UNIVERSITÄT AUGSBURG

STG-Based Resynthesis for Balsa Circuits

Stanislavs Golubcovs, Walter Vogler,
Norman Kluge

Report 2013-12 November 2013

INSTITUT FÜR INFORMATIK
D-86135 AUGSBURG

Copyright © Stanislavs Golubcovs, Walter Vogler, Norman Kluge
Institut für Informatik
Universität Augsburg
D–86135 Augsburg, Germany
http://www.Informatik.Uni-Augsburg.DE
— all rights reserved —

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 almost *all* components used by the Balsa compiler into STGs; in particular we separated the control path and the data path in the data components. 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.

This research was supported by DFG-project 'Optacon' VO 615/10-1 and WO 814/3-1. This report is the full version of the extended abstract [1].

1 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 [2].

Unfortunately, asynchronous circuits are more difficult to design due to their inherent complexity. They can be specified with *Signal Transition Graphs* [3] (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 language into an asynchronous circuit without analysis of the state space. This transformation is provided by hardware compilers such as BALSA [4] and TAN-

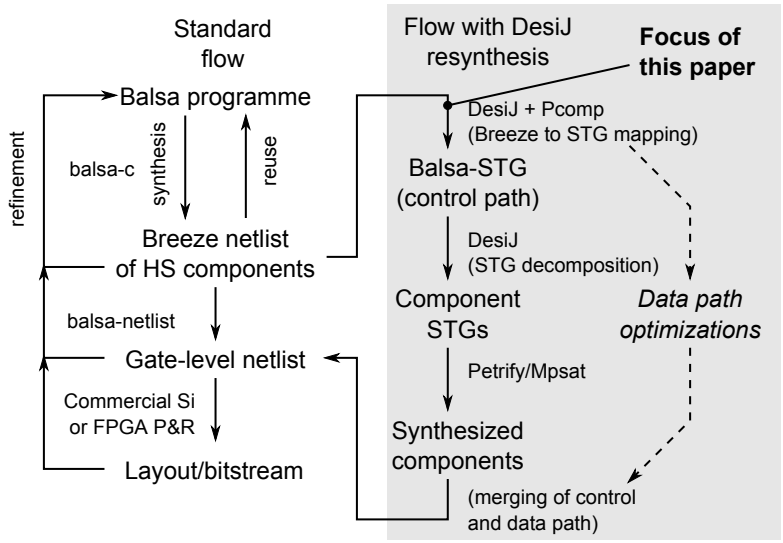


Figure 1.1: Resynthesis with DesiJ decomposition

GRAM [5]. The BALSA compiler converts the high-level *Balsa programme* into a network of *handshake (HS-)components* (Figure 1.1 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 leads to suboptimal performance 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*.

The idea of STG-based *resynthesis* (see e.g. [6]) is to translate each Breeze component into a *Breeze-STG* with the same behaviour. 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 (the empty word) 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 [7, 8]: dummy transitions are contracted, unnecessary places are removed, and also other net transformations may be performed.

We call the final result *Balsa-STG*; its construction is the focus of this paper (cf. Figure 1.1). The Balsa-STG is the input for some STG-specific synthesis methods like PETRIFY [9], PUNF and MPSAT [10], ILP methods [11], or STG-decomposition with DESIJ [12] (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 HS-components [13].

Potentially, 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. [14]. 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. As a consequence, after lambdarization, the STG might violate *output-determinacy* and, thus, lack a clear meaning as explained in [8]. 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 [15] that 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 almost *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 1.1), *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 sources are the high- and low-level behaviour expressions provided by Bardsley [16, 17]. 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. [18] 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 scale set. For instantiation, also a scale factor is provided; replicating the signals in the scale set, the expression is expanded automatically, and then turned into an STG.

- Unfortunately, the behaviour expressions in [16, 17] have their limitations. In some cases, they describe inconsistent (physically impossible) behaviour as already noticed in [13] or behaviour that is generally not expected for some components (e.g. DecisionWait). 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 HS-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 [16, 17].

With our translation from a Breeze netlist to Breeze-STGs, we can apply the ideas of [15]: 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 one can achieve better reduction when one afterwards relaxes injectivity in N with *shared-path splitting*. 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 [8]). An important point is that to make our approach work we have constructed HS-STG that do not have loops (sometimes also called read-arcs).

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 implemented all these ideas in our tool DESIJ and have found that the combination of all our ideas gives the best results, considerably reducing the number of dummy trans-

itions while only using safeness preserving contractions [8]. Finally, in all our realistic Balsa examples, dummies could be removed completely when applying a few more general contractions at the very end.

Basic notions are defined in Section 2; the succeeding section describes how we constructed the HS-STGs, also explaining the behaviour expressions. Section 4 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.

The appendix contains a list of handshake components together with their high-level (if possible) and low-level expressions (except for some simple cases). Often, we also show the HS-STGs to clarify the behaviour of some complex expressions. In many cases, we provide the gate-level models to clarify how the control path and the data paths are separated and how the component scales. In some cases the gate-level implementations were also necessary to figure out the adequate environment assumptions. This report presents improved HS-STGs compared to the conference version [1].

2 Basic definitions

Definition 1. A *Signal Transition Graph* (STG) is a Petri net that models the desired behaviour of an asynchronous circuit [3]. 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. The signals in $Out \cup Int$ are called *local*, since they are under the control of the STG (and the related circuit). 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 ${}^\bullet x \stackrel{\mathtt{df}}{=} \{y \in P \cup T \mid W(y,x) > 0\}$, the postset of x is $x^\bullet \stackrel{\mathtt{df}}{=} \{y \in P \cup T | W(x,y) > 0\}$. 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 $p_2{}^\bullet$.

