STG-Based Resynthesis for Balsa Circuits

Stanislavs Golubcovs[1], Walter Vogler[1], Norman Kluge[2]
[1]Institut für Informatik, Universität Augsburg, Germany
[2]Hasso-Plattner-Institut (HPI), Universität Potsdam, Germany

***Abstract*—Balsa provides a rapid development flow, where asynchronous circuits are created from high-level specifications, but the syntax-driven translation used by the Balsa compiler often results in performance overhead. To reduce this performance penalty, various control resynthesis and peephole optimization techniques are used; in this paper, STG-based resynthesis is considered. For this, we have translated the control parts of *all* components used by the Balsa compiler into STGs. A Balsa specification corresponds to the parallel composition of such STGs, but this composition must be reduced. We have developed new reduction operations and, using real-life examples, studied various strategies how to apply them.**

I. Introduction

Asynchronous circuits are an alternative to synchronous circuits. They have no global clock signal, which results in lower power consumption and electromagnetic emission. The absence of global timing constraints allows greater tolerance in voltage, temperature, and manufacturing process variations [1].

Unfortunately, asynchronous circuits are more difficult to design due to their inherent complexity. They can be specified with *Signal Transition Graphs* [2] (STGs), which are Petri nets where the transitions are labelled with *signal edges* (changes of signal states); however, the rapidly growing state space of these models quickly overwhelms any designer (trying to design a circuit) or even STG synthesis tool (trying to synthesize a circuit from an STG).

An alternative to direct STG synthesis is *syntax-directed translation* from some high-level hardware specification into an asynchronous circuit without analysis of the state space. This transformation is provided by hardware compilers such as Balsa [3] and Tangram [4]. The Balsa compiler converts the high-level *Balsa programme* into a network of *handshake (HS-)components* (Figure 1 shows the standard Balsa design flow on the left); these basic building blocks form the circuit and communicate via asynchronous handshakes, where each handshake is a communication channel C utilising the *request* and the *acknowledge signals* rC and aC to synchronize the components or to transfer some data. Unfortunately, this approach introduces significant performance overhead due to excessive handshake overhead and signal over-encoding.

In more detail, Balsa synthesis relies on a limited number of HS-components, each having an optimized implementation in the hardware description language Verilog. When translating a Balsa programme, the HS-components are instantiated by providing the actual channel names; the resulting *Breeze components* form a network where two of them are connected if they share a channel. The Breeze components are listed in the so-called *Breeze netlist*.

This research was supported by DFG-project 'Optacon' VO 615/10-1 and WO 814/3-1.

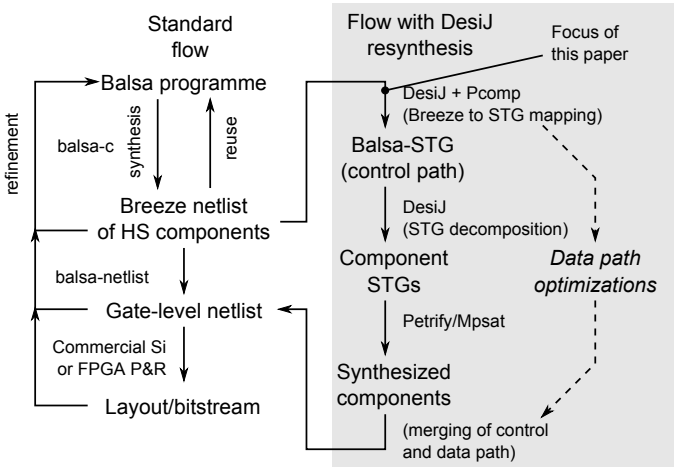


Figure 1: Resynthesis with DesiJ decomposition

The idea of STG-based *resynthesis* (see e.g. [5]) is to translate each Breeze component into a *Breeze-STG* with the same behaviour, cf. Figure 1 r.h.s. Corresponding to the network of Breeze components, the Breeze-STGs are composed in parallel resulting in the *initial Balsa-STG*: if two Breeze-STGs share a channel, they synchronize on (the edges of) the respective request and acknowledge signals – equally labelled transitions are merged and their signals are regarded as *internal* (i.e. as invisible for the environment of the circuit).

To solve the problem of over-encoding, we want to get rid of these internal signals. First, we turn the labels of the merged transitions into λ – this is called *lambdarization* and the transitions are then called *dummy transitions*. Second, *reduction operations* are applied as they are also used for STG-decomposition [6], [7]: dummy transitions are contracted, unnecessary places are removed, and also other net transformations may be performed.

We call the final result *Balsa-STG*; it is the input for some STG-specific synthesis methods like Petrify [8], Punf/MPSAT [9], ILP methods [10], or STG-decomposition with DesiJ [11] (ordered by increasing size of STGs the method can deal with). Since size matters so much, it is important to make the Balsa-STG really small; in particular for STG-decomposition, it is advantageous to get rid of all dummy transitions one way or the other. We remark that, in *STG-decomposition*, copies of the Balsa-STG are created, suitable

transitions are turned into dummies, and then essentially the same reduction operations are applied; the resulting components (the *component STGs*) can then be synthesized by other methods.

To realize resynthesis, one needs in principle a behaviour-equivalent *HS-STG* for each HS-component. These STGs can then be instantiated according to the Breeze netlist, yielding the Breeze-STGs. But some HS-components deal with data, while STGs can only deal with the control path of a design. Therefore, it has been suggested to create the Balsa-STG not from the full Breeze netlist but from a *cluster*, a connected subgraph of the respective network, consisting only of some types of pure control Breeze HS-components [12].

This has some problems: since the Breeze netlist describes a speed-independent circuit, the (full) initial Balsa-STG is deterministic and output-persistent, and the respective parallel composition is free of *computation interference*, see e.g. [13]. The cluster-based approach may miss some context-components, which usually restrict the behaviour. First, this might lead to conflicts between internal signals such that a speed-independent circuit cannot be synthesized. Of course, one tries to get rid of internal signals, but even then the effect of these conflicts is not clear. More precisely, after lambdarization, the STG might violate *output-determinacy* and, thus, lack a clear meaning [7]. We *prove* here that the full initial Balsa-STG is output-determinate after lambdarization.

Second, the parallel composition for the cluster might have computation interference; thus, one cannot apply the optimized parallel composition of [14], which produces fewer places and makes transition contractions easier.

