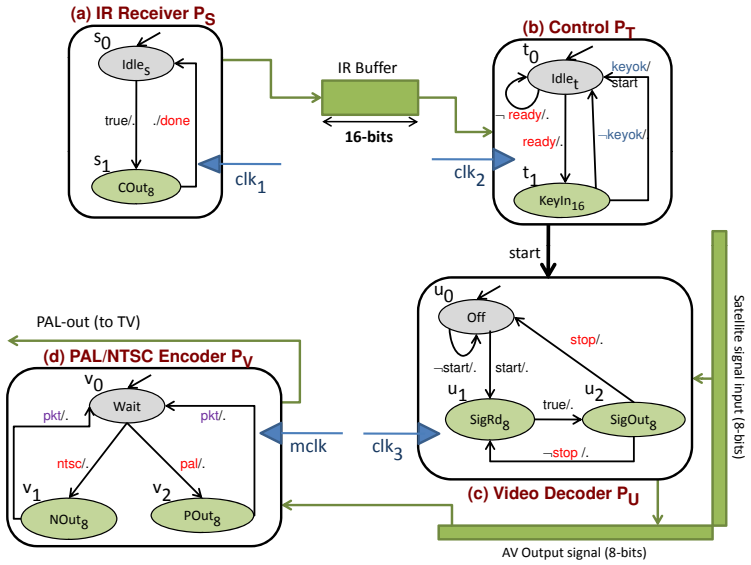


Fig. 1: Overview of the conversion process

This paper presents a SoC construction approach, which extends the work presented in [8], to allow formal correct-by-construction design of SoCs with mismatching IPs. An overview of this approach is provided in Fig. 1. It addresses clock, data-width and control signal mismatches, and can integrate more than 2 protocols at once, making it more general than existing protocol conversion techniques. IP protocols are represented as synchronous, finite-state automata called *Synchronous Kripke Structures* (SKS) [9], that may operate on different phase-aligned clocks. The user provides control *and/or* data constraints using the temporal logic `CTL`, and identifies each control signal as *shared* (emitted and read within the IPs), *environment*-generated (generated by the controllable environment), or *converter-generated* (generated by the converter whenever needed). A tableau-construction based algorithm is then employed to generate a converter, if possible, that ensures the satisfaction of given constraints regardless of how the environment behaves.

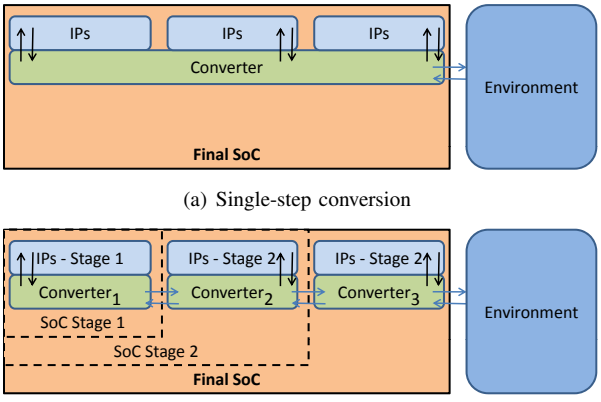
As compared to [8], this paper provides a simpler formalism (in Sections II and III), more specifically in the categorisation of control signals for conversion (Section III-B). We also show that SoCs can be constructed in two ways in Section III-D. *Single-step construction* (Fig. 2(a)) combines all IPs of a SoC

978-3-9810801-8-6/DATE12/©2012 EDAA



(a) Protocols of the set-top receiver box SoC (b) Clock automaton

Fig. 3: Conversion techniques



(a) Single-step conversion

(b) Incremental conversion

Fig. 2: Conversion techniques

using a centralized converter. On the other hand, *incremental conversion* (Fig. 2(b)) allows adding IPs to a SoC in stages. At each stage, the system is partially *closed* with respect to shared I/O (input/output) between known IPs, but is *open* with respect to I/O from its environment, allowing more IPs to be added later on. Another major contribution of this paper is that the extended algorithm generates a *maximally-permissive* non-deterministic converter. All possible *deterministic* converters are restrictions of this converter. We show that this feature is essential for incremental conversion, where IPs are added to an SoC in stages. This feature is absent in [8], which affects its applicability for incremental design. We also test the formulation using larger SoC benchmarks in Section IV.

