Equivalence Checking For Synchronous Elastic Circuits

Vidura Wijayasekara and Sudarshan K. Srinivasan
Department of Electrical and Computer Engineering
North Dakota State University
Fargo ND 58104
Email: vidura.wijaysekara@my.ndsu.edu and sudarshan.srinivasan@ndsu.edu

***Abstract*—Synchronous elastic circuits are clock-based latency insensitive circuits. Elastic circuits are typically synthesized from synchronous circuits. After synthesis, additional buffers can be arbitrarily inserted in the data path of an elastic circuit without altering its functionality to resolve timing issues. We have developed a verification tool that can check the equivalence of an elastic circuit (even after the inclusion of any number arbitrarily placed additional buffers) with its synchronous parent circuit. The tool inputs elastic circuits in VHDL. We have developed an algorithm that automatically computes efficient mapping functions used to map elastic circuit states with states of the synchronous parent circuit. Such mapping functions (required for equivalence checking) can be challenging to compute automatically, as the inclusion of additional buffers can drastically alter the pattern of data flow through the circuit. The capacity of the equivalence checker is demonstrated with results from 24 elastic circuit benchmarks, many of which have over 100,000 gates and 1,000 latches.**

I. Introduction

Latency insensitive (LI) [1], [2] design addresses the wire delay challenge for nanometer technologies within the synchronous framework. The central idea is to use relay stations—which function like latches—to break long wires that cause violations of timing requirements imposed by the clock. A handshaking protocol (known as LI protocol) is used to allow for insertion of buffers/relay stations without altering the functionality of the system. One of the primary impacts is that LI design aids in intellectual-property reuse in systems-on-chip by reducing the expensive iterations required for timing closure [3]. LI design is a very active area of research in both academia and industry and many LI design paradigms, implementations, and optimizations have been proposed [1], [2], [4], [5], [6], [7], [8].

Synchronous Elastic Networks (SEN) [4], [9] is one effective approach to implement LI designs and also synthesize LI systems from synchronous parents. The idea with SEN is to replace all flip flops with elastic buffers (EBs) that are constructed from two elastic half buffers (EHBs), namely a master EHB and a slave EHB. EHBs are gated latches whose clock input is produced by elastic controllers that are used to implement the LI protocol. The clock network is replaced by a network of elastic controllers, where each controller is used to control the elastic buffers in a design stage and synchronize with the controllers of adjacent design stages. In the resulting elasticized design, elastic buffers can be inserted in any place in the data path to break long wires. Figure I shows the example of an elasticized processor data path with two additional elastic buffers.

In this work, we present an automated equivalence checker that verifies the functionality of the elastic circuit (even after the inclusion of any number of arbitrarily placed additional elastic buffers) against its synchronous parent circuit. Why would such an equivalence checker be useful in practice? Consider the IP reuse based SoC design paradigm. The SoC design is constructed from digital IP components. Some of these IP components are obtained from third party vendors, some are proprietary IPs, while others may be designed from scratch. All these IP components have to be integrated together in a technology (that some of these IPs may not have been designed for). At this stage in the design cycle, some of the IPs will require infeasible and expensive redesign to resolve timing issues that arise in the new technology. The designer can instead opt to generate elastic versions of these IPs and solve the timing issues in the elastic domain without requiring expensive redesign. However, after generating the elastic design and fixing all timing issues using additional elastic buffers, the resulting elastic IP will have undergone a considerable transformation w.r.t. the original synchronous IP. The designer cannot assume that the resulting elastic IP will not have bugs. The designer will instead have to expend time and resources to verify the elastic IP.

Our fully automated equivalence checker comes to the aid here and addresses the verification problem of the elastic IP. One may argue that the transformations used in generating elastic circuits are already proved to be correct and so verification is not required. However, the IP circuits will undergo considerably significant mutations and there could be many sources of error in the elasticization process (such as buggy synthesis tools, and manual tinkering of the circuit). Also, commercial design processes will not assume the functional correctness of the resulting design and will require verification.

The equivalence checker takes as input the elastic and synchronous circuits in RTL VHDL. The checker can currently handle closed circuits, where the behavior of the elastic controller network is deterministic. For future work, we will extend the techniques implemented in the checker to handle elastic controllers with non-deterministic behaviors. The equivalence verification proof obligations are automatically

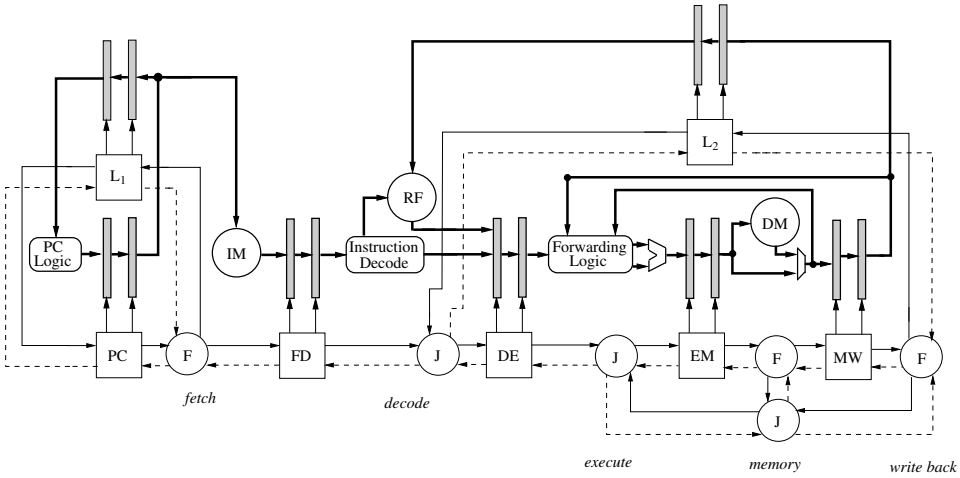

Fig. 1. High-level organization of a 5-stage elastic processor with two additional elastic buffers L1 and L2. The J and F blocks denote the join and fork structures.

generated. An SMT solver is used in the back end to check the proof obligations. The notion of equivalence used is described in Section II. The equivalence verification algorithms incorporated in the tool are described in Section III. The overall tool flow is described in Section IV. Experimental results, related work, and conclusions are given in Sections V, VI, and VII, respectively. The capacity of the equivalence checker is demonstrated with results from 24 elastic circuit benchmarks.

