# Data Path Optimisation and Delay Matching for Asynchronous Bundled-Data Balsa Circuits

Norman Kluge
*Hasso-Plattner-Institut*
Potsdam, Germany
*IHP*
Frankfurt (Oder), Germany
norman.kluge@hpi.de

Ralf Wollowski
*Hasso-Plattner-Institut*
Potsdam, Germany
ralf.wollowski@hpi.de

***Abstract*—Balsa provides an open-source design flow where asynchronous circuits are created from high-level specifications, but the syntax-driven translation often results in performance overhead. To improve this, we exploit the fact that bundled-data circuits can be divided into data and control path. Hence, tailored optimisation techniques can be applied to both paths separately. For control path optimisation, STG-based resynthesis has been used (applying logic minimisation). To continue the investigation, we additionally apply synchronous standard tools to optimise the data path. However, this removes the matched delays needed for a properly working bundled-data circuit. Therefore, we also present two algorithms to automatically insert proper matched delays. Our experiments show a performance improvement of up to 44 % and energy consumption improvement of up to 60 % compared to the original Balsa implementation.**

## I. Introduction

Nowadays, most circuits are designed using the synchronous design paradigm. There are many mature and established tools helping the designer to develop such circuits properly. However, asynchronous designs might be considered when a circuit with a specific property is needed, e.g. a power efficient one [1]. Unfortunately, there are only a few CAD tool suites providing a design flow starting with a high-level language specification down to silicon (and most of them are proprietary).

*Balsa* [2] provides an open source design flow, where asynchronous circuits are created from high-level specifications. The *Balsa compiler* converts a high-level *Balsa* program (see Listing 1 for an example) into a network of handshake (HS-)components. These basic building blocks form the circuit and communicate via asynchronous handshakes. However, the syntax-driven translation used by the *Balsa compiler* often results in performance overhead. To improve the performance, one can utilise the fact that bundled data circuits can be divided into data and control path. Hence, tailored optimisation techniques can be applied to both paths separately. For control path optimisations, Signal Transition Graph (STG) based (re)synthesis has been introduced e.g. in [3]–[6], applying logic minimisation methods to (parts of) the control. In [3] a SpecC specification is transformed into tailored Balsa code, from which STGs are generated directly using the early acknowledge HS-protocol. The other approaches are working with a (Breeze) netlist of HS-components communicating via the original HS-protocol (resynthesis): In [4], only HS-components modelling pure control were resynthesised. In [5] and [6], mixed HS-components (consisting of control and data path) were also considered. In contrast to [6], in [5] the control of all components used by *Balsa* were resynthesised.

[7] enhances the most comprehensive resynthesis approach of [5] proposing a design flow which tackles state explosion with STG decomposition and resolves reset problems of established logic minimisation tools with a new logic synthesiser. Promising results are presented concerning performance improvement. Additionally, [7] presents first experiments showing the significant further potential to improve the data path *en bloc* using synchronous optimisation tools. However, these circuits are not guaranteed to work properly, because all matched delays, needed for a properly working bundled-data circuit, were destroyed.

In this paper, we extend the proposed design flow from [7] as follows: To overcome the problem of improper matched delays, we apply data optimisation on each component separately (instead of en bloc for the entire data path), enabling systematic methods for the proper insertion of matched delays. The obvious approach is a delay matching algorithm which measures the delay of the data path (of each component) and inserts a delay of the same value on the control path (of the same component). This algorithm can be applied on all Balsa-based bundled-data circuits. However, the results of this algorithm are unimpressive. Considering the architecture of the resynthesised circuits, especially the delay caused by the control path, we designed a second delay matching algorithm yielding much better results than the simple (but more general) one. Interestingly, in most cases no matched delay is needed, improving most results of the measured physical quantities (like performance, area, energy) substantially.

The paper is organised as follows: in the next section we summarise essential basic notions in the context of resynthesis. Section III describes the overall design flow and the therein integrated resynthesis flow as an overview. In Sections IV and V we discuss our ideas regarding data path optimisation and delay matching, respectively. In Section VI we show the execution of one of our delay matching algorithms by an example. We discuss our implementation of the proposed methods in Section VII. The post-layout simulation results for several benchmarks are presented in Section VIII. We draw conclusion and future work in Section IX.

## II. Background

We assume the reader is familiar with Petri nets, STGs, their state graphs, SI Implementability, and STG decomposition, see [8], [9] for details. Nevertheless, some selected basic notions as well as the concept of STG decomposition are provided here.

### *Signal Transition Graphs*

A Signal Transition Graph (*STG*) is a Petri net that models the desired behaviour of an asynchronous circuit. An STG is a tuple $N = (P, T, W, l, M_N, In, Out, Int)$, where $(P, T, W, M_N, Sig^{\pm}, l)$ is a Petri net, $In$, $Out$ and $Int$ are disjoint sets of *input*, *output* and *internal signals*, and $Sig = In \cup Out \cup Int$ is the set of all signals;

*signature* refers to this partition of the signal set. $Sig^{\pm} = Sig \times \{+, -\}$ is the set of *signal edges* or *signal transitions*; its elements are denoted as $s+$, $s-$ resp. instead of $(s, +)$, $(s, -)$ resp. A plus sign denotes that a signal value changes from *low* to *high*, and a minus sign denotes the opposite direction. $l$ is the *labeling function* $l : T \mapsto Sig^{\pm} \cup \{\lambda\}$, associating each transition $t$ with one of the signal edges or with the empty word $\lambda$. In the latter case, we call $t$ a *dummy transition*; it does not correspond to any signal change. A transition $t$ is *enabled under a marking* $M$ if $\forall p \in {}^\bullet t : M(p) \geq W(p, t)$, which is denoted by $M[t\rangle$. An enabled transition can *fire* or *occur* yielding a new marking $M'$, written as $M[t\rangle M'$, if $M[t\rangle$ and $M'(p) = M(p) - W(p, t) + W(t, p)$, for all $p \in P$. A *transition sequence* $v = t_1 \dots t_n$ is *enabled under a marking* $M$ (yielding $M'$) if $M[t_1\rangle\ M_1[t_2\rangle \dots\ M_{n-1}[t_n\rangle M_n = M'$, and we write $M[v\rangle$, $M[v\rangle M'$ resp. The graphical representation of the STG is as usual; transitions labelled with an input signal edge have thick borders. The idea of *parallel composition* $N_1||N_2$ is that the two STGs run in parallel, synchronizing on common signals [10].