II. Motivating Example

Protocols are described as Synchronous Kripke Structures (SKS), which is a *finite state machine* represented as a tuple $\langle AP,\ S,\ s_0,\ I,\ O,\ R,\ L,\ clk\rangle$ where, $AP = AP_{control} \uplus AP_{data}$ is a set of propositions where $AP_{control}$ is the set of control labels and AP_{data} is the set of data labels. S is a finite set of states, with s_0 as the initial state. I and O are finite, non-empty set of inputs and outputs respectively. $R \subseteq S \times \{\mathtt{t}\} \times B(I) \times 2^O \times S$ is the transition relation where $B(I)$ represents the set of all boolean formulas over I. A transition triggers when the corresponding boolean formula over its inputs evaluates to true. This evaluation is done at every rising edge (tick) of the clock clk. If the transition is taken, the corresponding set of outputs may be generated. $L : S \rightarrow 2^{AP}$ labels each state by a set of atomic propositions describing the control and data I/O status in that state.

Fig. 3(a) presents the IPs of a set-top receiver box SoC. The IR receiver IP P_S writes 8-bit data (represented by the label $COut_8$ of state s_1) onto an IR buffer. The control IP P_T awaits an input `ready` and then reads 16-bit data from the IR buffer (represented by the label $KeyIn_{16}$ of state t_1). It emits `start` if the key is valid, which is read by the video decoder P_U. P_U then alternates between reading 8-bit data from the satellite signal input stream (in state u_1) and writing it to the AV output signal stream (in state u_2) until it receives the `stop` signal. The PAL/NTSC encoder P_V can convert an AV packet stream to a PAL or NTSC stream depending on the corresponding input received before each packet.

The IPs execute on different *phase-aligned* clocks, as shown by the clock automaton CA in Fig. 3(b), that describes the relationships between clock ticks. It is driven by the fastest clock $mclk$ in the system, which synchronizes with all other clocks. Clocks clk_1 and clk_2 have the same period, but do not synchronize (their rising edges occur at different instances or *ticks* of $mclk$). Clock clk_3 is twice as fast as these clocks but does not synchronize with either.

In order to integrate the above IPs into a SoC, a designer faces the following challenges. Firstly, the SoC contains IPs operating on different clocks, resulting in timing issues and a possibility of losing shared control signals (such as `start`). Also, different I/O signals needs to be treated differently in the final SoC. For example, input signal `keyok` is generated asynchronously in the environment, whereas signals like `stop` are needed for some IPs to progress but are not generated

anywhere in the system. Moreover, while the IR receiver emits 8-bit packets, the control unit reads 16-bit data, resulting in a *data-width* mismatch. Additionally, the SoC must satisfy the following specifications: (a) The control unit must continually check the validity of the IR buffer, and (b) The video decoder must never be disabled before a control unit check. Unlike the technique proposed in this paper, no existing technique for compositional reasoning or protocol conversion can address all of the above issues.

III. Methodology

A. Inputs

The first step in the conversion process is the conditioning of inputs to the conversion algorithm. Essentially, the algorithm has two inputs: protocols and specifications.

1) Protocols: As each SKS may execute on a different clock, each protocol is *oversampled* to describe its behaviour with respect to the fastest clock in the SoC. This step simplifies the problem to the conversion of SKS executing on a common clock. Fig. 4 shows how the IR receiver SKS (Fig. 3(a)) can be oversampled with respect to the fastest clock $mclk$ in the SoC. Note that the oversampled SKS makes a transition that matches a transition in the original SKS only when the clock automaton makes a transition during which the original clock input for the SKS was emitted. In the rest of the paper, we assume all SKSs are oversampled unless otherwise mentioned.