II. Background: Equivalence Notion

The notion of equivalence we use is Well-Founded Equivalence Bisimulation (WEB) refinement [11][12]. A formal and detailed description of WEB refinement is provided in [11][12]. Here, we provide a brief overview of the key features of WEB refinement relevant to the problem at hand. WEB refinement is a notion of equivalence that can be used to check if an implementation system satisfies its specification system, even if the implementation and specification are defined at very disparate levels of abstraction.

In the context of refinement, digital systems are modeled as transition systems (TSs). A TS is a three tuple and includes the set of states of the system and a transition relation that defines the state transitions of the system. The behaviors of a system modeled as a TS are defined using the notion of paths. A *path* in a TS is a sequence of states, say s^0,s^1,s^2, ..., where s^0 is the initial state and s^0 transitions to s^1, s^1 transitions to s^2, and so on. An infinite sequence of such states is a *full path*. The behaviors of a system is the set of all full paths in the TS of the system.

The implementation behaves correctly as given by the specification, if every behavior of the implementation is matched by a behavior of the specification and vice versa. However, the implementation and specification may not have the same timing behavior. For example, if the implementation is an elastic circuit and the specification is a synchronous circuit, the elastic circuit may take many steps to match a single step of the synchronous circuit. This phenomenon is known as stuttering. To account for such situations, multiple but finite transitions of the implementation system are allowed to match a single transition of the specification system.

Another issue is that to check equivalence, synchronous states and elastic circuit states need to be compared. However, these states look very different. While the synchronous states have flip-flops, elastic circuit states have elastic buffers and also possibly additional buffers. WEB refinement employs refinement maps, functions that map implementation states to specification states to bridge this abstraction gap.

Refinement-Based Correctness Formula: Manolios [12] has shown that it is enough to prove the following refinement-based correctness formula to establish the equivalence of an implementation and specification based on WEB refinement. *rank* is used to distinguish stutter from deadlock. *rank* is a witness function from implementation states to natural numbers whose value decreases when the implementation stutters.

Definition 1. *(Refinement-Based Correctness Formula) For every implementation state w, let s be a specification state such that s = refinement-map(w). If u is the specification successor of s (u = Sstep(s)) and v is the implementation successor of w (v = Istep(w)), then one of the following has to hold:*

1) $u = r(v)$ {*non-stuttering step*}
2) $s = r(v) \wedge rank(v) < rank(w)$ {*stuttering step*}

In the above, *Sstep()* and *Istep()* are functions that define the transitions of the implementation and specification. Note that in order to establish the equivalence of an implementation and specification system, all the reachable states of the implementation should be checked to see if they satisfy the above correctness formula. The correctness formula given above is expressible in a decidable fragment of first-order logic. Therefore, verifying equivalence of an implementation and a specification based on WEB refinement can be accomplished automatically using an SMT solver if a suitable refinement

map and rank function are available.

III. Automating Computation of Refinement Maps

If an implementation is a refinement of a specification, then a refinement map does exist [13]. However, finding/constructing a refinement map can be very challenging in practice and can require deep understanding and analysis of the systems being compared. Also, often times, the refinement map can be computationally expensive, resulting in the approach being infeasible. The primary contribution of this work is a fully automated procedure to compute refinement maps for the elastic verification problem described in the introduction section. The refinement maps synthesized by our procedure leads to an efficient and scalable verification approach. The key idea is to use reachability analysis of the elastic controller network (using the notion of token-flow diagrams) to systematically synthesize refinement maps.

We use the example of a 5-stage elastic processor circuit (shown in Figure I) to illustrate the ideas presented in the rest of the paper. We call this elastic circuit example E-2. The elastic processor pipeline has 5 elastic buffers corresponding to pipeline latches *pc*, *fd*, *de*, *em*, and *mw*. We also inserted two additional elastic buffers at arbitrary points in the data path that are labeled *l1* and *l2*. The elastic controller network is also shown in the figure. The controllers use *valid* (indicated by solid lines) and *stop* signals (indicated by dashed lines) to implement the LI protocol. The synchronous parent of E-2 would have regular registers instead of EBs, and will not have the additional EBs. A clock signal would replace the entire elastic controller network.

A. Token-Flow Diagrams For Elastic Circuits

In this section, we describe a procedure for generating token-flow diagrams for elastic circuits. Token-flow diagrams are used for reachability analysis of the elastic control layer and also for synthesizing refinement maps.

S_y is the set of stages in the synchronous circuit. S_y is an ordered set. Each stage in S_y is hence identified by a unique number i, where $0 \leq i < |S_y|$. E is the set of stages in the elastic circuit. E is an ordered set. Each stage in E is hence identified by a unique number i, where $0 \leq i < |E|$. Each stage of the elastic circuit corresponds to an elastic buffer and vice versa. The elastic buffers are classified as non-additional elastic buffers and additional elastic buffers. Note that each non-additional EB would correspond to a stage in the synchronous parent circuit. An EB is described to be in an empty, half, or full state, if the EB holds 0, 1, or 2 valid data units, respectively.