*SI Implementability*

A circuit is speed-independent if its fundamental behaviour does not depend on the delay of its gates [9, p. 83]. STGs are used for specifying the behaviour of speed-independent (SI) circuits. For an STG $N$, a *state vector* is a function $sv : Sig \mapsto \{0, 1\}$, assigning a boolean value to each signal. For an STG $N$ a *state assignment* assigns a state vector $sv_M$ to each marking $M$ of its reachability graph $RG_N$, forming the state graph. If there is a state assignment, an STG $N$ has the *complete state coding* property (*CSC*) if any two reachable markings $M_1$ and $M_2$ with the same state vector (i.e. $sv_{M_1} = sv_{M_2}$) enable the same output and internal signals. Otherwise, $N$ has a *CSC conflict* and no circuit can be synthesised directly. For logic synthesis, the entire state space of the STG must be explored, leading to state space explosion [11]. To tackle this problem, STG decomposition was introduced.

*STG Decomposition*

For the STG decomposition algorithm of [8], a *partition* of the output signals of the given STG $N$ is chosen, and the algorithm decomposes $N$ into component STGs, one for each set in this partition. For synthesis, equations for the outputs of each component are derived from the respective state graph, instead of deriving the equations from the overall state graph of $N$.

Very often, the cumulated states of all component state graphs yield a number much smaller than the state count of $N$, in which case the decomposition enables overcoming the state space explosion problem. Actually, it might already be beneficial if each state graph is smaller than the one of $N$, in particular for reducing peak memory usage. The decomposition is always correct, i.e. the parallel composition of the components matches the behaviour of $N$ – cf. [8].

*4-phase bundled data handshake protocol*

*Bundled data* describes a protocol where each bit of a data signal is encoded with one bit (in contrast to $n$-rail protocols) and is bundled with a request and an acknowledge signal. All three signals together form a *channel*. Depending on the flow direction of the data signals one can classify two types of channels: *push* (data flows from master to slave) and *pull* (other direction) channels. If there are no data signals, the channel is called a *sync* channel. Fig. 1 shows the proper signal transitions for a 4-phase protocol. For a *push* channel:

1) The master sets the request signal to high, after issuing the data
2) The slave sets the acknowledge signal to high after processing the data

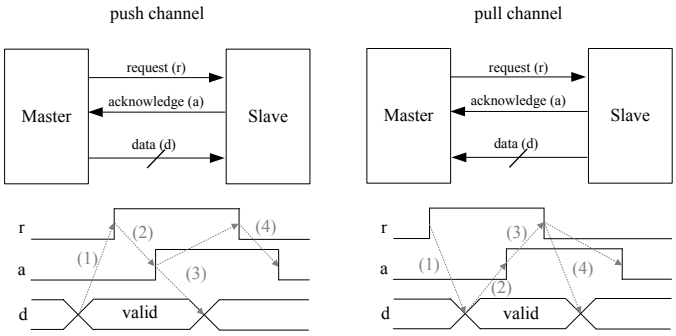


Fig. 1. 4-phase bundled data HS protocol

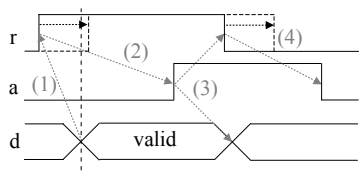


Fig. 2. Delay matching

3) The master might invalidate the data and sets the request signal to low
4) The slave sets the acknowledge signal to low

For a proper working circuit, it is important that these signal events are always happen in this order. Moreover, for step 1 it is crucial that issuing the data happens before setting request to high, because a high request indicates valid data. If the request would come first, the slave could start computation with false data or store false data. Fig. 2 shows such a false behaviour (in solid lines). Because an implementation of such a component does not guarantee the correct order of signals, one might insert delay for the request signal to ensure the correctness by shifting the positive edge of $r$ beyond the data validity mark (dashed lines), e.g. by inserting inverter chains[1]. Due to the SI implementation of the control circuit, the logic generating $a$ will wait for $r$. Hence, the following signals will adapt to the shifted signal. The process of shifting control signals to suit data timing is called *delay matching*.

## III. Resynthesis overview

We propose an adapted design system as shown in Fig. 3. Just like the original *Balsa* flow, we start with a Balsa program, which is transformed into a *Breeze netlist* with the Balsa compiler *balsa-c*. In our case, a Breeze netlist specifies a network of HS-components communicating via the 4-phase bundled data HS protocol. Afterwards, a Verilog netlist is generated from this Breeze file. In the original design flow this is done by *balsa-netlist* – we use the resynthesis tool from [7], which will do this step in an optimised manner. Using this netlist, the circuit is layouted (in the first attempt without any constraints) and a new pair of a Verilog Netlist and an SDF file is generated. Both are given to our new delay matching tool *ASGdelaymatch*, which checks matched delays for suited components, generating a constraints file. This process is repeated, as long as there are violations in the matched delays. The final netlist, its derived delay file and a testbench for the design are given to a simulator to undertake performance tests (discussed in Sec. VIII).

[1]Note that this will also shift the negative edge of $r$. However, there is no strict requirement to invalidate the data before $r$ falls