In the light of these observations, it is an important achievement that we have constructed HS-STGs for *all* HS-components and base our approach on the *full* Breeze netlist:

- For HS-components that deal with data, we have introduced new signals for communication between the data- and control-path of the final circuitry (cf. Figure 1), isolated the control-path of the components and translated them to STGs. This is similar to the treatment of arbitration, which is also "factored out" from the STG.
- To understand the behaviour of the HS-components, our main source are the high- and low-level behaviour expressions provided by Bardsley [15], [16]. We have implemented a translator from the former to the latter and then to STGs. In our restricted setting, the translation to STGs is not so difficult; cf. [17] for the treatment of a more general Petri net algebra.
- Some HS-components are actually scalable, i.e. they are families of components. We have added suitable operators for behaviour expressions such that each of these HS-components has one closed expression plus a scaling set. For instantiation, also a scaling factor is provided; replicating the signals in the scaling set, the expression is expanded automatically, and then turned into an STG.
- Unfortunately, the behaviour expressions in [15], [16] have their limitations. In some cases, they describe inconsistent (physically impossible) behaviour as already noticed in [12] or behaviour that is generally not expected for some components. Therefore, for each HS-component, we have considered the Verilog description produced by the Balsa tools. Such a description does not specify the expected behaviour of the component's environment: e.g. the Call component expects that it can deal with a call before receiving the next one. Thus, these descriptions alone are also not sufficient. But we have used them, on the one hand, to validate most of our expressions; on the other hand, in the remaining cases, we modified the low-level expressions such that they fit the Verilog description while making the environment assumptions indicated in the expressions of [15], [16].

With our translation from a Breeze netlist to Breeze-STGs, we can apply the idea of [14]: we use an optimized parallel composition, and we can enforce injectivity for the STGs beforehand to avoid complex structures in the initial Balsa-STG N. Additionally, we found that relaxing injectivity in N with *shared-path splitting* can lead to better reduction. Since we can prove that N after lambdarization is output-determinate, we can apply all and even more reduction operations than allowed for STG-decomposition (e.g. LOD-SecTC2 in [7]).

Deletion of redundant places is a standard reduction operation for STG-decomposition. Since repeated redundancy checks are costly, DesiJ in its standard setting only checks for the very simple shortcut places. Here, we introduce redundancy checks that are only performed on suitably chosen subgraphs. The experiments have shown that considering small subgraphs already helps to find most redundant places and is not very time consuming. We also introduce *merge-place splitting*, which can also be helpful when a dummy transition cannot be contracted due to structural reasons.

We have found that the combination of all our ideas gives the best results, considerably reducing the number of dummy transitions while only using safeness preserving contractions [7]. Finally, in all our realistic Balsa examples, dummies could be removed completely when applying more general contractions at the very end.

Basic notions are defined in Section II; the succeeding section describes how we constructed the HS-STGs, also explaining the behaviour expressions. Section IV is concerned with the Balsa-STG and our methods to construct it. The two sections after describe our strategy for using these methods and give experimental data supporting our approach. We end with a short conclusion.

II. Basic definitions

A *Signal Transition Graph* (STG) is a Petri net that models the desired behaviour of an asynchronous circuit [2]. An STG is a tuple $N = \langle P, T, W, l, M_N, In, Out, Int\rangle$ consisting of disjoint sets of places P and transitions T, the *weight function* $W : P \times T \cup T \times P \mapsto \mathbb{N}_0$, the *labelling function* $l : T \mapsto In\{+,-\} \cup Out\{+,-\} \cup Int\{+,-\} \cup \{\lambda\}$ associates each transition t with one of the signal edges (of an input, output or internal signal) or with the empty word λ. In the latter case,

we call t a *dummy transition*; it does not correspond to any signal change. We write $s\pm$ for a signal s, if we do not care about the direction of the signal edge. A marking (like the initial marking M_N) is a function $M : P \mapsto \mathbb{N}_0$ giving for each place the number of tokens on this place.

The preset of a node $x \in P \cup T$ is ${}^\bullet x \stackrel{\text{df}}{=} \{y \in P \cup T \mid W(y,x) > 0\}$, the postset of x is $x^\bullet \stackrel{\text{df}}{=} \{y \in P \cup T \mid W(x,y) > 0\}$. A place p is a *marked-graph (MG-)place* if $|{}^\bullet p| = 1 = |p^\bullet|$; it is a *choice place* if $|p^\bullet| > 1$ and a *merge place* if $|{}^\bullet p| > 1$. If $W(x,y) > 0$, we say there exists the *arc* xy; a *loop* consists of arcs xy and yx. Arcs can form paths; we call such a path from $p_1 \in P$ to $p_2 \in P$ *simple*, if all its nodes have single-element pre- and postsets except that there might be several "entry" transitions in ${}^\bullet p_1$ and several "exit" transition in $p_2{}^\bullet$.

A transition t is *enabled* at marking M ($M[t\rangle$) if $\forall p \in {}^\bullet t : M(p) \geq W(p,t)$. Then it can fire leading to the follower marking M' with $\forall p : M'(p) = M(p) + W(t,p) - W(p,t)$ ($M[t\rangle M'$). This can be generalized to transition sequences w as usual ($M[w\rangle$, $M[w\rangle M'$). If $M_N w\rangle M$ for some w, we call M *reachable* and w a *firing sequence*. An STG is *safe* if $\forall$ reachable M, $p \in P$: $M(p) \leq 1$. Two transitions $t_1 \neq t_2$ are *in conflict* under M if $M[t_1\rangle$ and $M[t_2\rangle$, but $M(p) < W(p,t_1) + W(p,t_2)$ for some p.

We can lift enabledness and firing to labels by writing $M[l(t)\rangle\rangle M'$ if $M[t\rangle M'$. Generalizing this to sequences, λ's are deleted automatically; if $M_N[v\rangle\rangle$, we call v a *trace*. An STG is *consistent* (which is usually required) if, for all signals s, in every trace of the STG the edges $s+$ and $s-$ alternate and there are no two traces where $s+$ comes first in the one and $s-$ in the other. The STG is *output-determinate* if $M_N[v\rangle\rangle M_1$ and $M_N[v\rangle\rangle M_2$ implies that, for every $s \in Out$, $M_1[s\pm\rangle\rangle$ iff $M_2[s\pm\rangle\rangle$.

The graphical representation of STGs is as usual; signal edges of outputs are blue and for inputs red and underlined; internal signals are never shown. MG-places are replaced by an arc from the transition in the preset to the one in the postset.

The idea of *parallel composition* $N_1 || N_2$ is that the two STGs run in parallel synchronizing on common signals; a precondition is that an internal signal of one is not a signal of the other. Let A be the set of common signals. For each $s \in A$, an occurrence of a edge $s\pm$ in one STG must always be accompanied by an occurrence of $s\pm$ in the other. Since we do not know a priori which $s\pm$-labelled transition of N_1 will occur together with some $s\pm$-labelled transition of N_2, we have to allow for each possible pairing. Thus, the *parallel composition* is obtained from the disjoint union of N_1 and N_2 by combining each $s\pm$-labelled transition of N_1 with each $s\pm$-labelled transition from N_2 if $s \in A$; the resulting transition has the same label. We can use this operation to model circuits that are connected on signals with the same name, in particular when building an initial Balsa-STG. In this case, we will assume here that N_1 and N_2 have neither an input nor an output in common. Thus, $s \in A$ is an input of one and an output of the other; we define generally that such an s is internal in the composition. We will also use $||$ in behaviour expressions; in [15], [16], the operands never have a signal in common. For constructing STGs in this context, we found it useful to allow $s \in A$ being a common input or a common output, and such an s remains in input, output resp. See [6] for a formal definition in a related setting.