Definition 2. *The token state of an elastic controller network circuit, T_E is defined as the set $\{\langle\ m_0,\ s_0\ \rangle, \langle\ m_1,\ s_1\ \rangle, ..., \langle\ m_{|E|-1},\ s_{|E|-1}\ \rangle\}$ such that $m_0, m_1, ..., m_{|E|-1}, s_0, s_1, ..., s_{|E|-1} \in \mathbb{N}$.*

The token state is used to capture the distribution of valid data in the elastic circuit. m_i and s_i indicate the token values corresponding to the master and slave EHBs, respectively. EHBs without valid data are assigned the token value 0. EHBs with positive non-zero token values indicates valid data. Unique data tokens are identified by unqiue token values.

We next define elastic token-flow diagrams (*etfd*), which are used to compute refinement maps. In the definition, $\rightarrow_{ecn}$ is the transition relation of the elastic controller network circuit.

Definition 3. *An elastic token-flow diagram (etfd) is a finite sequence of elastic token states such that, for any adjacent pair of token states in the sequence, say T_{Ei} and T_{Ej}, where T_{Ei} and T_{Ej} are elastic token states corresponding to the elastic controller network states e_i and e_j, $(e_i\ ,\ e_j) \in \rightarrow_{ecn}$.*

The token-flow diagram for a sequence of states of the E-2 elastic circuit is shown in Table II. First, we define a procedure that given a token state of the elastic circuit, computes the next token state of that circuit. The procedure also takes as input the connectivity matrix of an elastic circuit M_C, which is defined as follows.

Definition 4. *The connectivity matrix of an elastic circuit $M_C = [a_{ij}]_{|E|\times|E|}$ such that a_{ij} = 1 if there is a data channel from EB i to EB j. Otherwise a_{ij} = 0.*

TABLE I
Datapath connectivity matrix M_c for benchmark E-2

	pc	fd	de	em	mw	l1	l2
pc	0	1	0	0	0	1	0
fd	0	0	1	0	0	0	0
de	0	0	0	1	0	0	0
em	0	0	0	1	1	0	0
mw	0	0	0	1	0	0	1
l1	1	0	0	0	0	0	0
l2	0	0	1	0	0	0	0

Another data structure used in the procedure is the token transition matrix M_{TT}. For EBs with multiple destinations, it is possible that in a cycle, data is transferred in only some of the output channels, but not all. To keep track of this, we use the M_{TT} matrix. Initially, all entries in the matrix are assigned a value 0. If data transfers from a source EB on only some but not all output channels, then the channels on which the transition takes place is given a value 1 in M_{TT}. When in the future, the data from the source is transferred on all output channels, then all the output channels from that source are reassigned a value 0 in M_{TT}.

Note that unlike synchronous circuits, data need not always transfer from source EB to destination in a given cycle. In fact, data transfers only when valid data is available at the source EB and the destination EB is ready to accept data (which is indicated by deasserting the stop signal). Therefore, we define the function *ValidDataInputs(i)* that determines if all the sources of a destination EB have valid data.

Definition 5. *ValidDataInputs(i)* =

$$\bigwedge_{0\leq j<|E|} \Big\{ \big(M_c[j][i] = 1\big) \rightarrow \big(M_{TT}[j][i] = 0 \wedge s_j \neq 0\big) \Big\}$$

Procedure 1 Next token calculation for master EHBs

1: **for** $i \leftarrow 0$ **to** $|E|$ -1 **do**
2: **if** $m_i \neq 0$ **then**
3: $m'_i \leftarrow m_i$
4: **else if** *ValidDataInputs(i)* **then**
5: $TokenGenerator \leftarrow TokenGenerator + 1$
6: $m'_i \leftarrow TokenGenerator$
7: **for** $j \leftarrow 0$ **to** $|E|$-1 **do**
8: $M_{TT}[j][i] \leftarrow 1$;
9: **end for**
10: **else**
11: $m'_i \leftarrow 0$;
12: **end if**
13: **end for**

In the above definition, a destination EB i has valid data at all its sources j only if slave EHB (s_j) is not empty, and that data in s_j has not previously transferred on that channel ($M_{TT}[j][i] = 0$). We only check those EBs that are sources to EB i ($M_c[j][i] = 1$).

The algorithm that computes the next token state is given in two steps. In the first step (shown in Procedure 1), the next token value of all master EHBs (m'_i) is computed. The procedure enumerates through all the master EHBs and uses the following property of elastic circuits.

Property 1. *[4] For any elastic buffer in a half state (one of the EHB has valid data and the other EHB is empty), the master EHB will be empty and the slave EHB will have valid data.*

As can be inferred from the above property, if the master EHB is not empty, then the corresponding EB is full, meaning that both master and slave EHBs hold valid data. In this state, the master EHB retains its previous value and the EB does not accept any new data. Otherwise, if the master EHB is empty, then the procedure checks to see if all the sources to EB i have valid data. If this is the case, a new unique token number is generated and assigned to m'_i, the next value of m_i. The *TokenGenerator* is a counter that is initialized to a natural number value greater than the greatest token value in the input elastic token state. The M_{TT} matrix is updated. If otherwise, one or more of the sources do not have valid data, then the Master EHB will become empty. Therefore m'_i is assigned zero in this case.

The second step of the algorithm computes the next value of slave EHBs and is shown in Procedure 2. If the slave EHB is currently empty, then it will remain empty. Otherwise, if transfers have taken place on all the output channels (determined by the *DataTransferredAll* function), then the slave EHB can let go of its current token value and is updated to empty. The *DataTransferredAll* function is defined as follows.