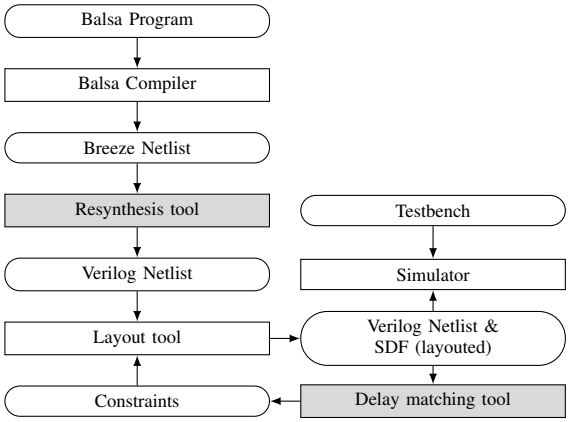


Fig. 3. Adapted resynthesis design system

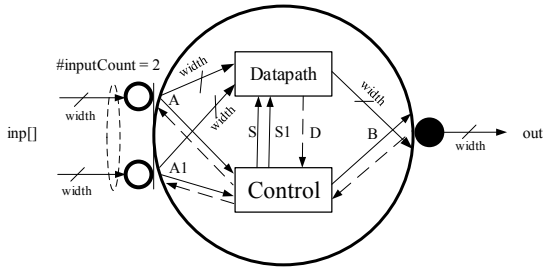


Fig. 4. Splitting of a *BrzCallMux* with scale factor $n = 2$

The main idea of the resynthesis approach is to separate the control part of the HS-components' behaviour from its data path. This results in a new internal structure, as shown in Fig. 4 for the *BrzCallMux* component. The *BrzCallMux* component has $n$ writing channels on the left and multiplexes these onto the channel on the right hand side. In our approach, we split the component into a data and a control part. All incoming and outgoing request (solid line) and acknowledge signals (dashed line) are connected to the control, while all data signals (bold solid line with width information) are connected to the data path. For an easier processing later on, we introduced new names for channels. The *inp*[] channel of the *BrzCallMux* is a scaled channel, meaning there might be $n$ (input) components connected to it, building an $n : 1$ multiplexer. Because this channel is scaled in the original definition, *A*, the corresponding channel in our implementation, is also scaled; in the example $n = 2$, resulting in channels *A* and *A1*. For the communication between the control and data path, we have to introduce new signals. In the case of the *BrzCallMux*, we need a scaled request-only channel *S* (for $n = 2$: *S* and *S1*) from control to data that states which channel was requested to multiplex and as answer an (unscaled) acknowledge-only channel *D* which states that the multiplexing (data) operation was done.

The general architecture of the resynthesis tool (*ASGresyn*) is shown in Fig. 5. The HS-components of the Breeze netlist are divided into pure control HS-components and mixed HS-components. The latter contain both control and data path (like the *BrzCallMux* component). HS-components like *Sequence* or *Concur* do not have any data lines attached, resulting in no need for a data path in our approach. However, the *Arbiter* component also doesn't have data lines, but needs a special part for making the decision (Mutex element) – this is done within the data path.

For the control part of every HS-component instance (pure control

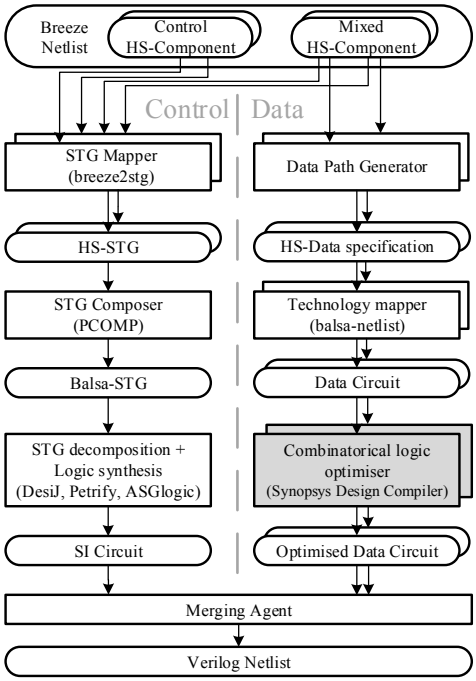


Fig. 5. Resynthesis tool (ASGresyn) architecture

and mixed) a behaviour equivalent HS-STG is constructed. The HS-STGs of all components are merged with optimised parallel composition [10]. The resulting STG represents the overall control behaviour of the circuit and is called *Balsa-STG*. Due to complexity problems, STG decomposition may be applied and an SI circuit is generated with a logic synthesis tool. For every mixed HS-component a corresponding data path is generated and optimised for suitable components (this procedure is discussed in the next Section). Both parts are merged and instantiated. The procedures concerning the control path and the initial generation of the data path are discussed in more detail in [7].

## IV. Data path optimisation

Because of the separation into control and data path in the resynthesis process, optimisations can be applied on both parts separately. All data parts of resynthesised circuits we have inspected are "just" combinatorial logic (except for latches), i.e. no asynchronous feedback loops are present. In this case, there is no difference to synchronous logic, thus we can apply established synchronous optimisation mechanisms (using established synchronous optimisation tools).

Because synchronous optimisation tools do not consider bundled-data requirements, they usually ruin matched delays. E.g. in some components, the control is delayed with two inverters to be slower than the data path. In this case, a synchronous tool will optimise that function by replacing the two inverters by a wire (zero delay)[2]. Thus, we have to introduce new proper matched delays after data path optimisation.

In general, this optimisation with synchronous tools may be applied to any data path in the resynthesised circuits. However, for many components this optimisation step yields no difference to the initial implementation (except for the delay matching logic), because the original implementation already was optimal. Thus, we only

[2]Note that neither Balsa nor the resynthesis tool produce a constraints file

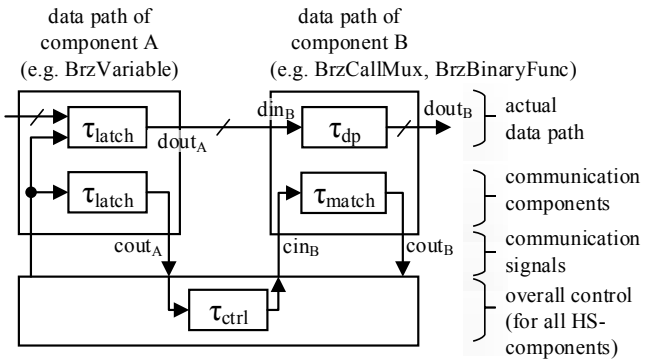


Fig. 6. Delay matching approach

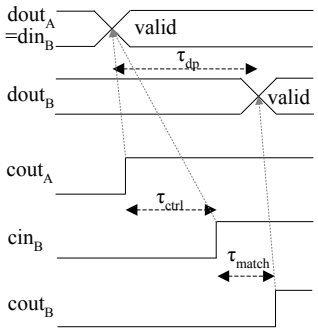


Fig. 7. Delay matching approach – timing

apply this optimisation to suitable components like *BrzBinaryFunc*, *BrzUnaryFunc*, *BrzCase* and *BrzWhile*.

Contrary to [7], this approach applies optimisation to data path for every component separately, which might result in a lesser performance improvement than applying it to the entire data path en-bloc. However, the en-bloc solution is not guaranteed to work properly, because of missing matched delays. Because the structure of the circuit is altered arbitrary, it is nearly impossible to find control paths and their (logically) corresponding data paths. Thus, delay matching can not be applied easily. However, keeping the structure by performing only local data path optimisation, delay matching can be applied in a systematic way.