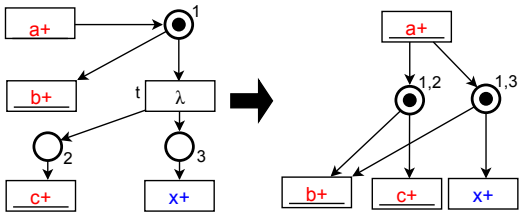


Figure 2: t-contraction

A most important operation for our approach is transition contraction; see Figure 2 for a type-2 secure contraction, since it has no merge place in its postset (though there is a choice place in its preset).

Definition 1. Let N be an STG and t be a dummy transition such that t is not incident to any arc with weight greater than 1 and t is not on a loop. We define N' as the *t-contraction of* N as follows:

$$\begin{aligned} T' &= T \setminus \{t\} \\ P' &= \{(p,\star) \mid p \notin {}^\bullet t \cup t^\bullet\} \cup {}^\bullet t \times t^\bullet \\ W'((p_1,p_2),t') &= W(p_1,t') + W(p_2,t') \\ W'(t',(p_1,p_2)) &= W(t',p_1) + W(t',p_2) \\ M'((p_1,p_2)) &= M(p_1) + M(p_2) \end{aligned}$$

The sets of signals and the labelling for $t' \in T'$ remain unchanged. In this definition $\star \notin P \cup T$ is a dummy element used to make all places of N' to be pairs; we assume $M(\star) = W(\star,t') = W(t',\star) = 0$.

The contraction is called *secure* if $({}^\bullet t)^\bullet \subseteq \{t\}$ (*type-1 secure*) or ${}^\bullet(t^\bullet) = \{t\}$ and $M_N(p) = 0$ for some $p \in t^\bullet$ (*type-2 secure*). It is called (structurally) *safeness preserving* [7] if $|{}^\bullet t| = 1$ or $t^\bullet = \{p\}$ with ${}^\bullet p = \{t\}$ and $M_N(p) = 0$.

III. From behaviour expressions to STGs

A. Translations

In [15], [16], the behaviour of the 46 HS-components is described with high-level behaviour expressions. These are formed from channel names C with some operators, and their meaning is explained with a structural translation: each expression e is translated into two low-level ones $\triangle e$ and $\triangledown e$, describing the set and the reset phase of e. The phases are needed for the inductive definition, the final meaning of e is the sequential composition $\triangle e; \triangledown e$. Low-level expressions are formed from signal edges $rC+$, $rC-$, $aC+$, $aC-$ with some operators, using signs already used for high-level expressions.

We describe the syntax of expressions by listing the translations, giving in each case the name of the high-level operator.

channel: $\triangle C = rC+; aC+$
$\triangledown C = rC-; aC-$

sequential composition: $\triangle(e;f) = \triangle e; \triangledown e; \triangle f$
$\triangledown(e;f) = \triangledown f$
parallel composition: $\triangle(e||f) = (\triangle e; \triangledown e)||(\triangle f; \triangledown f)$
$\triangledown(e||f) = \lambda$
synchronized parallel composition: $\triangle(e,f) = \triangle e||\triangle f$
$\triangledown(e,f) = \triangledown e||\triangledown f$
enclosure: $\triangle(C:e) = rC+; \triangle e; aC+$
$\triangledown((C:e) = rC-; \triangledown e; aC-$
choice: $\triangle(e|f) = \triangle e|\triangle f$
$\triangledown(e|f) = \triangledown e|\triangledown f$
loop: $\triangle(\#e) = \#(\triangle e; \triangledown e)$
$\triangledown(\#e) = \lambda$

The low-level expressions are not explained, but their meaning is fairly intuitive. E.g., the meaning of channel C is a sequence of four signal edges, describing a 4-phase-handshake on C; this type of handshake is fixed by the translation. We understand these expressions by giving a natural translation to a class of STGs; such translations have been studied in the literature, see e.g. [17] for a thorough treatment. Each such STG has a set of initial and a set of final places. As a building block, such an STG has no tokens; for the final translation, one puts a token on each initial place, and this is the HS-STG sought after. When we consider this initial marking for an STG in our class and a reachable marking where all final places are marked, then all other places are empty; intuitively, the STG has reached the end of its behaviour.

A signal edge corresponds to an STG with one transition, which is labelled with the signal edge and has the only initial place in its pre- and the only final place in its postset.

For sequential composition ";", one replaces the final place set of the first and the initial place set of the second STG by their Cartesian product; each pair is a new place which inherits the adjacent transitions of each of its components. Thus, a behaviour of the first STG can be followed by a behaviour of the second. The initial set of the first STG is initial for the result, the final set of the second STG is final.

Parallel composition "||" is as defined above; the initial and the final set are formed as union of the initial sets, the final set resp., of the component STGs. The expression λ corresponds to a skip, its translation is a single place that is initial and final.

For choice "|", one replaces the two sets of initial places by their Cartesian product and the same for the final places; each pair is a new place which inherits the adjacent transitions of each of its components; the first product is the new set of initial places, the second the new set of final places. Thus, the first firing of a transition decides whether the behaviour of the first or the second STG is exhibited.

Finally, when the loop construct "#" is applied to an STG, one replaces the final place set and the initial place set by their Cartesian product; again, adjacent transitions are inherited; the product is the new set of initial places and the new set of final places. Now, the behaviour of the original STG can be repeated arbitrarily often.

It is well-known that the combination of the last two operators can lead to a problem (cf. Section 4.4.13 in [17]):

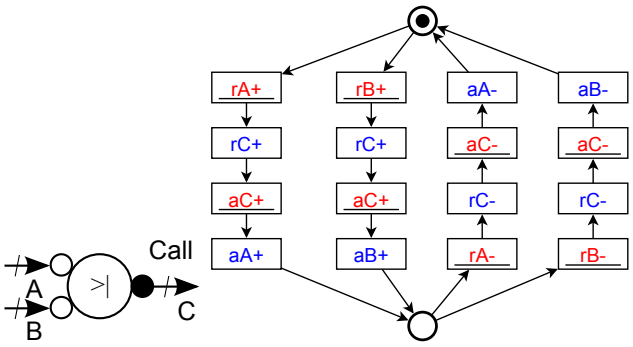


Figure 3: Inconsistent Call component

if one operand of a choice is a loop-STG, the loop can be performed a number of times and then the other operand can be chosen; this is presumably unexpected. Luckily, this never happens in our restricted context; almost always, a loop is the top operator or just inside a parallel composition.

Another potential problem is the following. It is often very desirable to work with safe STGs only, see e.g. [11]. All operators except loop generate a safe STG from safe operands. But if the operand of a loop is a parallel composition, the result might violate safeness (cf. Section 4.4.9 in [17]). Again, this situation does not turn up here.

All the translations have been implemented in DesiJ; they offer a flexible and very convenient way to produce interesting STGs. All HS-STGs are safe, hence also each initial Balsa-STG is safe.