Definition 6. *DataTransferredAll(i)* =

$$\bigwedge_{0 \leq j < |E|} \Big\{ \big(M_c[i][j] = 1\big) \rightarrow \big(M_{TT}[i][j] = 1\big) \Big\}$$

Procedure 2 Next token calculation for slave EHBs

1: **for** $i \leftarrow 0$ **to** $|E|$-1 **do**
2: **if** $s_i = 0$ **then**
3: $s'_i \leftarrow 0$
4: **else**
5: **if** *DataTransferredAll(i)* **then**
6: $s'_i \leftarrow 0$
7: **for** $j \leftarrow 0$ **to** $N-1$ **do**
8: $M_{TT}[i][j] \leftarrow 0$
9: **end for**
10: **else**
11: $s'_i \leftarrow s_i$
12: **end if**
13: **end if**
14: **if** $(s'_i = 0) \wedge (m'_i \neq 0)$ **then**
15: $s'_i \leftarrow m'_i$
16: $m'_i \leftarrow 0$
17: **end if**
18: **end for**

The *DataTransferredAll(i)* function examines only those entries in M_{TT} corresponding to EB i that are destinations of EB i ($M_c[i][j] = 1$). The functions examines the output channels of EB i to determine if transfers have been completed on these channels ($M_{TT}[i][j] = 1$).

Also, the entries in the M_{TT} matrix corresponding to the output channels of EB i are assigned a value 0 to indicate that transfers have taken place on all the output channels from EB i. If there are still output channels in which transfers are yet to be completed, then the slave EHB retains its token.

After the slave EHBs have been updated (lines 1-13 of Procedure 2), the procedure checks the new token values of the master and slave EHBs. If the EB is in a half state where the slave is empty and the master has a token, then this token is transferred from master to slave (lines 14-16 of Procedure 2), due to Property 1. This completes the computation of the next token state of the elastic circuit.

B. Reachability for Elastic Controller Networks

The reachable states of an elastic controller network can be computed using token-flow diagrams. At reset, elastic buffers are initialized to the half state and additional buffers are initialized to the empty state. Such a reset state is a requirement of the SEN paradigm. To compute token-flow diagrams for reachability, the initial token-state is constructed by assigning a token value 0 to the empty EHBs and a unique non-zero natural number token value to each of the non-empty EHBs.

We use the notion of a binary token state for reachability, which is defined below.

Definition 7. *The binary token state corresponding to the token-state T_E of an elastic circuit is defined as the set* $\{\langle$

TABLE II
TOKEN FLOW DIAGRAM FOR E-2

State	0 (pc)		1 (fd)		2 (de)		3 (em)		4 (mw)		5 (l1)		6 (l2)	
	m	s	m	s	m	s	m	s	m	s	m	s	m	s
0	0	1	0	2	0	3	0	4	0	5	0	0	0	0
1	0	0	1	2	0	0	0	6	0	4	0	1	0	5
2	0	7	0	1	0	8	0	6	6	4	0	0	0	4
3	0	0	0	7	0	9	0	10	0	6	0	7	0	0
4	0	11	0	7	0	0	0	12	0	10	0	0	0	6
5	0	0	0	11	0	13	0	12	12	10	0	11	0	10
6	0	14	0	0	0	15	0	16	0	12	0	0	0	0
7	0	0	0	14	0	0	0	17	0	16	0	14	0	12
8	0	18	0	0	0	19	0	17	17	16	0	0	0	16
9	0	0	0	18	0	0	0	20	0	17	0	18	0	16
10	0	21	0	0	0	22	0	20	20	17	0	0	0	17
11	0	0	0	21	0	0	0	23	0	20	0	21	0	17
12	0	24	0	0	0	25	0	23	23	20	0	0	0	20
13	0	0	0	24	0	0	0	26	0	23	0	24	0	20

bm_0, bs_0 $\rangle$, $\langle$ bm_1, bs_1 $\rangle$, ..., $\langle$ $bm_{|E|-1}$, $bs_{|E|-1}$ $\rangle\}$ *such that*

$$bm_i/bs_i = \begin{cases} 1 & m_i/s_i > 0 \\ 0 & m_i/s_i = 0 \end{cases}$$

The binary token state captures the distribution of data in the elastic circuit without distinguishing the data units. Thus, two elastic controller network states with the same binary token states are essentially equivalent.

The reachable states of the elastic controller network are computed by simulating the token-flow diagram until a binary token state is reached that has already been visited before. Note that since we consider only deterministic elastic controller networks, the reachable states will be a finite sequence of states that will converge and hence can be computed as an *etfd*. The resulting token states correspond to the reachable states of the controller network of the elastic circuit. Table II shows the token-flow diagram for computing the reachable states of the E-2 benchmark. As can be seen from the table, states 7, 9, 11, and 13 map to the same binary token states. Also, states 8, 10, and 12 map to the same binary token states. Therefore, the reachable states of the elastic controller network of E-2 are states 0 through 8, after which the states start to repeat.

C. Token-Flow Diagrams for Synchronous Circuits

Token flow diagrams for the synchronous parent circuit (*stfd*), which is the specification, are also computed. Given an *etfd*, an *stfd* captures how the tokens in the *etfd* would progress in the synchronous parent circuit. The token-flow diagram for the synchronous circuit corresponding to the *etfd* of Table II is shown in Table III. The refinement map is constructed by examining the token states of the elastic and the synchronous circuits.

As such, *etfd* and *stfd* are matrices, where rows correspond to token states and columns correspond to design stages. In the *etfd* matrix, each element corresponds to two token values $etfd[i,j]_m$ and $etfd[i,j]_s$, indicating master and slave tokens, respectively.