## V. Delay matching

Because we are using bundled-data circuits, we have to ensure that the corresponding control signals are slower than the data signals. Up to now, most of Balsa's and our own data path implementations use predefined delay matching, i.e. the components have hard-coded delays (e.g. two inverters for a latch in *BrzVariable*). These delays are sometimes not sufficient (e.g. for very large *BrzCallMux* components [1]) because the data path is still slower than the control path. Additionally, after data path optimisations matched delays are often reduced to zero delay (cf. Section IV). Thus we need a proper matched delay insertion mechanism with dynamically generated delays.

Component B on the right-hand side in Fig. 6 shows the typical structure of the data path of a component. The actual data path has the input $din_B$ and the output $dout_B$ and connects them with combinatorial logic with a delay of $\tau_{dp}$. Note that in general data paths may have multiple inputs. However, for our considerations only the worst case path has to be examined. Additionally, there is the communication component of the data path, which handles communication with the control. In this case, this part has no connection to the actual data path. It consists of a combinatorial logic block with the input $cin_B$ and the output $cout_B$ and a delay of $\tau_{match}$. As stated above, after data path optimisation (using synchronous tools) $\tau_{match}$ often is reduced to zero. Fig. 7 shows the (possible) timing and data-validity declarations (in grey) between the signals.

For example, for the *BrzCallMux* from Fig. 4 $din_B$ is either the data signal(s) from channel $A$ or channel $A1$ (depending on which one is the slower one), $dout_B$ is the data signal(s) from channel $B$; $cin_B$ is channel $S$ or channel $S1$ (depends on the algorithm which to choose), $cout_B$ is channel $D$.

### A. Simple delay matching

To get a proper implementation for the bundled-data encoding we have to ensure that $\tau_{match} > \tau_{dp}$. Thus the value of $\tau_{match}$ depends on the value of $\tau_{dp}$. As far as we know, it is not possible to constrain a path with a relative time. Thus we have to split this process into three phases:

1) Measure the delay of the actual data path $\tau_{dp}$
2) Determine a suitable $\tau_{match}$
3) Set and enforce $\tau_{match}$

Because data paths are combinatorial, we can again use synchronous tools to measure $\tau_{dp}$. We have to measure the max delay from all input data wires to all output data wires to get the worst case path for the actual data path. The obvious approach for this simple delay matching algorithm is to enforce the min delay of the path from $cin_B$ to $cout_B$ ($\tau_{match}$) to be the measured $\tau_{dp}$. Unfortunately, the results for this approach (in conjunction with control resynthesis and data path optimisation) are too slow. In most cases the generated circuits are even slower than the original Balsa implementation (cf. Section VIII). However, this approach yields a properly delay matched circuit and is thus guaranteed to function (in terms of delay matching). Also, this approach works with plain Balsa as well as with resynthesised circuits.

### B. Control-aware delay matching

For most components we observed that the data signals are valid much earlier than the corresponding control signal. Beyond that, it is often valid even before the control request for the component has risen (In Fig. 7 $dout_B$ would be valid before the rise of $cin_B$). This is because most components consist of combinatorial logic only, without interaction between the actual data path and the communication component (as shown in component B in Fig. 6). Mostly, the source of the data is a memory component (e.g. *BrzVariable*). The computation starts when the data ($din_B$) is valid and is finished after $\tau_{dp}$ without waiting for the start signal from the control ($cin_B$).

Under the assumption that there are no wire delays (for post-synthesis considerations; we will extend this later for post-layout), $din_B = dout_A$; the computation in component B starts when $dout_A$ is valid (at time $t_1$). Thus $dout_B$ is valid after $\tau_{dp}$ (at time $t_3$), which marks the end of the computation (of the corresponding function $f$):

$$dout_B(t_3) = f(din_B(t_1))$$
$$t_3 = t_1 + \tau_{dp} \quad (1)$$

Because $dout_A$ is valid at time $t_1$, $cout_A$ eventually rises at time $t_2$ (We allow $t_1 = t_2$ even if this is not strict bundled-data):

$$t_1 \leq t_2 \quad (2)$$

After $cout_A$ has risen (at time $t_2$), $cout_B$ rises after $\tau_{ctrl} + \tau_{match}$ (at time $t_4$):

$$\begin{aligned} cout_B(t_4) &= cout_A(t_2) \\ t_4 &= t_2 + \tau_{match} + \tau_{ctrl} \end{aligned} \tag{3}$$

Signal $cout_B$ should rise (at $t_4$) after $dout_B$ is valid ($t_3$):

$$t_3 < t_4 \tag{4}$$

Substituting (1) and (3) into (4) yields:

$$\begin{aligned} t_1 + \tau_{dp} &< t_2 + \tau_{match} + \tau_{ctrl} \\ \tau_{match} &> \tau_{dp} - \tau_{ctrl} + (t_1 - t_2) \end{aligned} \tag{5}$$

Because of (2), $(t_1 - t_2) \leq 0$. Thus, we get the highest possible $\tau_{match}$ for $(t_1 - t_2) = 0$. Because we are using $\tau_{match}$ as a (best case) min delay, we can set (even if this is not the smallest possible solution[3]):

$$\tau_{match} > \tau_{dp} - \tau_{ctrl} \tag{6}$$

That means, we can subtract all the time for the actions of the control path that happen between the finishing of the writing of the data and requesting a computation result (that depends on this data).

To get an always working (safe) and minimal solution we need to determine the smallest possible $\tau_{match}$. Hence we have to determine a max $\tau_{dp}$ and a min $\tau_{ctrl}$. $\tau_{dp}$ is measured the same way as described above. For $\tau_{ctrl}$ we have to consider the underlying Balsa-STG. In this STG, we have to find all shortest *transition sequences* between transitions with label $cout_A-$ and $cin_B+$. We use the falling edge of $cout_A$ instead of the rising edge as starting point because for some components it is not yet ensured that the data path is valid when the control rises; for the falling edge it is[4]. Therefore we use this less optimal but safe variant.