As a simple example, we consider the Call component with channels A and B for input and C for output. The behaviour is defined with the high-level expression $Call = \#(A:C|B:C)$, which we expand as follows:

$$\begin{aligned}
\triangle Call &= \#(\triangle(A:C|B:C); \triangledown(A:C|B:C)) \\
\triangledown Call &= \lambda \\
\triangle(A:C|B:C) &= (rA+; rC+; aC+; aA+) \\
&\quad | \quad (rB+; rC+; aC+; aB+) \\
\triangledown(A:C|B:C) &= (rA-; rC-; aC-; aA-) \\
&\quad | \quad (rB-; rC-; aC-; aB-)
\end{aligned}$$

This in turn is translated to the STG in Figure 3 according to the description above; also see below.

B. Scalable HS-components

As mentioned in the introduction, some HS-components are scalable; the respective high-level expression could contain e.g. a term like $(B;C;D)||\ldots||(B(n-1);C;D(n-1))$. To have compact (and closed) expressions for such cases, we introduced new operators. An expression $\#||e$ for the *repeated parallel* $\#||$ is always accompanied by a scaling set, containing some channel names appearing in e. For the translation into a low-level expression, also a scaling factor n is given. Now, as a first step, $\#||e$ is expanded to a repeated parallel composition where the first operand is e and the others are replications

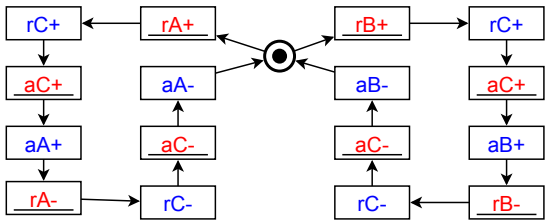


Figure 4: Consistent Call component

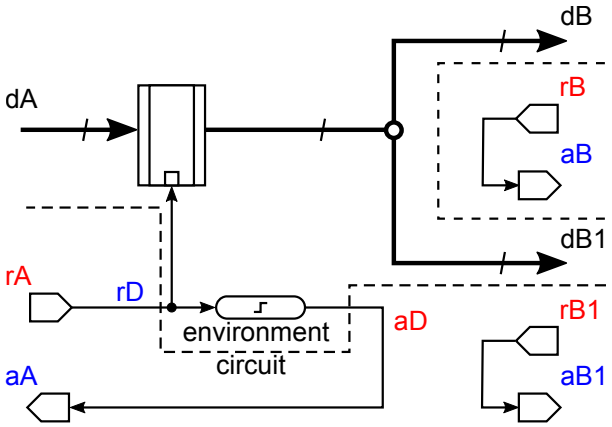


Figure 5: Gate-level implementation of Variable with two readers

of e; in the i-th replication ($i = 1, \ldots, n-1$), each channel name appearing in the scaling set is indexed with i. Then, the expanded expression is translated to low level and to an STG.

For the example mentioned, we would write $\#||(B; C; D)$ with scaling set $\{B, D\}$. For scaling factor 3, the expanded expression would then be $(B; C; D)||(B1; C; D1)||(B2; C; D2)$. Similarly, we have introduced *repeated choice* "$\#|$", *synchronized parallel* "$\#$," and *sequence* "$\#;$". $\#||e$ is also defined for a low-level expression e; this has to be expanded to another one without $\#||$ in the same way – see the treatment of Decision Wait below.

C. Some concrete HS-components

The STG in Figure 3 shows a loop with two choices in sequence. But actually, the second choice is already decided by the first; the way it is, the STG is not consistent and will be rejected e.g. by Petrify. (One could imagine a variant of Petrify, though, which simply ignores inconsistent behaviour since it is physically impossible.) We give a better expression for the intended behaviour; this is a low-level expression only and, as in a few other similar cases, we do not have a high-level one. It translates to the STG in Figure 4.

$$\begin{aligned} & \#((\triangle(A:C); \nabla(A:C)|(\triangle(B:C); \nabla(B:C))) \\ = \; & \#((rA+; rC+; aC+; aA+; rA-; rC-; aC-; aA-) \\ | \; & (rB+; rC+; aC+; aB+; rB-; rC-; aC-; aB-))) \end{aligned}$$

An example for scaling and for the communication with the data path is the *Variable* component, which allows storing

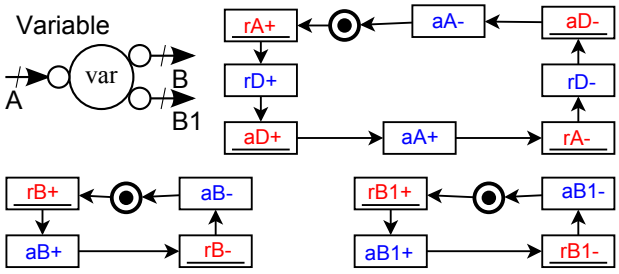


Figure 6: Variable with two read channels

information in its local memory elements. For better understanding, its gate-level model with support for two readers B and $B1$ is presented in Figure 5. The component uses the additional custom channel D: rD is the output signal controlling the memory latches when the data must be latched, whereas aD is the input from the environment. The path from rD to aD includes the rising edge delay line, which ensures that the data is latched before the component acknowledges its writer port on signal aA. The delay line cannot be synthesized, hence, it is factored out into the environment. The scale set is $\{B\}$ and the high-level expression is $\#(A : D)|| \; \#||(\#B)$; with scale factor 2, it translates to the following low-level expression and the STG in Figure 6:

$$\begin{aligned} & \#(rA+; rD+; aD+; aA+; rA-; rD-; aD-; aA-) \\ || \; & \#(\triangle B; \nabla B)||\#(\triangle B1; \nabla B1) \end{aligned}$$

Reader requests are always acknowledged as the latest data is always visible to them for reading; however, the environment of this component must ensure that reading and writing does not occur at the same time.

The *Decision Wait* component is another interesting example where a low-level expression is required. The original specification does not allow channel B to initiate communication, which is wrong. Based on the knowledge of its actual implementation, we realized that its actual definition should be described as follows. We also give its expansion for scale set $\{B, C\}$ and scale factor 2.

$$\begin{aligned} & \#(\#|(\triangle(B:C); \nabla(B:C))) \\ || \; & \#(rA+; \#|(\triangle C; aA+; rA-; \nabla C); aA-) \\ = \; & \#((rB+; rC+; aC+; aB+ \\ & ; rB-; rC-; aC-; aB-) \\ & |(rB1+; rC1+; aC1+; aB1+ \\ & ; rB1-; rC1-; aC1-; aB1-)) \\ || \; & \#(rA+; ((rC+; aC+; aA+; rA-; rC-; aC-) \\ & |(rC1+; aC1+; aA+; rA-; rC1-; aC1-)) \\ & ; aA-) \end{aligned}$$

In this example, the parallel composition has common signal C on both sides, which causes synchronization over transitions $rC+$, $rC-$, $aC+$, $aC-$, see Figure 7.