The synchronous circuit is comprised of registers as opposed to EBs. Therefore, a stage in the design has only one token value in contrast to two token values (master and slave) for EBs. Second, the registers always hold valid data and therefore always have a non-zero token value. Note that the notion of valid data is w.r.t. elastic control and is used to contrast the progress of data in the elastic and synchronous circuits. For example, bubbles in the design due to synchronous control (for example, pipeline control) are still considered to be valid data in both circuits. Invalid data are only the result of bubbles introduced by elastic control.

TABLE III
TOKEN FLOW DIAGRAM FOR SYNCHRONOUS CIRCUIT OF E-2

State	pc	fd	de	em	mw
0	1	2	3	4	5
1	7	1	8	6	4
2	11	7	9	10	6
3	14	11	13	12	10
4	18	14	15	16	12
5	21	18	19	17	16
6	24	21	22	20	17

The synchronous specification and the elastic implementation are flow equivalent [4]. Therefore, the flow of tokens in a stage of the synchronous specification circuit can be obtained by removing bubbles (tokens with value 0) and stale tokens (token values duplicated in the sequence) from the flow of tokens of the slave EHB of the corresponding stage in the elastic circuit. The slave EHB is chosen because all the valid data that flow through an EB is retained in the slave EHB for at least one cycle. Therefore, we construct the *stfd* column by column. Each column of the *stfd* is obtained by removing the tokens with value zero and duplicate tokens from the column in *etfd* corresponding to the slave EHB. An *stfd* satisfies the following property.

Property 2. *The sequence of unique non-zero token values in any column of an stfd and the corresponding slave column of etfd are identical.*

D. Refinement Map Computation

The refinement map for the equivalence verification problem at hand takes as input an elastic circuit state and returns the

corresponding synchronous circuit state. Due to the presence of additional buffers, design stages in the elastic circuit progress at different speeds when compared to corresponding stages in the synchronous circuit. For example, in the E-2 benchmark and its corresponding synchronous specification, for states with the same program counter value, the value in the decode stage may not be the same. Thus to map elastic states to synchronous states, we choose a design stage and use it as a reference point to construct the mapping. An elastic state w will be mapped to a synchronous state s such that the values of the latches/registers in the stage corresponding to the reference point in both w and s match.

Next, we want to define a projection function that given an elastic state as input, constructs the corresponding synchronous state. A systematic approach to computing the projection function is possible by comparing the distribution of data tokens in an elastic state and its synchronous counterpart. The token-flow diagrams can be used to determine the distribution of data tokens in the reachable states of both elastic and synchronous circuits.

Procedure 3 Refinement Map

```
1:  for elastic-state ← 0 to |etfd|-1 do
2:    h ← elastic-state
3:    repeat
4:      t_p ← etfd[h, reference-point]_s
5:      h ← h − 1
6:    until t_p ≠ 0
7:    sync-state ← 0
8:    while stfd[sync-state, reference-point] ≠ t_p do
9:      sync-state ← sync-state + 1
10:   end while
11:   for j ← 0 to |S_y|-1 do
12:     h ← 0
13:     while etfd[elastic-state − h, j]_s ≠ stfd[sync-state, j]
        do
14:       h ← h − 1
15:     end while
16:     refinement-map[elastic-state, j] ← h
17:   end for
18: end for
```

Since the distribution of valid data is different for different reachable states of the elastic controller network, one projection function is defined for each controller reachable state by examining the token flow diagrams of the elastic and synchronous circuits, *etfd* and *stfd*, respectively. Note that a reachable state of the controller network corresponds to many states of the elastic circuit. The objective of the projection function is to construct the synchronous state from the elastic state. To achieve this, history information may be used. For example, to get the value of the *fd* latch, for the synchronous state from the elastic state, the master and slave EHBs of *fd* can be examined. However, it is also possible that *fd* is empty or the *fd* value in the elastic state is not the required value to construct the synchronous state. Note that this is because while stages in the synchronous circuit progress together, stages in the elastic circuit can progress at different rates. In such a situation, the projection function can examine history values of *fd* to obtain the required value.

As such, the refinement map is a matrix *refinement-map*$_{|etfd|\times|S_y|}$, where each row corresponds to a reachable state of the elastic controller network. Each row has one entry for each stage in the synchronous circuit. Each $[i, j]$ entry in *refinement-map* indicates which history should be projected for design stage j in controller state i. For example, an entry of 0 indicates the current value should be projected and an entry of -2 indicates that the value of the stage 2 cycles before should be projected.

An algorithm for computing the *refinement-map* matrix is given in Procedure 3. The algorithm enumerates over the rows of *etfd*. We describe the algorithm using row 11 of *etfd* shown in Table II. For this circuit, *reference-point*=0 (the program counter is chosen as the reference point). t_p is the token value corresponding to the *reference-point* in *etfd*. For *elastic-state*=11, t_p=21, which is the first valid token (when searching upward) in the *pc* column of *etfd*. Next, we find the synchronous token state (*sync-state*) corresponding to elastic token state 11 by searching the *reference-point* in *stfd* for t_p. A match is found for *sync-state*=5. Then we compute *refinement-map*[11] by searching backward from row 11 of *etfd* for a match with tokens in each stage of *sync-state*=5. *refinement-map*[11] = [-1, -2, -3, -3, -3]. Using the *refinement-map* matrix, proof obligations are generated for each elastic controller state as described next.

We choose a stage of the elastic design as a reference point (*reference-point*) such that the procedure for finding a refinement map will always complete successfully. Note that *reference-point* should be a stage that has a counterpart in the synchronous circuit. Therefore, *reference-point* has to be a non-additional EB. For example, *fd* can be an *reference-point*, whereas *l1* cannot. To find a suitable *reference-point*, we define the following graph.