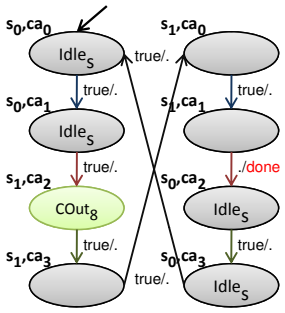


Fig. 4: Oversampled IR receiver $P_{S_{CA}}$

Next, we compute the parallel composition of the SKSs. The parallel composition of protocols P_1 and P_2, denoted by $P_1||P_2$, is computed as a Cartesian product of P_1 and P_2 [9]. The composition operator $||$ is associative and distributive.

The last step in the preparation of protocols is *partitioning of inputs and outputs*. This step is described in Fig. 5. This step explicitly states how different I/O signals are managed by the converter. Each signal is placed in one of the following partitions by the user. Inputs that are read from (or outputs emitted to) the environment are placed in the *uncontrollable signals* partition. The converter must ensure that these signals are passed to/from the protocols as soon as they are available. *Shared signals* are emitted and read by protocols. The converter must ensure that such signals are read from the protocols when they are emitted and buffered until they are ready to be read back by the protocols. *Missing control signals* are protocol inputs that are generated neither by the environment nor within the protocols. The converter might need to generate these signals itself in order to ensure progress.

For example, in Fig. 3(a), the signal `keyok` is uncontrollable as it is read from the operating environment (a remote

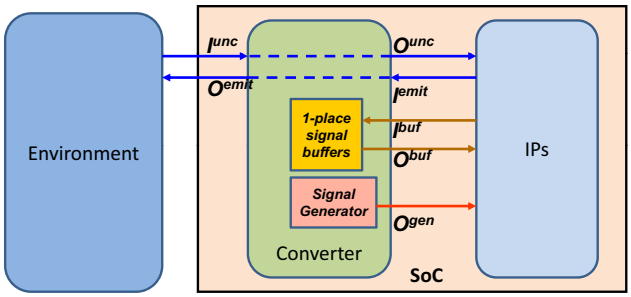


Fig. 5: Different types of inputs and outputs

control device interface). Signal `start` is a shared signal as it is emitted by the control unit P_T and read by the video decoder P_U. Also, `ready` (P_T) and `stop` (P_U) are missing control signals because they are emitted neither by the environment nor by any protocols in the SoC.

2) Specifications: Our conversion algorithm takes as input specifications written in temporal logic `CTL`, that describe the desired communication between the IPs. The various types of specifications admitted by the algorithm are:

- *Control constraints* are `CTL` properties over state labels in the protocols. E.g., the `CTL` properties $\texttt{AGAF}KeyIn_{16}$ and $\texttt{AG}(SigRd_8 \Rightarrow \texttt{A}(\neg Off\ \texttt{U}\ KeyIn_{16}))$ require that the set-top system (in Fig. 3(a)) remains receptive to reading data from the remote control, and the video decoder must never be disabled before a control unit check.
- *Data constraints* are `CTL` properties over *data counters* that are used to keep track of the data communication between IPs. A data counter is associated with a data buffer, and is incremented/decremented whenever data is added to/removed from the buffer. For the set-top box example, we introduce a counter `I` for the IR Buffer (shown in Fig. 3(a)). As the capacity of the IR buffer is 16 bits, we introduce the data-constraint $AG(0 \leq \texttt{I} \leq 16)$ to ensure that the buffer never overflows (or underflows). Note that we do not differentiate between the different (upto 2^{16}) data values in the buffer but the number of bits contained in the buffer at any one time during execution. This allows us to prevent state-space explosion due to data.
- *Control-data constraints* involve the use of state labels as well as counter values within the same formula. E.g., the formula $AG((Idle_s \wedge Idle_t \wedge Off \wedge Wait) \Rightarrow (\texttt{I} = 0))$ requires that whenever all IPs of the set-top box are in their initial states, the IR buffer should be empty.