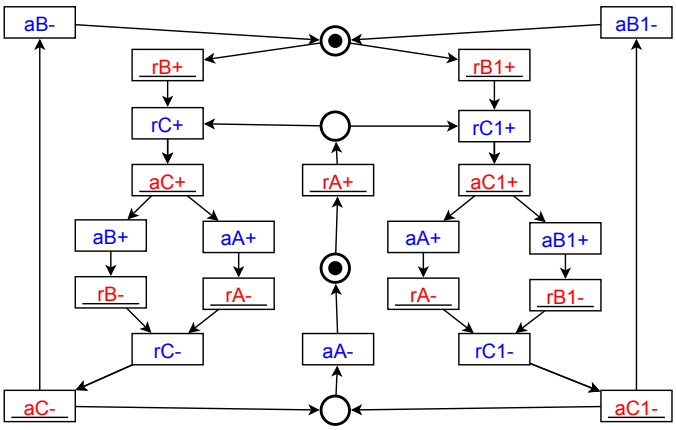


Figure 7: Correct Decision Wait

IV. Constructing the Balsa-STG

A. The initial Balsa-STG

For constructing an STG for the control part of a Balsa programme, we initialize HS-STGs according to the Breeze netlist and determine the parallel composition of these Breeze-STGs. As argued in the introduction, this composition is free of computation interference; hence, we can apply the methods from [14]: first, we can *enforce injectivity* and, second, we can apply the optimized parallel composition, which produces fewer places. Call the resulting initial Balsa-STG N'.

We will prove now that the STG N arising from lambdarizing the internal signals in N' is output-determinate, since this guarantees that N' has a clear meaning [7]; also, we know then a lot about behaviour-preserving reduction operations, see below. For the purpose of our proof, internal signals can be treated as output signals; so we can assume that all transitions in N' are labelled with (edges of) output or input signals, that N' is deterministic, and that no transitions in conflict under a reachable marking are labelled with outputs (output persistence). N is derived from N' by lambdarizing all transitions with label in some set of output signals.

Lemma 2. *$M_N[w_1\rangle M_1$, $M_N[w_2\rangle M_2$ with $l(w_1) = l(w_2) \implies \exists v_1, v_2, M : M_1[v_1\rangle M \wedge M_2[v_2\rangle M \wedge l(v_1) = \lambda = l(v_2)$*

Proof: If $|w_1| + |w_2| = 0$, all markings equal M_N and all sequences λ.

Assume that the lemma holds whenever $|w_1| + |w_2| = n$, and that we are given w_1 and w_2 with $|w_1| + |w_2| = n + 1$. There are three cases:

a) Assume $w_1 = w_1' t$ with $l(t) = \lambda$ and $M_N[w_1'\rangle M_1'[t\rangle M_1$. For w_1', M_1', w_2 and M_2, choose v_1', v_2' and M' by induction.

If $v_1' = vtv'$ with t not in v: Since t was labelled with an output, t is not in conflict with any transition in v, i.e. $M_1'[v\rangle$ and $M_1'[t\rangle$ implies $M_1'[tv\rangle$ and thus $M_1'[tvv'\rangle M'$. We can define $v_1 = vv'$, $v_2 = v_2'$ and $M = M'$.

Otherwise: As above, $M_1'[v_1'\rangle$ and $M_1'[t\rangle$ implies $M_1'[tv_1'\rangle M$, where $M'[t\rangle M$. We further define $v_1 = v_1'$ and $v_2 = v_2' t$.

b) analogous for w_2

c) Otherwise: $w_1 = w_1' t_1$, $w_2 = w_2' t_2$ and $l(t_1) = l(t_2) \neq \lambda$; we have $M_N[w_1'\rangle M_1'[t_1\rangle M_1$and $M_N[w_2'\rangle M_2'[t_2\rangle M_2$. For w_1', M_1', w_2' and M_2' choose v_1', v_2' and M' by induction.

Since all transitions in v_1' and v_2' were outputs, they are not in conflict with t_1 or t_2. By $M_1'[t_1\rangle M_1$ and $M_1'[v_1'\rangle M'$, we have $M_1'[v_1'\rangle M'[t_1\rangle M$ and $M_1'[t_1 v_1'\rangle M$. Similarly, $M_2'[v_2'\rangle M'[t_2\rangle$. Since t_1 and t_2 have the same label and N' is deterministic, we conclude $t_1 = t_2$, $M'[t_2\rangle M$ and $M_2'[t_2 v_2'\rangle M$. Thus, we can further define $v_1 = v_1'$ and $v_2 = v_2'$. ∎

Theorem 3. *N is output-determinate.*

Proof: Consider $M_N[w_1\rangle M_1[t\rangle$ and $M_N[w_2\rangle M_2$ with $l(w_1) = l(w_2)$ and $l(t) \in Out^{\pm}$. Take v_1, v_2 and M according to Lemma 2. As in the last subcase of the above proof, we have $M_1[v_1\rangle M[t\rangle$ due to $M_1[v_1\rangle M$ and $M_1[t\rangle$. Thus, $M_2[l(t)\rangle\rangle$ due to $M_2[v_2\rangle M[t\rangle$. ∎

In the full version of this paper, we will give an output-persistent parallel composition of Breeze-STGs without computation interference and a cluster of it such that the resp. smaller parallel composition has computation interference and is not output-persistent; cf. the discussion of the cluster-based approach in the introduction.

B. Reduction operations

To obtain the final Balsa-STG, we want to make the lambdarized initial Balsa-STG N smaller – and in particular, we want to remove as many dummy transitions as possible. For this, one applies reduction operations. Since N is output-determinate and according to [7], we can apply any language-preserving operation that turns an output-determinate STG into another one, which is then a so-called *trace-correct implementation*; this notion actually also allows some more operations. Observe that N is also consistent, and this must be preserved – which is the case for language-preserving operations.

The main operations are secure transition contraction and removal of redundant places. For the former, we mainly use the safeness preserving version, since it does not introduce so complicated structures that hinder further contractions. Contractions reduce the number of dummies, place removal keeps this and reduces the number of places. So far, this guarantees termination for the reduction phase.

We have found a new practical way to deal with redundant places, and we found two new reduction operations. We will present these now.

C. Removing redundant places based on a subgraph

Removing redundant places simplifies an STG and can enable additional secure contractions.

Definition 4. A place $p \in P$ is *structurally redundant place* [18] if there is a reference set $Q \subseteq P \setminus p$, a valuation $V : Q \cup \{p\} \to \mathbb{N}$ and some number $c \in \mathbb{N}_0$ satisfying the system of equations:

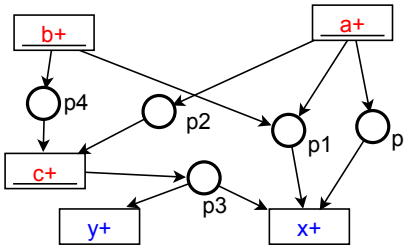


Figure 8: Checking place p for redundancy

1) $V(p)M_N(p) - \sum_{s\in Q} V(s)M_N(s) = c$
2) $\forall t \in T : V(p)(W(t,p) - W(p,t))$
   $- \sum_{s\in Q} V(s)(W(t,s) - W(s,t)) \geq 0$
3) $\forall t \in T : V(p)W(p,t) - \sum_{s\in Q} V(s)W(s,t) \leq c$

This property can be checked with a Linear Programming solver (LP-solver) where the values of V and c are the unknowns to be found. If p is structurally redundant, it can be removed without affecting the firing sequences of the STG. The main problem comes from checking STGs with large number of places because the inequations have to be solved individually for each of the places at least once.