To get these transition sequences we first apply some simple reductions like *FSP* and *FST* from [12] on the Balsa-STG to tackle state space explosion. Afterwards we construct the reachability graph and search for all markings $M$ ($M'$) where transitions with label $cout_A-$ ($cin_B+$) are enabled resp. $M_{start}$ ($M_{end}$) is the set of all $M$ ($M'$) resp. Then $\forall M \in M_{start}$ and $\forall M' \in M_{end}$ we search all transition sequences $v = t_1 \dots t_n$ where $t_1$ is a transition with label $cout_A-$ and $M[v\rangle M'$, yielding the set $V$. We only want the shortest transition sequences so we remove all $v'$ such that $\forall v, v' \in V$ with $v \neq v'$ and $v$ is a subsequence of $v'$. Finally we undo the reductions (FSP, FST) from step one[5].

We now measure for all $v \in V$ with $v = t_1 \dots t_n$ the min delay from $l(t_k)$ to $l(t_{k+1})$ with $1 \leq k < n$. Exceptions for these measurements are:

- $l(t_k)$ and $l(t_{k+1})$ represent communication signals for the same data path: Here we would measure another matched delay (or one we want to improve in the future). Therefore we assume these delays to be 0 (which is the best case).
- $l(t_k)$ and $l(t_{k+1})$ represent communication signals with the environment: Because we don't know how fast the environment will be, we also assume these are 0 (which is the best case).
- $l(t_{k+1})$ represents an (STG) internal signal: We might not be able to measure this, because this signal might not be visible at the circuit's interface. Thus, we omit this signal and measure from $l(t_k)$ to $l(t_{k+2})$ (if present).

For every transition sequence $v \in V$, we sum these times for all the transitions in sequence. For parallel parts we choose the largest of the sequential transition sums, because the faster one(s) have to wait for the slowest one at the point of synchronisation. For $\tau_{ctrl}$, we choose the minimal (best case) sum of all $v \in V$. If $\tau_{match}$ ($< \tau_{dp} - \tau_{ctrl}$) is negative, we *don't need a matched delay, because the control path is already slower than the data path*. This approach leads to much better results than the simple algorithm (cf. Section VIII).

In general, this approach may be applied to most data path components in the resythesised circuits. However, some components (like *BrzCase*) do not need a matched delay, because they either use *completion detection* or they have only incoming data but no outgoing data. In many other components the original matched delays are working fine (even for very large component instances). Hence, we currently only apply delay matching to the components *BrzCallMux*, *BrzBinaryFunc*, *BrzUnaryFunc* and *BrzBinaryFuncConstR*.

### C. Post-layout delay matching

To extend delay matching for the post-layout phases, we also have to consider wire delays. Thus, between the rise (or fall) of the signals of $dout_A$ and $din_B$ are a additional delays. However, for our approach we can just add the worst case wire delay to $\tau_{dp}$. Currently, we ignore wire delays on the control side, resulting in a higher $\tau_{ctrl}$ value and maybe in a higher $\tau_{match}$ value. However, in many cases $\tau_{match}$ is already smaller than 0.

## VI. Example: Modulo 10

Listing 1. Example Balsa program: mod10

```
1  procedure mod10(input  i: byte;
2                  output o: byte) is
3    variable a : byte
4  begin
5    i -> a;
6    loop while a >= 10 then
7      a := (a - 10 as byte)
8    end;
9    o <- a
10 end
```

Listing 1 shows a simple Balsa program: It calculates input modulo 10. The algorithm reads data on input $i$ into variable $a$, and subtracts 10 from it as long as it is larger than 10. The result is written on output $o$. Applying resynthesis with STG decomposition we get a structure as shown in Fig. 8. The control part is divided into four non-trivial components (S0, S1, S3, S5) and five trivial ones (consisting only of assign statements i.e. wires). The data path consists of six components; data path optimisation as described in Section IV can be applied on components #1, #4, #5, #10. However, only components #5 and #10 (both *BrzBinaryFunc*) are effected by these optimisations. Because the matched delays on (the unchanged) component #1 and #4 are still present and working, we do not apply delay matching on them.

However, the matched delay of component #5 is non-existent any more after data path optimisation. Thus we have to insert a proper one by applying the control-aware delay matching algorithm. First of all, we measured a max delay for $\tau_{dp}$ of 231.10 ps. To calculate a proper value for $\tau_{match}$, we need a value for $\tau_{ctrl}$. For this we need the starting and ending point for the transition sequences. The ending point is the requesting control signal for component #5, which is rD_5. For the starting point we need to check which data path component(s) are connected to component #5. In this case it is component #0 (storage of variable $a$) via the signal d3 and thus the starting point for the

[3]This might be used for further improvements

[4]This results from hand-made optimisations for some components

[5]We are working on a proof for correctness of this transition sequence extraction algorithm. However, the overall approach still works without this algorithm, but it increases the tool's runtime significantly

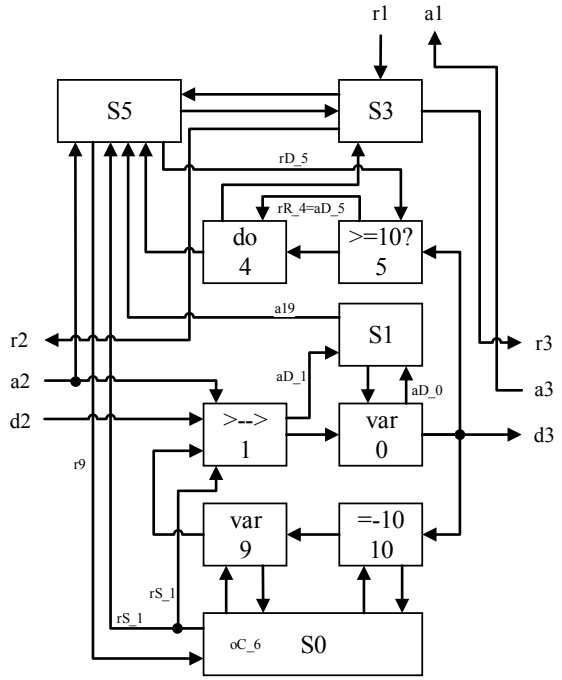


Fig. 8. Architecture of mod10 after resynthesis (with STG decomposition)