B. Converters

A converter $\mathcal{C}$ controls a protocol P, which can indeed be the parallel composition $P_1||P_2||\ldots||P_n$ of multiple IP protocols. The converter uses the information about how different I/O signals are to be treated, as shown in Fig. 5, and can perform the following actions. All uncontrollable signals, if read, are *forwarded* to the environment/protocols in the same clock tick. The converter *buffers* shared signals read from P and then can forward them later on (after one or more clock ticks), allowing synchronized communication between two protocols executing on different clocks, or having mismatching protocols. The converter maintains a 1-place buffer for each shared signal. It may also *hide* a buffered signal

from P in a tick to disable an undesirable transition. Finally, it may *artificially generate* some control signals that are emitted neither by the environment nor in P.

The converter executes using the base (fastest) clock of the SoC so that it can manage every transition in P. At each clock tick, the converter follows a precise sequence of interactions (called a *micro-step*) with the environment and the protocols. Firstly, the converter reads the uncontrollable signals present in the environment as input formula b_{unc}. It then forwards the uncontrollable signals as well as any converter-controlled signals (buffered or generated) to the protocols as the set o. It then reads the signals generated by the IPs, emitting any uncontrollable outputs (o_{emit}) to the environment, and buffering relevant control signals (i_{buf}). These steps form a single transition (or *macro-step*) $\mathtt{C} \xrightarrow{b_{unc}/o;o_{emit};i_{buf}} \mathtt{C}'$ of a converter state C.

$\mathcal{C}$ is formally a *converter SKS*, or CSKS. Each state precisely corresponds to a unique state in the protocol being converted. It operates using the fastest clock in the SoC, and each transition in $\mathcal{C}$ triggers a unique corresponding protocol transition. Each state is also associated with a control buffer, indicating the set of signals contained in the converter's buffers. In the initial state, the buffers are empty. After each transition, the buffers are adjusted by adding any newly buffered signals (and removing any signals emitted) during that transition.

The *lock-step composition* [9] describes the converted system, and is defined using the // operator. $\mathcal{C}//P$ is a SKS where each state corresponds to a unique state C in $\mathcal{C}$ and its corresponding state s in the protocols. Each transition corresponds to a transition in C and the matching transition in s. The converted system therefore contains only those transitions and executions in P that are allowed by $\mathcal{C}$.

C. Converter Generation Algorithm

The converter generation algorithm Alg. 1, based on the conversion algorithm presented in [9], is a recursive algorithm that constructs a graph corresponding to all possible *valid* behaviours of the protocol such that given constraints are satisfied. The initial call is made with the arguments $s = s_0$, the initial state of P (the protocol(s) to be converted); $\mathtt{I} = \mathtt{I}_0$ a set of counter valuations where all counters are reset to 0; $\mathtt{FS} = \mathtt{FS}_0$, the set of all CTL constraints; $\mathtt{E} = \emptyset$, an empty set of buffered signals; and $\mathtt{H} = \emptyset$, the set referring to the entries created during previous calls to the algorithm. These initial arguments (except H) correspond to the *root* node, or entry, in the graph. An entry represents the assertion $(s, \mathtt{I}, \mathtt{E}) \models \mathtt{FS}$.

The algorithm first checks if the same entry has been processed earlier (lines 2–4), and returns failure (a FALSE_NODE), in case the ancestor entry contains EU or AU formulas. Otherwise it returns the ancestor node itself. This prevents infinite unrolling, and the *finitization* is based on rules for on-the-fly model checking, also described in [9]. Then, a new entry is processed by simplifying the set of constraints FS by removing one formula (line 7) and either checking its satisfaction (lines 8-9) or by simplifying it into sub-formulas and making recursive calls to Alg. 1 (lines 10-36). The algorithm returns the processed entry, or FALSE_NODE if the unrolling/satisfaction process returns failure.