Therefore, the standard of DesiJ was to only check for so-called shortcut and loop-only places with graph-theoretic methods. The new idea is to use an LP-solver on a small sub-STG. Checking a place p of N, the places of Q are most likely on paths from ${}^\bullet p$ to $p^\bullet$. Hence, we define the depth-n-STG as the induced sub-STG that has p and all places with distance at most $2 \cdot n - 1$ *from* some $t_1 \in {}^\bullet p$ and *to* some $t_2 \in p^\bullet$, and as transitions the presets of all these places plus $p^\bullet$. Places detected as redundant on such a sub-STG are indeed redundant:

Proposition 5. *Consider an STG N, a place p and some induced sub-STG N' containing p and ${}^\bullet p \cup p^\bullet$ and with each place also its preset. If p is redundant in N', it is redundant in N as well.*

Proof: Consider the reference set Q in N', the respective valuation $V : Q \cup \{p\} \to \mathbb{N}$ and $c \geq 0$ satisfying the conditions 1)–3) above in N'. Condition 1) clearly carries over to N. Conditions 2) and 3) have only to be checked for $t \in T - T'$ being adjacent to some $s \in Q \cup \{p\}$. By choice of N', $t \notin {}^\bullet p \cup p^\bullet$ and $\forall s \in Q : W(t,s) = 0$. Hence, the first product in both conditions is 0 and conditions 2) and 3) reduce to: $\sum_{s\in Q} V(s)W(s,t) \geq 0$ and $-\sum_{s\in Q} V(s)W(s,t) \leq c$; these clearly hold. ∎

Figure 8 demonstrates a simple example where place p is checked for redundancy. At depth 1, only place $p1$ is considered, it adds $b+$ to the equation and cannot prove p is redundant; $p3$ is not added at this depth because its distance from $a+$ is 3, and similarly for $p2$. At depth 2 places $p1$, $p2$, $p3$ are added to the equations; now p is seen to be redundant due to $p2$ and $p3$.

For completeness, we mention: $p^\bullet = \emptyset$ means the place is not affecting any transitions, hence it is redundant without any checks; if ${}^\bullet p = \emptyset$, building the subgraph we only consider the distance from a place to $p^\bullet$.

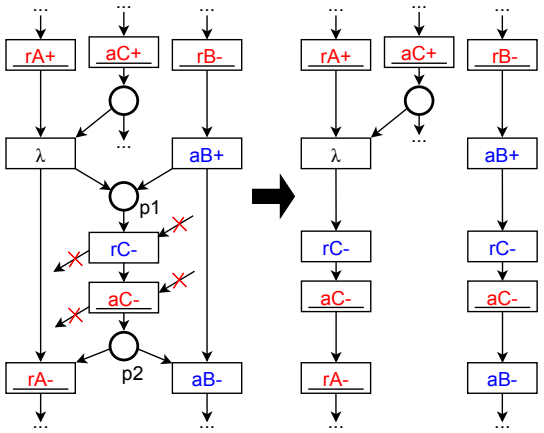


Figure 9: Shared-path splitting

Finally, a quick decision rule can be added with respect to the induced subgraph N': if $\exists t \in p^\bullet : |{}^\bullet t| = 1$, then p is not redundant in this subgraph and we can avoid the time consuming call of an LP-solver.

D. Splitting

When working with large STGs composed of Breeze-STGs, certain patterns occur quite often that block contractions. We propose two structural operations that simplify the STG structure and allow more dummy transitions to be contracted; both are based on the idea of splitting some transitions and places.

1) Shared-path splitting: The first example demonstrates a fairly common structure: a single simple path from p_1 to p_2 without dummy transitions is shared among two or more firing sequences (Figure 9). Each of the several entry transitions of p_1 (like the dummy) is connected to its own exit transition from p_2; the connection is an MG-place; all places considered are unmarked. Since the dummy cannot be contracted securely, we split the path as indicated and delete the marked-graph places, since they are redundant now. Then, the dummy can be contracted, which is even safeness preserving now if the dummy has only one postset place at this stage. If each splitting is followed by a contraction, the whole transformation reduces the number of dummies and termination for the Balsa-STG construction is guaranteed.

It may seem that this splitting may lose some firing sequences, for instance: $\ldots \lambda \to rC^- \to aC^- \to aB^+ \to aB^-$ is not possible after splitting. However: if a token is put onto the path, it must be removed before another token is put onto p_1; otherwise, we could have two tokens on p_1, violating the consistency of the STG.

The transformation can also be applied if, symmetrically, the dummy is an exit transition. Also, instead of the marked-graph place mentioned, there could be a longer simple path instead; if we keep this, the splitting is also correct.

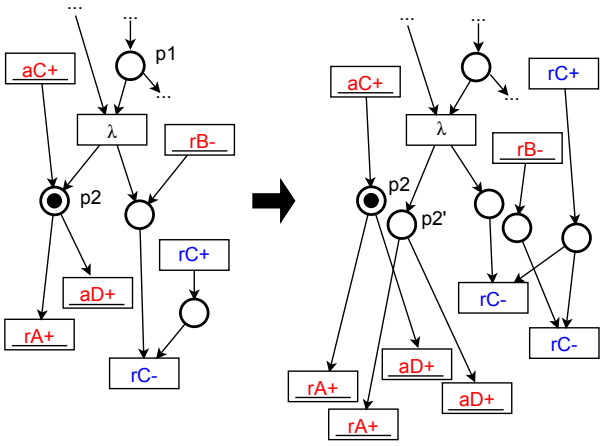


Figure 10: Merge-place splitting

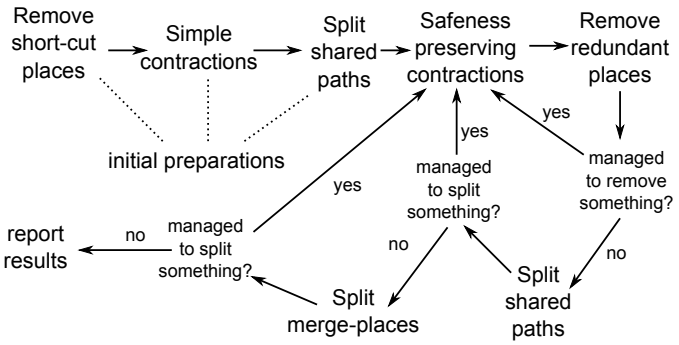


Figure 11: Elect reduction strategy

Path splitting is more or less the inverse of enforcing injectivity; in fact, it is often applied to paths that were introduced when enforcing injectivity before the parallel composition. Still, our results show: path splitting improves the results in any case, but path splitting with enforcing injectivity first is sometimes better than without.