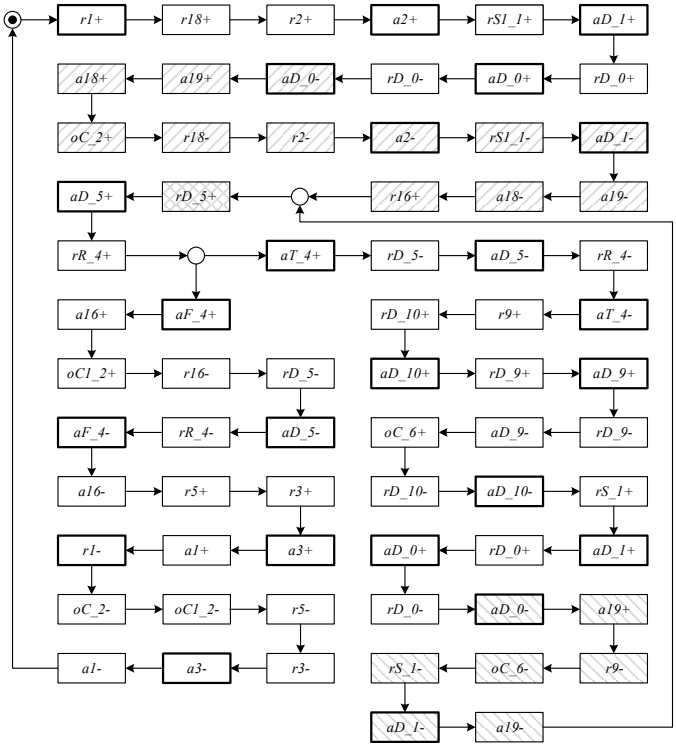


Fig. 9. Balsa-STG of mod10

transition sequence is its acknowledge control signal aD_0. Because variable *a* gets written at two different positions in the algorithm (at the beginning and in the loop), there are two transitions with label aD_0. Therefore we get two transition sequences after searching in the Balsa STG of mod10, which is shown in Fig. 9[6]:

1) aD_0−, a19+, r9−, oC_6−, rS_1−, aD_1−/2, a19−/2, rD_5+ ( )
2) aD_0−/2, a19+/2, a18+, oC_2+, r18−, r2−, a2−, rS1_1−, aD_1−, a19−, a18−, r16+, rD_5+ ( )

Resuming with sequence #1 we get the following measurements:

| | | | |
|---|---|---|---|
| aD_0− → a19+ | (S1) | 35.05 ps | |
| a19+ → r9− | (S5) | 114.62 ps | |
| r9− → oC_6− | (S0) | | Omit (internal) |
| r9− → rS_1− | (S0) | 150.11 ps | |
| rS_1− → aD_1−/2 | | 0 ps | Matched delay of component #1 |
| aD_1−/2 → a19−/2 | (S1) | 83.29 ps | |
| a19−/2 → rD_5+ | (S5) | 46.50 ps | |
| Sum | | 429.57 ps | |

The measurement for the second transition sequence yields 509.01 ps. Thus the first transition sequence is the faster one and therefore the best case. We use its time to calculate $\tau_{match}$:

$$\begin{aligned}\tau_{match} &> \tau_{dp} - \tau_{ctrl} \\ &> 231.10\,ps - 429.57\,ps \\ &> -198.47\,ps\end{aligned}$$

Thus the control path is already slower than the data path and no matched delay is needed. For component #10 we get the same outcome.

## VII. Implementation

We implemented the proposed methods into the ASG tool suite. The improved data path optimisation algorithm was implemented into *ASGresyn* (Fig. 5 shows the new part highlighted in light grey). We use the *Synopsys Design Compiler* as combinatorial logic optimiser.

For delay matching we created a new tool called *ASGdelay-match* implementing the simple and control-aware algorithm for post-synthesis and post-layout operations. For measurement of the paths we use the `report_timing`-command from either *Synopsys Design Compiler* or *Synopsys PrimeTime*. After checking the paths, we enforce the desired delays in the circuit by constraining the path with `set_min_delay`. For post-synthesis operations, we run *Synopsys Design Compiler* and enforce the delays by annotating the constraints and run the `compile`-command.

However, we observed a problem by only setting the min-delay to the matched delay path: This causes the *Design Compiler* to choose a gate which has at least this delay, but there is no constrained restriction on the upper bound. Therefore in most cases a single delay gate is chosen, because with no performance restrictions, area improvement seems to be favoured. E.g. let's assume we have delay gates with the delays 4, 8, 16 and 32. For a $\tau_{match}$ of 20, a delay gate with the value 32 is inserted. Setting a max delay for the path of 22, a 16 and a 4 delay gate is inserted, resulting in better performance but with negligibly more area consumption. Thus, we additionally set a max delay (with `set_max_delay`) of the matched path of $1.1 \times \tau_{match}$. However, if there is no solution between these min and max value, one may be violated. Unfortunately, *Design Compiler*

[6]Note that in this trivial example the STG is purely sequential, but Balsa-STGs in general specify concurrent behaviour.

tends to violate the min delay, which is unacceptable for our purpose. Hence, we implemented the algorithm to work iteratively, checking (and increasing) the delays until they are valid.

For post-layout, a full layouting step has to be done, using the given constraints (see Fig. 3). Because layouting takes much longer than running `compile` on *Design Compiler*, we first attempt post-synthesis delay matching to get initial constraints to reduce the number of layouting runs needed.

*ASGresyn* and *ASGdelaymatch* are both open source an can be downloaded on GitHub: https://github.com/hpiasg.

## VIII. Experimental results

Using the proposed flow in Fig. 3, we ran some benchmarks to measure the performance improvement. In this case we define performance as the latency time for a specific calculation example for a benchmark. All simulations are post-layout simulations, thus gate and wire delays are considered. We perform layouting with *Cadence Encounter*, disabling all improvement mechanisms. We do not yet consider (asynchronous) layout requirements like spatial proximity of gates belonging to the same component, or isochronic forks [9]. However, in all conducted experiments so far the result still works in simulation (even though we might violate these properties). We used a standard 130 nm technology from the IHP GmbH. This library does not contain any asynchronous-specific cells like C-Elements[7] or Mutexes.

First, we will describe how to read Table I: We compare the synthesis approaches *Balsa*, which represents the original Balsa implementation with our approach (*Resyn*). Using resynthesis, we may apply data path optimisation (*DPO*, *Yes*) or not (*No*). Additionally, *DPO* may be *Sub*, meaning that we apply subsequent data path optimisation: after resynthesis with control optimisation only, we mark all control components with `don't touch` and optimise the entire data path with *Design Compiler's* `compile`. The result is one block of data path, so no delay matching can be applied because the different paths are not distinguishable any more (cf. Section IV). Although the circuits are not guaranteed to work, they often show correct behaviour. We assume that these solutions, if they are working, yield the best possible performance we can get[8]. Column *DM* represents different delay matching approaches: No delay matching (*No*), the simple algorithm from Section V-A (*Simple*) and the control-aware delay matching algorithm from Section V-B (*Ctrl*). We compare the values of latency, area, average power, peak power and energy consumption. As baseline we always use the Balsa solution.