Alg. 1 extends the conversion algorithm presented in [9] in the processing of disjunctions (lines 13–16) and sets of AX and EX formulas (lines 23–36). In [9], the conversion algorithm looks for *one* way the assertion $(s, \mathtt{I}, \mathtt{E}) \models \mathtt{FS}$ can be satisfied. E.g., it does not process the second operand of a disjunction if the recursive call relating to the first operand returned success. However, Alg. 1 enumerates *all* possible ways in which an assertion is satisfied. Although this results in a larger graph than in [9], the complexity of conversion is unaffected. In the worst case, both algorithms must explore all possible nodes.

Algorithm 1 NODE isConv(s, I, FS, E, H)

```
1: curr = createNode(s, I, FS, E);
2: if anc ∈ H = curr then
3:     return FALSE_NODE if FS contains AU/EU formula, else return anc
4: end if
5: H_1 = H ∪ {curr}; FS_1 = FS;
6: if FS contains a formula F which is neither of type AX nor EX then
7:     FS_1 := FS_1 − F; Node ret:= FALSE_NODE;
8:     if F is a (negated) proposition/counter constraint satisfied in s, I then
9:         ret:= isConv(s, I, FS_1, E, H_1)
10:    else if F = φ ∧ ψ then
11:        ret:= isConv(s, I, FS_1 ∪ {φ, ψ}, E, H_1)
12:    else if F = φ ∨ ψ then
13:        ret:= createORNode(s, I, FS_1 ∪ F, I);
14:        ret.addChild(isConv(s, I, FS_1 ∪ {φ}, E, H_1));
15:        ret.addChild(isConv(s, I, FS_1 ∪ {ψ}, E, H_1));
16:        ret= FALSE_NODE if both its children are FALSE_NODE;
17:    else if F = AGφ, EGφ, A(φ U ψ), ir E(φ U ψ) then
18:        ret:= isConv(s, I, FS_1 ∪ F′, E, H_1) where F′ is the
           conjunctive/disjunctive unrolled formula corresponding to F;
19:    end if
20:    curr.addChild(ret); curr = FALSE_NODE if ret=FALSE_NODE;
21:    return curr;
22: end if
23: curr.type := OR_NODE;
24: FS_AX = {φ | AXφ ∈ FS}, FS_EX = {φ | EXφ ∈ FS}
25: for each conforming subset Succ of the successor set of s do
26:    for Each possible distribution of FS_EX over Succ do
27:        Create child nxt of curr; nxt.type = X_NODE;
28:        for each state s′ in Succ do
29:            E′ = AdjustBuffer(E,s ⟶ s′);I' = AdjustCounters(I,s');
30:            nxt.addChild(isConv(s', I', FS_AX ∪ FS_EX', E', H_1));
31:        end for
32:        nxt = FALSE_NODE if any of its child nodes is FALSE_NODE;
33:    end for
34: end for
35: curr = FALSE_NODE if all of its child nodes is FALSE_NODE;
36: return curr;
```

1) Soundness, Completeness, and Maximality: We prove the correctness of the conversion algorithm in [9]. Specifically, given a protocol P (or a parallel composition of protocols), a set FS of CTL properties, and the identification of all I/O signals as shared, generated or uncontrollable, a converter $\mathcal{C}$ to achieve the satisfaction of all formulas FS in a converted system $\mathcal{C}//P$ exists *iff* the $\mathtt{isConv}(s_0, \mathtt{I}_0, \mathtt{FS}, \emptyset, \emptyset)$ returns a non-FALSE_NODE. The extended version of the algorithm presented in this article, which has the added functionality of enumerating all possible ways of satisfying given constraints, can also be observed to be sound and complete. In addition to the results of soundness and completeness, we also prove that the graph returned by Alg. 1 is equivalent to a protocol resulting from the lock-step composition of the given protocol(s) with the *maximally-permissive, non deterministic* converter. Using the soundness and completeness results and showing