2) Merge-place splitting: The second type of splitting directly addresses the case where a dummy transition t cannot be contracted securely because it has a choice place p_1 in its preset and some merge places like p_2 in its postset; cf. Figure 10. We can split off a new p_2' from p_2 (and similarly for all merge places) and replicate the transitions in $p_2^\bullet$ as shown: the replicates form $p_2'^\bullet$, while ${}^\bullet p_2'$ only contains the dummy; p_2 keeps all the other transitions of its preset and (in case) the tokens. We only apply this splitting, if each $t' \in p_2^\bullet$ satisfies $t^\bullet \cap {}^\bullet t' = \{p_2\}$, the resp. arc weight is 1, and the label is not λ. Then, the splitting does not change the behaviour; if we contract the dummy afterwards, the whole transformation again decreases the number of dummies, so termination is guaranteed.

This method is more general than shared-path splitting; however, the resulting STG structure is not as good for contractions. For instance, if we apply it to the earlier example in Figure 9, the dummy transition would have multiple places in both, its preset and postset, and contracting it would not be safeness preserving. Hence, merge-place splitting is more suitable as an "emergency plan" when other methods fail.

V. A strategy for STG-reduction with splitting

Based on experiments, we propose the strategy shown in Figure 11 for STG-reduction. We start with some initial preparations, first removing redundant places that can quickly be detected using graph-theoretic methods; this can enable more contractions later on. Next we perform simple contractions, treating a dummy transition t only if ${}^\bullet t = \{p\}$ for a non-choice place p and $t^\bullet = \{p'\}$ for a non-merge place p'. This operation shrinks the size of the STG without creating any new redundant places, and it guarantees that there are no dummies on any shared paths. Because it is so specific, it doesn't "spoil" the structure for the shared-path splitting.

Shared-path splitting is highly sensitive to the STG structure, so it is best to do it once before any more general safeness preserving contraction; in our examples, the overwhelming majority of path splits are carried out at this point.

Then, we repeatedly do the following: we apply safeness preserving contractions as often as possible; this may introduce redundant places, so we try to remove these, also using an LP-solver now. This in turn can enable new contractions.

If repeating these phases does not result in any progress, we try to split paths to enable further contractions. If this fails, we try splitting merge-places. In our experiments, these splits do not occur very often; if they also fail, the programme returns the resulting STG with its reduced number of dummies. Depending on the method for synthesizing a circuit from this STG, we can use general secure contractions (which may destroy safeness); this always worked in our examples. Alternatively, we can backtrack as in STG-decomposition: the remaining dummies originated from some internal signals; we can decide to keep these signals, i.e. we do not lambdarize them in the initial Balsa-STG and restart reduction (subject for future research).

VI. Experimental data

For testing our methods, we have chosen 5 reasonably large *realistic* Breeze file samples (Table I). These are the History Unit module HU from the Viterbi decoder, and some modules from the Samips processor presented in the table with the size of their lambdarized initial Balsa-STGs (number of arcs, places, transitions and λ-transitions). Note that, for the construction of these STGs, injective labelling and optimized parallel composition were used. All tests were run on a 32-bit Java platform, Intel I7-2600 CPU 3.40 GHz processor.

Since the initial preparations are quick, we also show the size of the STGs that enter the iteration: the numbers of dummies decrease, while the numbers of non-dummy transitions (difference between $|T|$- and $|\lambda|$-column) actually increase (due to shared-path splitting).

The first test concerns using an LP-solver for detecting further redundant places on subgraphs of varying depths (Figure 12a). We measured the overall time for the complete

Table I: Balsa benchmarks, initial sizes

	|arcs|	$\|T\|$	$\|P\|$	$\|\lambda\|$	|arcs|	$\|T\|$	$\|P\|$	$\|\lambda\|$
	initial				after preparations			
HU	5871	2612	2590	1886	2794	1246	1171	326
EX	8359	3611	3931	2980	3872	1569	1832	655
CP0	4813	2103	2182	1587	2248	1023	1032	240
RB	6632	2870	2929	2110	3419	1563	1523	317
D	4978	2045	2155	1482	2977	1259	1333	379

Table II: Final results when all optimization options are on

	|arcs|	$\|T\|$	$\|P\|$	$\|\lambda\|$	$\|P\|$	$\|\lambda\|$
					no init	
HU	2121	930	822	6	822	6
EX	2159	937	984	7	971	51
CP0	1700	789	745	4	750	8
RB	2804	1347	1253	7	1255	11
D	2046	886	872	5	824	27

reduction and the number of redundant places found by the LP-solver, and we added these up for our 5 examples. One sees that time increases with depth, and this is also true for each example separately. The first contributing factor here is the simple check filter, which prevents launching the LP-solver when the place obviously is not redundant. Its effect is drastic on small subgraphs; for instance for depth 1 of HU, the LP-solver is launched only 16 times. For depth 5, this number increases to 237, and to 473 for depth 15. The second factor is the size of the inequations; on larger subgraphs the solver becomes slower.

Figure 12a indicates that with depth 10 we rather quickly get a reasonable coverage of the redundant places that can be found at all with an LP-solver. Again, this is also true for each example separately. So we fixed this value for our further experiments.

Having fixed this value, computation time is not an issue anymore. What we consider now is the quality of the resulting STG, measured as the number of remaining dummies. Our experiments have confirmed the overwhelming success of the optimized parallel composition (PCOMP in [14]); so we always used this. Figure 12b-f show the results for each example when we use each of enforcing injectivity, the LP-solver and our splitting methods or not. It is clear that splitting should be used (lighter bars); also, the solver is useful in all cases. The case for enforcing injectivity is not so clear: in one case it really helps, in the others there is not much of difference. Table II shows the sizes and numbers of remaining dummies if we use all options. It also shows the number of places and dummies when we skip the initial preparations; this shows that for a good quality result the early splitting of shared paths is important.

VII. Conclusions

This paper shows how a Breeze netlist can be converted into an equivalent STG. We addressed the issues of converting the initial component specifications in the Balsa Manual by using high- and low-level expressions. We took special care to separate control and data path, inserting communication signals for putting them together again in the end.

In the parallel composition of the components, more than a half of the signals are synchronized; they are lambdarized, and we have shown how to get rid of them completely; this can reduce the over-encoding in Balsa significantly.

We have noticed that using an LP-solver on STG subgraphs can be very helpful for contracting more dummies without sacrificing much time (as it would happen if the full-depth solver were employed). Additionally, the restructuring techniques shared-path and merge-place splitting have proven to be extremely useful. When we added these ideas to some optimizations from the literature, they reduced the number of dummies to 10% (cf. the fifth and the last bar in Figure 12b-f). Hopefully, our findings will be confirmed by further experiments.

This approach allows to build large speed-independent STG-specifications; synthesizing circuits from them is the next step. There are different approaches for this, and our results should be interesting for all of them. In the future, we will look into STG-decomposition for large realistic STGs using our tool DesiJ.

Acknowledgments: We thank Ralf Wollowski and Andrey Mokhov for inspiring discussions about merge-place splitting and redundant places resp., and Will Toms for helpful comments about Balsa as well as the examples provided.

References

[1] K. v. Berkel, M. B. Josephs, and S. M. Nowick, "Scanning the technology: Applications of asynchronous circuits," in *IEEE Proceedings*, vol. 20, pp. 100–109, September 1998.