Our resynthesis implementation has more parameters than the ones shown in the table. Most important: applying STG decomposition with DesiJ [13] or not, using *Petrify* [14] or *PUNF/MPSAT* [15] to solve CSC, using *Petrify* or *ASGlogic* to perform logic synthesis. However, we will not discuss this in detail. Sometimes these parameters are yielding very different solutions (and therefore different results in the physical quantities), sometimes no solutions, or – even worse – not working solutions (e.g. due to Petrify's reset problem as described in [7]). Currently it is unpredictable which parameter combinations yield the best solution for optimisation towards a specific physical quantity.

Now we will discuss the results of Table I: Some parameter combinations are not useful and are not shown for all benchmarks. However, for *mod10* from Section VI the table shows all possible combinations. E.g. the application of delay matching on Balsa circuits and resynthesis circuits without data path optimisation is not useful for optimisation: They result in most cases in higher values for latency, area and energy. In most cases, delay matching is not needed for these circuits, because the initial ("hand-made") delays are working properly. However, in [1] it is shown that these initial delays may not be suitable for very large HS-components.

The comparison of the results of the Balsa synthesis and resynthesis with no data path optimisation and no delay matching shows the improvement applying control resynthesis only [7]: for the performance we get an improvement of about 12 %, for area of about 8 % and for energy also of about 12 %; for power we get a large variety of results: In some cases average power is getting worse when applying control resynthesis; peak power sometimes gets significantly worse. However, considering data path optimisation, we get more diversity in the results, depending on how many data path components are suitable for optimisation. For the *mult* benchmark, which implements a multiplier, we get a performance improvement of 7.3 % for control and 23 % for control and data optimisation. So, data path optimisation has a larger impact than control optimisation for this benchmark. In contrary, for the *counter* benchmark, the improvement of data path optimisation for post-layout simulations is negligible. However, during post-synthesis simulations, the improvement for both data path and control was up to 76.6 %. It is not yet clear, why this improvement is lost in the layout step.

Considering the area results, data path optimisation always has a larger impact on improvement than control resynthesis. The reason for that may be that in most Balsa circuits the data path has a larger share in area than the control path. The power and energy consumption values also have a larger decrease when applying data path optimisation compared to control optimisation only.

Comparing the results of the two delay matching algorithms, we see that the control-aware algorithm always yields a faster, smaller and a more energy efficient circuit. On the other hand, the simple algorithm always generates a circuit with a smaller average power. However, these are often not the same circuits since different synthesis parameters are used. Considering performance and area, we observed that for most components the control-aware algorithm does not insert matched delays ($\tau_{match} < 0$), whereas the simple algorithm naturally does. The gaps between the power values are a bit surprising for us. May be this is due to the fact that the control-aware algorithm performs the same operations in less time. Hence our hypothesis is that within a control-aware circuit more components are working in parallel. Concerning energy (which is the product of latency and average power), the control-aware algorithm gets better results because it is much faster but does not consume that much more power in relation.

## IX. Conclusion and future work

In this paper we added data path optimisation and delay matching to our resynthesis design flow. We applied local data path optimisation to HS-components by using established synchronous tools. We presented two delay matching algorithms, a simple obvious one and one considering the control delay happening between two data actions. The simple one yields impractical results, whereas the control-aware algorithm provides excellent improvement. Our experiments show a performance improvement of up to 44 % and energy consumption improvement of up to 60 % compared to the original Balsa implementation.

However, currently very large Balsa benchmarks can not be resynthesised. Logic synthesis tools are not yet capable of decomposing

[7] Instead of original C-Elements, we use set- and reset-dominant C-Elements/RS-Latches.

[8] We will try to reach these results in the future

TABLE I
RESULTS (POST-LAYOUT)