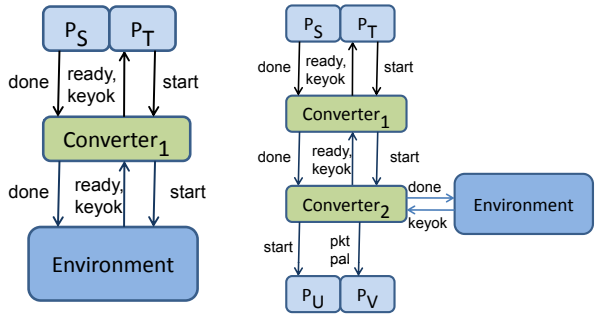
A deterministic converter can be created by first extracting a sub-graph from the graph constructed by Alg. 1 such that only one non-FALSE_NODE child of every OR_NODE (lines 13, 23) is retained. This sub-graph can then be interpreted as a converter by the extraction algorithm presented in [9].

that the enumeration of all possible ways in which disjunctions and AX/EX formulas can be satisfied ensures maximality.

D. Approaches to SoC design

1) Single-step conversion: Single-step conversion involves the generation of a single converter to guide all protocols (as shown in Fig. 2(a)). We illustrate how this technique works by using the set-top box example shown in Fig. 3(a). Firstly, all SKS P_S, P_T, P_U and P_T are oversampled by using the common clock automaton (Fig. 3(b)), and their parallel composition $P_S||P_T||P_U||P_V$ (after oversampling) is computed. Next, a call to Alg. 1 is made by passing the initial state of the parallel composition, and all CTL constraints provided by the user. For the set-top box example, the control constraints AGAF$KeyIn_{16}$ and AG($SigRd_8 \Rightarrow$ A($\neg Off$ U $KeyIn_{16}$)), and the data constraint $AG(0 \leq \texttt{I} \leq 16)$ are included in the input formula set. These inputs, along with categorisation of I/O (e.g. marking start as a shared signal, keyok as an uncontrollable input, and pal,ntsc,stop,ready as converter-generated signals), are used by the conversion algorithm to generate a converter. The conversion algorithm, using the above inputs, successfully generates a converter $\mathcal{C}$, obtained by determinizing the maximally-permissive non deterministic converter obtained from Alg. 1.

2) Incremental Conversion: In incremental conversion, IPs are added to the SoC *incrementally* in multiple stages (as shown in Fig. 2(b)). Whenever one or more IP is added to the system, the conversion algorithm is used to generate a converter that integrates the newly-added IP(s) to the system. We illustrate how this technique works by generating a converter for the set-top box example (Fig. 3(a)) in 2 stages, as shown in Fig. 6.



(a) Stage 1 SoC layout (b) Stage 2 SoC layout

Fig. 6: Two-stage construction of the set-top box SoC

In the first stage, we integrate the protocols P_S and P_T. We first over-sample them and then compute their parallel composition $P_S||P_T$. As the two protocols interact each with the other using the IR buffer, we include the data constraint $AG(0 \leq \texttt{I} \leq 16)$ (as explained in Sec. III-A2), and mark all I/O signals (done, ready, start etc.) as uncontrollable.

The above inputs are used to automatically generate a maximally-permissive, non deterministic converter $\mathcal{C}_1$ (Fig. 6(a)), which executes using $mclk$ and ensures the satisfaction of the data constraint. The converter however does not further constrain the system in stage 1. The layout of the system obtained from stage 1 is shown in Fig. 6(a). $\mathcal{C}_1$ can be determinized to generate one deterministic converter. However, since the goal is to add more IPs, the non deterministic converter $\mathcal{C}_1$ is used for the next stage.

The converted system $\mathcal{C}_1//(P_S||P_T)$ (equivalent to the graph generated by the algorithm in stage 1), is used as one of the IP inputs for stage 2. Here, we over-sample the remaining IPs P_U and P_T and the parallel composition $(\mathcal{C}_1//(P_S||P_T))||P_U||P_V$ is computed. We now include AGAF$KeyIn_{16}$ and AG($SigRd_8 \Rightarrow$ A($\neg Off$ U $KeyIn_{16}$)) as input constraints to the algorithm. Finally, the common signal start is marked as shared (hence bufferable in the stage 2 converter), signals done and keyok are marked as uncontrollable signals, while all remaining signals are marked as converter-generated. These inputs are read by the conversion algorithm which successfully generates a converter $\mathcal{C}_2$ to satisfy all constraints described above. In fact, $\mathcal{C}_2$, the non-deterministic converter generated in this stage is equivalent to the non-deterministic converter generated in the single-step conversion process. $\mathcal{C}_2$ can be processed to extract a deterministic converter. Note that $\mathcal{C}_2$ prevents $\mathcal{C}_1$ from interacting directly with the environment, and instead serves as the communication interface between the two.