Definition 8. *$G_{elastic}(V,E)$ of an elastic circuit is a directed graph, such that vertices (V) are the EBs of the circuit and the edges (E) correspond to data channels directed opposite to the direction of data flow.*

For example, if there is a data channel from EB_i to EB_j then there is an edge from V_j to V_i in the graph.

Property 3. *Every cycle in graph $G_{elastic}(V,E)$ includes at least one EB node, which is a non-additional buffer.*

The above property is a result of the elastic design process that incorporates additional EBs only to break paths between two non-additional EBs. The elastic design/synthesis process does not create cycles of additional EBs [4].

Definition 9. *$T_{elastic}$ is the set of directed graphs obtained by removing edges from $G_{elastic}(V,E)$ that are incoming to all non-additional elastic buffers.*

Lemma 1. *Every directed graph in set $T_{elastic}$ is a tree with*

a non-additional EB as the root.

Lemma 2. $|T_{elastic}| = |S_y|$.

The above lemmas derive from Property 3. The lemmas indicate that $T_{elastic}$ contains one tree each for every non-additional EB.

Definition 10. *Latency of EB i of an elastic circuit is the depth of the tree* $\in T_{elastic}$ *with root i.*

Definition 11. *reference-point of an elastic circuit is the non-additional EB with the greatest latency in the circuit.*

Note that the *reference-point* need not be unique. Any EB with the greatest latency can be chosen as the *reference-point*.

Theorem 1. *If S is a closed synchronous circuit and E is the elastic implementation of S obtained using the SEN approach [4][9], then Procedure 3 will complete for any such S and E.*

Proof: Procedure 3 can be analyzed in 3 parts. In the first part (lines 2-6), the procedure searches backward starting from row *elastic-state* of *etfd* for a non-zero token value in the column corresponding to the slave of the *reference-point*. For every row of *etfd*, this search will complete successfully because in the reset state (row 0 of *etfd*) of the elastic circuit, slave EHBs of every non-additional EB is initialized with a non-zero token value. t_p is the token-value found as the result of this search.

In the second part (lines 8-10), the procedure searches for t_p in column *reference-point* of *stfd* starting from row 0. The search will complete due to Property 2, which states that the sequence of tokens in the slave column (stage) of *etfd* and the corresponding column of *stfd* are identical modulo bubbles and duplicated values.

In the third part (lines 11-17), the procedure searches backward in *etfd* starting from row *elastic-state* for every token in row *sync-state* of *stfd*. Note that the searches are done in corresponding columns of *etfd* and *stfd*. The search in each column will complete because the reference point was chosen as the point of synchronization. From Definition 11, the reference point has the greatest latency, so its progress is the slowest. Therefore, other columns (corresponding to other stages in the design) would have progressed faster and hence would have generated the tokens in row *sync-state*. Therefore, searching backward, each of the searches will complete. ■

IV. Tool Flow

The overall tool flow is shown in Figure 2. The equivalence checker takes as input the elastic and synchronous circuits in RTL VHDL and generates the verification proof obligations in SMT-LIB [14] format. The SMT logic used is QF_ABV, which is the logic of closed quantifier-free formulas over the theory of bitvectors and bitvector arrays. The proof obligations are then discharged using an SMT solver. The front end of the tool uses Verific Design Automation's parser platform [15] to parse the VHDL input circuits to an internal representation.

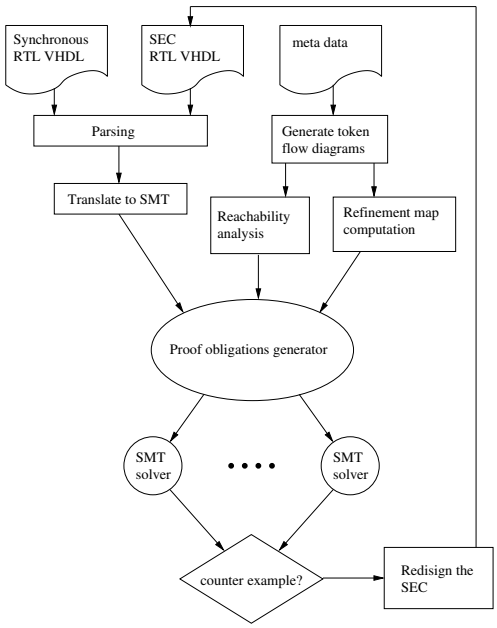

Fig. 2. Tool Flow

Also as input as meta data, the list of EBs/registers and their interconnection information is required for both input circuits. This information is used to generate the connectivity matrix M_C for both circuits, which are then used to construct the token flow diagrams for both circuits as described in Sections III-A and III-C. Also, the tool translates the VHDL input circuits into SMT-LIB functions that implement the transition relation corresponding to the circuits.

From the token flow diagram of the elastic circuit, the reachable states of its elastic controller network are computed (as described in Section III-B). The reachability analysis provides the reachable states and the transitions of the elastic controller network. Based on the reachability analysis, the tool generates invariant proof obligations to check that the elastic circuit satisfies these transitions. For example, if the E-2 circuit is in state 10, then it should transition to state 11 and if in state 11, it should transition to state 10. Note that states 10 and 12 are the same controller states, as they have the same distribution of tokens. The invariant proof obligation for the transition from state 10 to 11 is shown below.

$$\Big\{half(pc_{10}) \wedge empty(fd_{10}) \wedge half(de_{10}) \wedge half(em_{10}) \wedge full(mw_{10}) \wedge empty(l1_{10}) \wedge half(l2_{10})\Big\} \longrightarrow \Big\{empty(pc_{11}) \wedge half(fd_{11}) \wedge empty(de_{11}) \wedge half(em_{11}) \wedge half(mw_{11}) \wedge half(l1_{11}) \wedge half(l2_{11})\Big\}$$

In the above, *half(x)*, *empty(x)*, and *full(x)*, indicate that the EB x is in a half state, empty state, and full state. fd_{10} is the value of EB *fd* in state 10, and similarly for others.