| Benchmark | Synthesis approach | | | Latency | | Area | | Avg. power | | Peak power | | Energy | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | DPO | DM | $ps$ | | $\mu m^2$ | | $\mu W$ | | $\mu W$ | | $pJ$ | |
| mod10 | Balsa | | No | 25,521 | 100.0 % | 2,749.3 | 100.0 % | 327.8 | 100.0 % | 2,397 | 100.0 % | 16.0 | 100.0 % |
| | Balsa | | Simple | 26,996 | 105.8 % | 2,816.6 | 102.4 % | 317.4 | 96.8 % | **2,364** | **98.6 %** | 16.2 | 101.5 % |
| | Resyn | No | No | 21,869 | 85.7 % | 2,568.7 | 93.4 % | 359.5 | 109.7 % | 9,541 | 398.0 % | 14.6 | 91.3 % |
| | Resyn | No | Simple | 74,462 | 291.8 % | 2,766.4 | 100.6 % | **115.4** | **35.2 %** | 2,614 | 109.1 % | 16.1 | 100.6 % |
| | Resyn | Yes | Simple | 29,639 | 116.1 % | 1,168.2 | 42.5 % | 132.2 | 40.3 % | 2,485 | 103.7 % | 7.3 | 45.7 % |
| | Resyn | Yes | Ctrl | **14,171** | **55.5 %** | **1,100.9** | **40.0 %** | 243.1 | 74.2 % | 2,432 | 101.5 % | **6.6** | **41.4 %** |
| | Resyn | Sub | No | *10,674* | *41.8 %* | *943.3* | *34.3 %* | 273.9 | 83.6 % | *1,857* | *77.5 %* | *5.6* | *35.1 %* |
| mult | Balsa | | No | 111,613 | 100.0 % | 8,172.8 | 100.0 % | 460.0 | 100.0 % | 3,103 | 100.0 % | 87.1 | 100.0 % |
| | Resyn | No | No | 103,416 | 92.7 % | 7,488.9 | 91.6 % | 422,0 | 91.7 % | 26,200 | 844.3 % | 75.7 | 86.9 % |
| | Resyn | Yes | Simple | 197,002 | 176.5 % | 5,769.3 | 70.6 % | **179.8** | **39.1 %** | **2,306** | **74.3 %** | **52.1** | **59.9 %** |
| | Resyn | Yes | Ctrl | **85,942** | **77.0 %** | **5,720.9** | **70.0 %** | 337.6 | 73.4 % | 3,605.7 | 116.2 % | 52.9 | 60.8 % |
| | Resyn | Sub | No | *78,715* | *70.5 %* | *4,735.7* | *57.9 %* | 283.5 | 61.6 % | 2,447 | 78.9 % | *41.1* | *47.3 %* |
| gcd | Balsa | | No | 1,164,546 | 100.0 % | 5,506.2 | 100.0 % | 328.0 | 100.0 % | 5,983 | 100.0 % | 401.1 | 100.0 % |
| | Resyn | No | No | 1,003,350 | 86.2 % | 5,025.8 | 91.3 % | 320.4 | 97,7 % | 11,000 | 183.9 % | 337.7 | 84.2 % |
| | Resyn | Yes | Simple | 2,397,486 | 205.9 % | 3,364.0 | 61.1 % | **74.7** | **22.8 %** | 3,782 | 63.2 % | 186.9 | 46.6 % |
| | Resyn | Yes | Ctrl | **663,787** | **57.0 %** | **3,118.1** | **56.6 %** | 225.1 | 68.6 % | **3,582** | **59.9 %** | **157.0** | **39.2 %** |
| | Resyn | Sub | No | *593,355* | *51.0 %* | *2,514.9* | *45.7 %* | 212.7 | 64.8 % | *2,844* | *47.5 %* | *132.8* | *33.1 %* |
| counter | Balsa | | No | 52,782 | 100.0 % | 1,612.0 | 100.0 % | 293.2 | 100.0 % | 1,541 | 100.0 % | 15.5 | 100.0 % |
| | Resyn | No | No | 46,580 | 88.2 % | 1,479.3 | 91.8 % | 302.4 | 103.1 % | 57,200 | 3,711.9 % | 14.1 | 91.0 % |
| | Resyn | Yes | Simple | 70,231 | 133.1 % | 1,072.0 | 66.5 % | **157.6** | **53.8 %** | 1,070 | 69.4 % | 11.1 | 71.5 % |
| | Resyn | Yes | Ctrl | **46,566** | **88.2 %** | **1,006.8** | **62.5 %** | 224.4 | 76.5 % | **898** | **58.2 %** | **10.4** | **67.5 %** |
| | Resyn | Sub | No | *43,656* | *82.7 %* | *803.1* | *49.8 %* | 207.4 | 70.7 % | *733* | *47.5 %* | *9.1* | *58.5 %* |

AND gates in all cases, thus we can not perform technology mapping for all logic functions. These tools need further improvement.

To improve the data path further, we may consider the optimisation in the subsequent variant (cf. rows with *Sub* in Table I) where no delay matching has been applied but the circuits are working properly anyway; beyond that they yield better results in most physical quantities. We want to explore possibilities to generate properly working circuits in a systematic way having quantities closer to these results.

Currently, we perform the layout step without considering requirements for asynchronous circuits, although all circuits work in the simulation already. We have to identify these requirements and implement the process.

## ACKNOWLEDGEMENTS

We thank Milos Krstic and Steffen Zeidler from IHP GmbH (Frankfurt/Oder) for their helpful advice. Best thanks to Walter Vogler and Stanislavs Golubcovs for fruitful cooperation making this work possible.

## REFERENCES

[1] S. Zeidler, M. Goderbauer, and M. Krstic, "Design of a low-power asynchronous elliptic curve cryptography coprocessor," in *Electronics, Circuits, and Systems (ICECS), 2013 IEEE 20th International Conference on*. IEEE, 2013, pp. 569–572.

[2] A. Bardsley and D. A. Edwards, "The Balsa asynchronous circuit synthesis system," in *Forum on Design Languages*, Sep. 2000.

[3] T. Yoneda, A. Matsumoto, M. Kato, and C. Myers, "High level synthesis of timed asynchronous circuits," in *Asynchronous Circuits and Systems, 2005. ASYNC 2005. Proceedings. 11th IEEE International Symposium on*, March 2005, pp. 178–189.

[4] F. Fernández-Nogueira and J. Carmona, "Integrated circuit and system design. power and timing modeling, optimization and simulation," L. Svensson and J. Monteiro, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. Logic Synthesis of Handshake Components Using Structural Clustering Techniques, pp. 188–198.

[5] S. Golubcovs, V. Walter, and N. Kluge, "STG-based resynthesis for balsa circuits," in *Proceedings of the 2013 13th International Conference on Application of Concurrency to System Design*, ser. ACSD '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 140–149.

[6] A. Alekseyev, I. Poliakov, V. Khomenko, and A. Yakovlev, "Optimisation of balsa control path using stg resynthesis," 21st UK Asynchronous Forum, 2009.

[7] N. Kluge and R. Wollowski, "Optimising bundled-data balsa circuits," in *2016 22nd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'16)*, May 2016, pp. 99–106.

[8] W. Vogler and B. Kangsah, "Improved decomposition of signal transition graphs," *Fundamenta Informaticae*, vol. 76, pp. 161–197, 2006.

[9] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, *Logic synthesis of asynchronous controllers and interfaces*, ser. Advanced Microelectronics. Springer-Verlag, 2002.

[10] A. Alekseyev, V. Khomenko, A. Mokhov, D. Wist, and A. Yakovlev, "Improved parallel composition of labelled Petri nets," in *Proceedings of the 2011 Eleventh International Conference on Application of Concurrency to System Design*, ser. ACSD '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 131–140.

[11] A. Valmari, "The state explosion problem," in *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the Volumes Are Based on the Advanced Course on Petri Nets*. London, UK, UK: Springer-Verlag, 1998, pp. 429–528.

[12] T. Murata, "Petri nets: Properties, analysis and applications." *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, April 1989.

[13] M. Schaefer, D. Wist, and R. Wollowski, "Desij–enabling decomposition-based synthesis of complex asynchronous controllers," in *Application of Concurrency to System Design, 2009. ACSD '09. Ninth International Conference on*, July 2009, pp. 186–190.

[14] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Transactions on Information and Systems*, vol. E80-D, no. 3, pp. 315–325, 1997.

[15] V. Khomenko, M. Koutny, and A. Yakovlev, "Detecting state coding conflicts in stg unfoldings using sat," in *Application of Concurrency to System Design, 2003. Proceedings. Third International Conference on*, June 2003, pp. 51–60.