Incremental construction can be done by using any combination of IPs in stages 1 and 2. For example, we could have integrated P_S and P_U first, and then add the other IPs in stage 2. However, the only difference is that the user may need to provide different specifications in this case. For example, as P_S and P_U do not interact directly, the input specification to the algorithm would be AG*true* (a default property that is used whenever no other specifications are available). The choice of which IPs to integrate first depends solely on the designer (governed by issues such as desired IP placements on chip).

3) Comparison: Traditional SoC design favors the single-step construction process as the system is integrated only after all IPs are identified. Single-step construction also reduces the time spent in specifying the behaviour of each intermediate stage for incremental conversion. However, incremental conversion has a few advantages that can make it useful in some situations. Firstly, a converter that controls all IPs of a system (as constructed in single-step conversion), that can number well into the hundreds, is difficult to realize in hardware as each IP must read its I/O through the converter. This may result in increasing the wiring congestion on chip and can result in latency errors, as identified in [10]. On the other hand, in incremental conversion, converters can be built to control the interaction of IPs located closer to each other. The user can also choose to generate different converters to implement different bus policies. Incremental conversion can also aid in the *reuse of SoCs* where an existing (pre-converted) system can be integrated with more IPs to extend its functionality. Hence, incremental conversion can play an important part in situations where a large number of IPs have to be integrated, and issues relating to wire congestion and latency errors are more important to avoid than the extra effort involved in generating multiple converters.

IV. Results

The proposed conversion algorithm has been implemented in Java, and Tab. I presents results obtained from AMBA SoC case studies. The SKS abstractions of IP protocols were extracted from AMBA white papers and HDL implementations.

The columns of Tab. I respectively describe each SoC, its size (number of states without counter-based unrolling) of the parallel composition of its IP protocols, and the behaviour enforced by the converter, respectively. Each benchmark contains IPs that execute using different clocks. Both incremental and single-step conversion were successfully used for every case.

IPs	|S|	CTL Properties
1. Master (single-write) Slave (single-read), AMBA ASB Arbiter	96	No databus over/under flows
2. Master (multi-write) Slave (single-read), AMBA ASB Arbiter	96	Correct sequence of writes/reads to avoid data buffer over/under flows
3. Master (multi-write 26 bit), Slave (multi-read 4 bit), AMBA ASB Arbiter	96	Correct invocation sequence to avoid over/underflows in data bus
4. Master 1 (7 bit write), Slave (8 bit read/write), Master 2 (11-bit read), AMBA ASB Arbiter	384	Sequenced invocation of masters and slaves to avoid overflows
5. 3 Master, 2 Slaves, AMBA ASB Arbiter	3456	Correct sequencing of data (M1→S1→M2→S2→M3)
6. 6-master SoC (3 masters known)	448	Partial closing, correctly invoke known masters
7. 6-master SoC (all masters known)	28572	Correct activation sequence
8. Master, slave (2-channel) (Different data-widths)	16	No data-loss
9. Master (28-bit write) Slave 1 (6-bit read) Slave 2 (8-bit read)	64	No data-loss

TABLE I: Implementation Results

The first four cases involve one or more masters that write data (single or multiple times per invocation), and a slave that is required to read this data. The data-widths of data operations is the same in case 1, a multiple of one another in case 2, and completely unrelated in cases 3 and 4. For each of these cases, and especially for cases 3 and 4 that cannot be handled by existing techniques, our approach was able to produce correct converters. Case 5 involves generating a converter that sequences data flow between multiple masters and slaves by enforcing a custom scheduling scheme from a generic AMBA ASB arbiter.