Next, the tool computes the refinement map by examining the token flow diagrams as described in Section III-D. One refinement map function for each of the elastic controller

TABLE IV
RESULTS

Benchmark	No. of gates	No. of latches	Equivalence Checker Runtime (sec)	SMT statistics	
				Time (sec)	Memory (kB)
E-32-0	36,875	620	0.804	9.39	16.54
E-32-1	37,225	688	0.952	16.51	13.75
E-32-2	37,635	768	0.932	20.43	15.22
E-32-3	38,045	848	1.020	13.99	18.86
E-32-4	38,455	928	0.988	18.33	20.51
E-32-5	38,865	1,008	0.964	17.63	20.99
EB1-32-2	37,624	768	0.972	4.97	14.59
EB2-32-2	37,635	768	0.984	1.64	13.99
E-64-0	130,939	1,132	1.548	84.03	53.32
E-64-1	131,609	1,264	2.004	134.14	47.5
E-64-2	132,339	1,408	2.144	113.58	46.48
E-64-3	133,069	1,552	2.152	84.65	65.25
E-64-4	133,799	1,696	2.160	139.35	63.81
E-64-5	134,529	1,840	2.184	89.2	64.28
EB1-64-2	132,328	1,408	2.108	31.2	45.25
EB2-64-2	132,339	1,408	2.128	6.04	44.44
E-128-0	494,171	2,156	3.972	560.88	588.13
E-128-1	495,481	2,416	5.388	764.72	157.77
E-128-2	496,851	2,688	5.568	947.30	313.51
E-128-3	498,221	2,960	5.488	445.12	341.87
E-128-4	499,591	3,232	5.680	792.69	240.52
E-128-5	500,961	3,504	5.788	606.68	237.73
EB1-128-2	496,840	2,688	5.420	273.84	158.45
EB2-128-2	496,851	2,688	5.516	14.46	156.73

network states is then generated in SMT. Using the refinement map and the transition relation functions in SMT, the proof obligations required for equivalence verification based on WEB refinement are generated (see Section II). One proof obligation is generated for each transition based on the reachability analysis. Each of the proof obligations are generated in separate SMT files. Hence these obligations can be checked in parallel. If one or more of the proof obligations do not succeed, then the SMT solver will generate a counter example indicating one or more bugs.

A. Liveness

WEB refinement takes into consideration liveness. Establishing that an implementation refines its specification guarantees that the implementation will always progress and never deadlock w.r.t. the specification. Liveness is ensured using the mechanism of rank functions, functions that map the implementation (SEN) states to natural numbers. The goal is to devise a witness rank function such that for the stuttering steps of the implementation, the rank always decreases. There are no requirements of the rank of the implementation in non-stuttering steps, as the liveness of the implementation is witnessed in the fact that the implementation matches specification's progress in a non-stuttering step.

For the equivalence problem at hand, it is enough check that the implementation satisfies the invariants generated by the reachability analysis to guarantee liveness. To see why consider the following. The SEN framework is correct by construction and is guaranteed to be live. Therefore, every cycle in the reachability graph generated form the token flow diagrams should include at least one non-stuttering transition. Note that if all transitions in a cycle are stuttering, then this indicates deadlock. As long as the implementation satisfies the reachability invariants, it is guaranteed to never be in a stuttering cycle, guaranteeing liveness w.r.t. its specification. Also, if there are no stuttering cycles, its not hard to see that a rank function can be devised such that the rank decreases for every stuttering step.

V. RESULTS

The capacity of the equivalence checker is demonstrated using 24 elastic circuits. The circuits are based on the elastic 5-stage processor shown in Figure I. The circuits were obtained by varying the size of the datapath and the number and location of additional buffers/relay stations in the data path. A maximum of 5 additional buffers were used. Verification statistics are shown in Table IV. The benchmark names are of the form "E-m-n" or "EB-m-n". "E" indicates that the circuit was proved correct and "EB" indicates a buggy circuit. "m" indicates the size of the data path and "n" is the number of additional buffers in the elastic circuit. We are not aware of any other equivalence checker that can verify the equivalence of elastic circuits and their synchronous parents. Therefore, we do not have another tool to quantitatively compare the efficiency of our results.

To demonstrate the results when checking buggy versions of elastic circuits, we developed two buggy versions of the elastic processor. B1 has a data path bug in its forwarding logic. The address of source register 1 in the execute stage is compared with the destination register address of the memory stage, when instead the address of source register 2 should be compared. B2 has a bug in the elastic controller, where the

controller for the *de* elastic buffer receives its valid input from the fetch stage instead of the decode stage.

Verification was performed on a 2.1GHz AMD (R) Athlon (TM) 2700+ CPU with a 256 KB L1 cache. The SMT solver used was Z3 [16]. The equivalence checker run time is the time taken to parse the input circuits and generate the proof obligations in SMT. The SMT time is the total time required to check all the proof obligations corresponding to the verification of the benchmark. The SMT memory is the maximum memory required for checking all the proof obligations of that benchmark.