[2] T. Chu, *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, Massachusetts Institute of Technology, 1987.

[3] "The Balsa Asynchronous Circuit Synthesis System: http://www.cs.manchester.ac.uk/apt/projects/tools/balsa/."

[4] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, "The VLSI-programming language Tangram and its translation into handshake circuits," in *Proc. Eur. design automation*, pp. 384–389, IEEE, 1991.

[5] T. Kolks, S. Vercauteren, and B. Lin, "Control resynthesis for control-dominated asynchronous designs," in *2nd Adv. Research in Asynchronous Circuits and Systems 1996*, pp. 233–243, IEEE, 1996.

[6] W. Vogler and R. Wollowski, "Decomposition in asynchronous circuit design," in *Concurrency and Hardware Design* (J. Cortadella *et al.*, eds.), Lect. Notes Comp. Sci.2549, 152–190. Springer, 2002.

[7] V. Khomenko, M. Schaefer, and W. Vogler, "Output-determinacy and asynchronous circuit synthesis," *Fundamenta Informaticae*, vol. 88, no. 4, pp. 541–579, 2008.

[8] "Petrify: http://www.lsi.upc.es/~jordicf/petrify/petrify.html.."

[9] V. Khomenko, "A usable reachability analyser," tech. rep., Newcastle University, 2009.

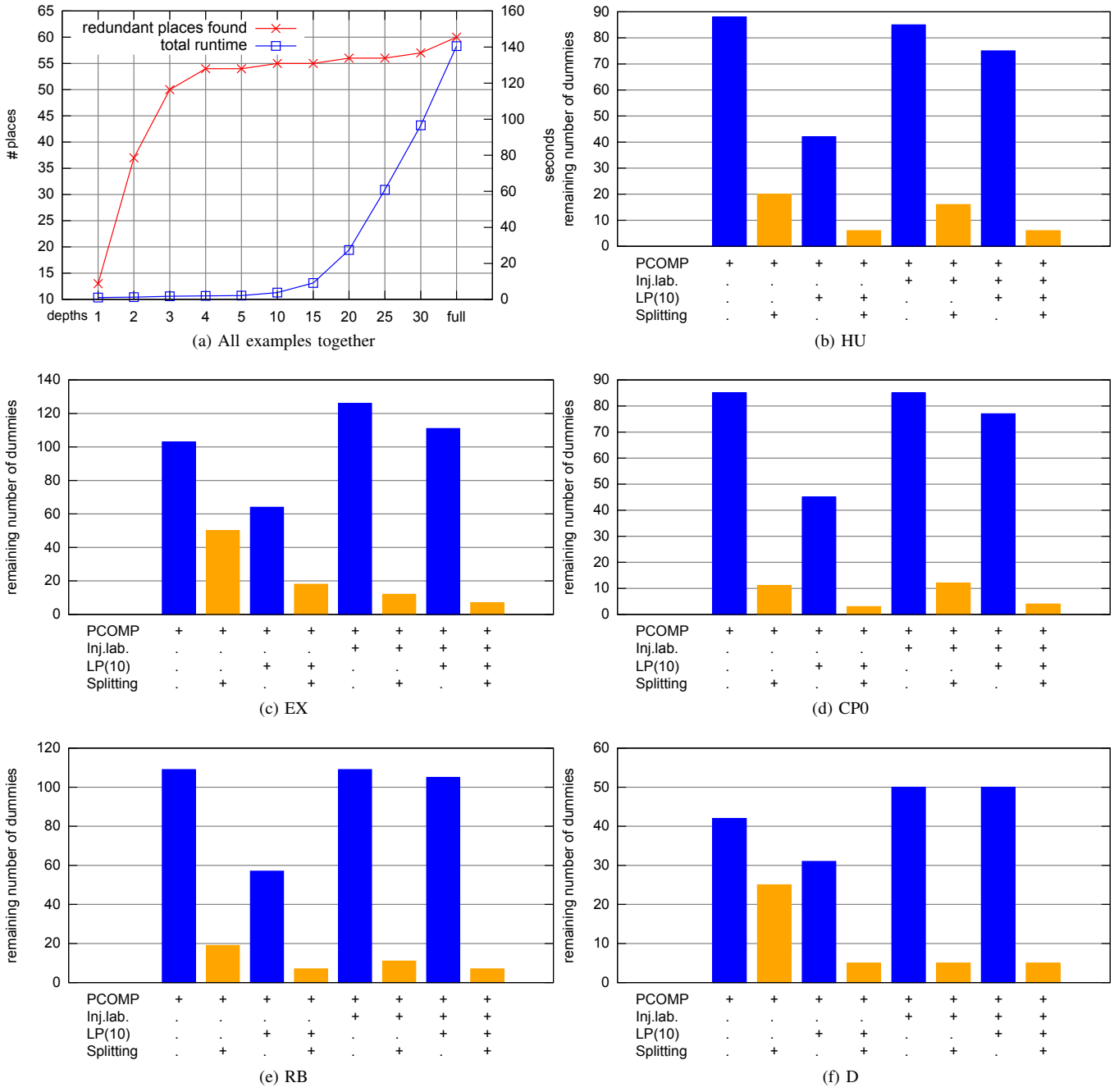
[10] J. Carmona, J.-M. Colom, J. Cortadella, and F. Garcia-Valles, "Synthesis of asynchronous controllers using integer linear programming," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 25, no. 9, pp. 1637–1651, 2006.

[11] V. Khomenko, M. Schäfer, W. Vogler, and R. Wollowski, "STG decomposition strategies in combination with unfolding," *Acta Inf.*, vol. 46, no. 6, pp. 433–474, 2009.

[12] F. de la Cruz Fernandez, "Logic synthesis of handshake components using clustering techniques," Master's thesis, Univ. Politécnica de Catalunya, 2007.

[13] J. C. Ebergen, "Arbiters: an exercise in specifying and decomposing asynchronously communicating components," *Sci. Comput. Program.*, vol. 18, no. 3, pp. 223–245, 1992.

[14] A. Alekseyev, V. Khomenko, A. Mokhov, D. Wist, and A. Yakovlev, "Improved parallel composition of labelled Petri nets," in *ACSD 2011*, pp. 131–140, IEEE, 2011.



(a) All examples together

(b) HU

(c) EX

(d) CP0

(e) RB

(f) D

Figure 12: a) Total time spent on different depths b)-f) Remaining dummies

[15] A. Bardsley, *Implementing Balsa Handshake Circuits*. PhD thesis, Department of Computer Science, University of Manchester, 2000.

[16] *Balsa: A Tutorial Guide*. http://ftp.leg.uct.ac.za/pub/linux/gentoo/distfiles/BalsaManual3.5.pdf, 2006.

[17] E. Best, R. Devillers, and M. Koutny, *Petri Net Algebra*. Springer-Verlag, 2001.

[18] G. Berthelot, "Transformations and decompositions of nets," in *Petri Nets: Central Models and Their Properties* (W. Brauer *et al.*, eds.), Lect. Notes Comp. Sci.254, 359–376. Springer, 1987.