Cases 6–7 show that we can *close* a converted SoC to its known IPs. In case 6, only 3 of a possible 6 masters that a bus arbiter can connect to are known (part of the converted system). The generated converter ensures that the converted system remains open with respect to unknown masters such that the arbiter can respond to handshaking signals from both known and unknown (environment/uncontrollable) IPs. In case 7, all masters are known, and the converter completely closes the system to ensure that all masters are activated in the order of requesting, by using buffering and event forwarding.

In case 8, the conversion algorithm works with multiple data buffers (in this case the data-bus and a shared buffer). This handling of multiple data buffers between the same pair of protocols is another unique feature of the proposed approach. Similarly, case 9 shows how the proposed approach can handle multiple read/write operations of varying data-widths by different IPs (2 readers/1 writer) over a single medium. Again, these approaches cannot be handled by existing approaches.

Overall, it was found that the proposed approach can work with IPs with different clocks, and can generate converters for problems (multiple data buffers, non-related data-widths, multiple protocols) that cannot be addressed by any one existing approach. Moreover, it is not necessary for the user to provide *all* CTL constraints that define the correct behaviour of the system, since the maximally-permissive converter generated by one step of conversion can be used with more IPs/CTL properties in the next.

It should be noted that although both approaches result in the same functional behaviour, incremental design requires more user effort and results in an additional converter for each iteration step, which requires additional chip resources. However, incremental conversion gives greater flexibility and allows decentralization of control, where each converter connects to a smaller number of components, allowing a reduction in wire congestion as compared to a centralized converter.

V. Conclusions

This paper presented protocol conversion framework for the correct-by-construction design of of SoCs. The flexible algorithm allows single-step or incremental design of SoCs. The proposed approach can generate maximally-permissive converters for a given SoC. Using various benchmarks from the ARM AMBA bus, we show that common causes of mismatches such as multiple-clocks, multi-directional IP communication, and control and data-width mismatches, that cannot be resolved by existing techniques, can be resolved in our framework. The future directions for this work include the resolution of more complex mismatches such as interface inconsistencies between IPs, and unrelated clock sources. We also intend to carry out more a comparative study of our technique with existing manual and automatic conversion techniques in the near future.

VI. Acknowledgements

We would like to acknowledge the invaluable input on this work received from Dr. Alain Girault and Dr. Gregor Goessler.

References

[1] L. de Alfaro and T. A. Henzinger, "Interface automata," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 5, pp. 109–120, 2001.

[2] S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee, "On relational interfaces," in *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*. New York, NY, USA: ACM, 2009, pp. 67–76.

[3] M. D. Bozga, V. Sfyrla, and J. Sifakis, "Modeling synchronous systems in BIP," in *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*. New York, NY, USA: ACM, 2009, pp. 77–86.

[4] K. Avnit, V. D'Silva, A. Sowmya, S. Ramesh, and S. Parameswaran, "A formal approach to the protocol converter problem," in *DATE*, March 2008, pp. 294–299.

[5] R. Kumar, S. Nelvagal, and S. I. Marcus, "A discrete event systems approach for protocol conversion," *Discrete Event Dynamic Systems*, vol. 7, no. 3, pp. 295–315, 1997.

[6] 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 *ICCAD*, 2002, pp. 132–139.

[7] M. Tivoli, P. Fradet, A. Girault, and G. GöSSler, "Adaptor synthesis for real-time components," in *TACAS*. Springer, 2007, pp. 185–200.

[8] R. Sinha, P. S. Roop, S. Basu, and Z. Salcic, "Multi-clock SoC design using protocol conversion," in *DATE*. IEEE, 2009, pp. 123–128.

[9] R. Sinha, "Automated techniques for formal verification of SoCs," Ph.D. dissertation, University of Auckland, 2009.

[10] L. Carloni, K. McMillan, A. Saldanha, and A. Sangiovanni-Vincentelli, "A methodology for correct-by-construction latency insensitive design," in *ICCAD*, 1999, pp. 309–315.