VI. Related Work

Carloni et al. [2] developed a theory for Latency-Insensitive design. In this work, they introduced latency equivalence, a notion of correctness that can be used to design latency insensitive systems in a correct-by-construction compositional manner. Li et al. [17] have used property-based verification to check latency equivalence, liveness, and storage capacity for three latency-insensitive designs. Suhaib et al. [18] have developed a general property-based and simulation-based validation framework that can be used with a large number of LI design methods. Their approach is also based on latency equivalence. Another approach to property-based verification of elastic systems is based on static data flow structures (SDFS) [19]. SDFS can be used to model synchronous and asynchronous elastic systems. The SDFS models are translated to petri-net models that are amenable to analysis and by model checking tools. In contrast, our contributions are in equivalence checking for elastic circuits. In general, simulation-based validation, property-based verification, and equivalence verification compliment each other very well as can be witnessed in commercial design cycles. Also, our equivalence framework does not require additional properties as the synchronous circuit is the specification.

Cortadella et al. [4] have verified the elastic controller implementations against a high-level specification that describes how these controllers should behave. Kristic et al. [9] have shown that additional buffers (empty buffers) can be inserted in the datapath without altering the functionality of the design. Synthesis approaches based on correct-by-construction transformations ensure that the synthesis methods are reliable. However as noted earlier, the synthesis process and any further modifications to the circuit can introduces bugs. The target of our equivalence verification framework is to catch these bugs.

Srinivasan et al. [20] have developed a refinement-based verification method for elastic circuits. They provide methods and rules to construct equivalence proofs between pipelined elastic circuits and their synchronous parents. Their proofs are constructed manually. Our equivalence checker builds on this work. Following are the novelty of our work, over and above what was presented in [20]. (1) We have formally defined token-flow diagrams and developed procedures to automatically derive token-flow diagrams for elastic circuits. (2) We have generalized the algorithms to be applicable to any circuit structure. In [20], the approach was applicable only to linear pipelines. (3) Generalization required formalizing the concept of reference-points and incorporation of the use of reference-points in the algorithm to compute refinement maps. (4) The computation of token-flow diagrams and refinement maps is fully automated and implemented in a tool, whereas previously all of this was done manually. (5) Completeness result for the algorithm that computes refinement maps is derived. (6) The tool has been successfully applied to elastic circuits with as many as 0.5M gates.

VII. Conclusions

We have presented an automated equivalence checker that can verify elastic circuits against their synchronous parent circuits. The efficiency of the tool was demonstrated using 24 elastic VHDL benchmarks. The most complex elastic circuit verified has over 0.5**M** gates and over 3,500 latches. We believe that this equivalence checking technology can have a positive impact on the use of latency insensitive design in commercial design cycles, especially in the context of IP reuse to deal with timing issues.

There are several areas for future progress. Current algorithms are limited to dealing with deterministic elastic controller networks. We plan to extend the methods to deal with non-deterministic behavior of the elastic controllers as seen in open circuits, and circuits with variable latency units. Currently our applications are limited to closed circuits. Another area of future work is to explore the use of automated abstraction techniques to improve the scalability and capacity of the tool.

References

[1] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Latency insensitive protocols. In *CAV*, pages 123–133, 1999.

[2] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 20(9):1059–1076, 2001.

[3] L. P. Carloni and A. L. Sangiovanni-Vincentelli. Coping with latency in soc design. *IEEE Micro*, 22(5):24–35, 2002.

[4] M. R. Casu and L. Macchiarulo. Adaptive latency-insensitive protocols. *IEEE Design and Test of Computers*, 24:442–452, 2007.

[5] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *DAC*, pages 657–662, 2006.

[6] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[7] M. Galceran-Oms, A. Gotmanov, J. Cortadella, and M. Kishinevsky. Microarchitectural transformations using elasticity. *ACM Journal on Emerging Technologies in Computing Systems*, 7:18:1–18:24, Dec. 2011.

[8] G. Hoover and F. Brewer. Synthesizing synchronous elastic flow networks. In *DATE*, pages 306–311. IEEE, 2008.

[9] S. Krstic, J. Cortadella, M. Kishinevsky, and J. O'Leary. Synchronous elastic networks. In *FMCAD*, pages 19–30. IEEE Computer Society, 2006.

[10] C.-H. Li and L. P. Carloni. Leveraging local intracore information to increase global performance in block-based design of systems-on-chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(2):165–178, 2009.

[11] C.-H. Li, R. L. Collins, S. Sonalkar, and L. P. Carloni. Design, implementation, and validation of a new class of interface circuits for latency-insensitive design. In *IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2007)*, pages 13–22. IEEE, 2007.

[12] P. Manolios. Correctness of pipelined machines. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design–FMCAD 2000*, volume 1954 of *LNCS*, pages 161–178. Springer-Verlag, 2000.

[13] P. Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, August 2001. See URL `http://-www.cc.gatech.edu/~manolios/publications.html`.

[14] P. Manolios. A compositional theory of refinement for branching time. In D. Geist and E. Tronci, editors, *12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003*, volume 2860 of *LNCS*, pages 304–318. Springer-Verlag, 2003.

[15] SMT-LIB, 2012. See URL `http://www.smtlib.org/`.

[16] D. Sokolov, I. Poliakov, and A. Yakovlev. Analysis of static data flow structures. *Fundam. Inform.*, 88(4):581–610, 2008.

[17] S. K. Srinivasan, Y. Cai, and K. Sarker. Refinement-based verification of elastic pipelined systems. *IET Computers & Digital Techniques*, 6(2):136–152, 2012.

[18] S. Suhaib, D. Mathaikutty, D. Berner, and S. Shukla. Validating families of latency insensitive protocols. *IEEE Transations on Computers*, 55(11):1391–1401, 2006.

[19] Verific Design Automation, Inc., 2012. See URL `http://-www.verific.com/`.