The graphical representation of STGs is as usual; signal edges are blue for outputs and red and underlined for inputs; internal signals never appear in figures. MG-places are often

replaced by an arc from the transition in the preset to the one in the postset.

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. Another important property is *output determinacy*. As argued in **[8]**, an STG only makes sense if it satisfies this property; here we also consider internal signals, which have to be treated analogously to outputs.

Definition 2. An STG is *output-determinate* if $M_N[v\rangle\rangle M_1$ and $M_N[v\rangle\rangle M_2$ implies that, for every $s \in Out \cup Int$, $M_1[s\pm\rangle\rangle$ iff $M_2[s\pm\rangle\rangle$ **[8]**.

The labelling of an STG is called *injective* if for each pair of non-dummy transitions t and t', $l(t) \neq l(t')$; see e.g. [15]. There, a construction is defined that transforms an STG into an equivalent one with an injective labelling. Here, we just show an example in Figure 2.1.

In the following definition of *parallel composition* $\|$, we will have to consider the distinction between input and output signals. The idea of parallel composition is that the composed systems run in parallel synchronizing on common signals – corresponding to circuits that are connected on signals with the same name. The definition below combines two applications: In the first case, we see the STGs as representing circuits; this is needed when building the initial Balsa-STG. Here, always an input of one component is synchronized with an output of another component, and the merged signal is considered to be internal. If e.g. s is an output of N_1 and an input of N_2, then an occurrence of an edge $s\pm$ in N_1 is 'seen' by N_2, i.e. it must be accompanied by an occurrence of $s\pm$ in N_2. Since we do not know a priori which $s\pm$-labelled transition of N_2 will occur together with some $s\pm$-labelled transition of N_1, we have to allow for each possible pairing.

In the second case, we want to construct an HS-STG in a structural fashion; cf. Section 3.2. In this case, we always merge an input with an input and an output with an output.

Definition 3. The *parallel composition* of STGs N_1 and N_2 is defined if $Int_i \cap (In_j \cup Out_j \cup Int_j) = \emptyset$ for $\{i,j\} = \{1,2\}$. Then, let $A = (In_1 \cup Out_1) \cap (In_2 \cup Out_2)$ be the set of common signals; the *parallel composition* $N = N_1 \parallel N_2$ is obtained from the disjoint union of N_1 and N_2 by

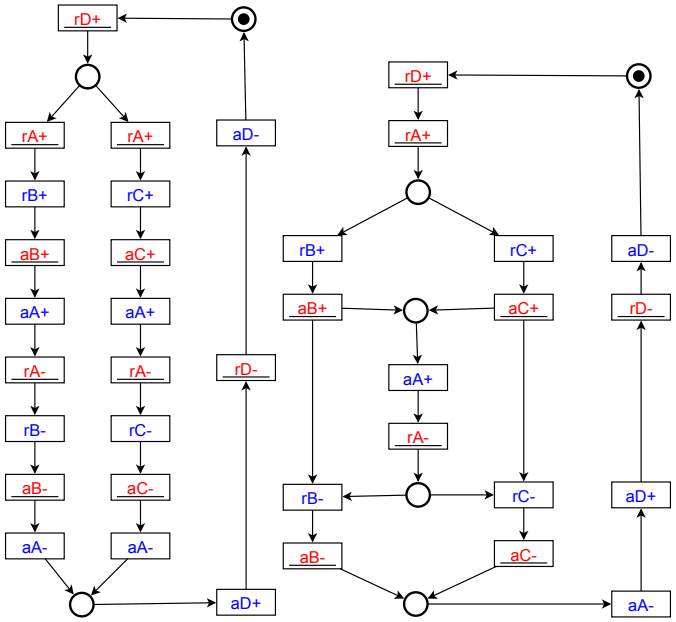


Figure 2.1: Enforcing injectivity (before and after)

combining each $s\pm$-labelled transition t_1 of N_1 with each $s\pm$-labelled transition t_2 from N_2 if $s \in A$. In the formal equations below, $\star$ is used as a dummy element, which is formally combined e.g. with those transitions that do not have their label in the synchronization set A. (We assume that $\star$ is not a transition or a place of any net.) Thus, N is defined by

$$\begin{aligned} P &= P_1 \times \{\star\} \cup \{\star\} \times P_2 \\ T &= \{(t_1, t_2) \mid t_1 \in T_1, t_2 \in T_2, l_1(t_1) = l_2(t_2) \in A\{+,-\}\} \\ &\quad \cup \{(t_1, \star) \mid t_1 \in T_1, l_1(t_1) \notin A\{+,-\}\} \\ &\quad \cup \{(\star, t_2) \mid t_2 \in T_2, l_2(t_2) \notin A\{+,-\}\} \end{aligned}$$

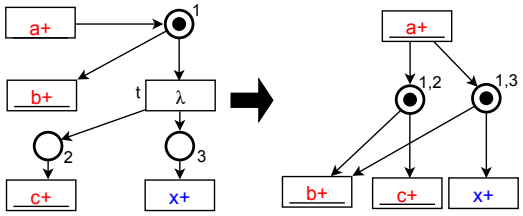


Figure 2.2: t-contraction

$$W((p_1,p_2),(t_1,t_2)) = \begin{cases} W_1(p_1,t_1) \text{ if } p_1 \in P_1,\ t_1 \in T_1 \\ W_2(p_2,t_2) \text{ if } p_2 \in P_2,\ t_2 \in T_2 \\ 0 \text{ otherwise} \end{cases}$$

$$W((t_1,t_2),(p_1,p_2)) = \begin{cases} W_1(t_1,p_1) \text{ if } p_1 \in P_1,\ t_1 \in T_1 \\ W_2(t_2,p_2) \text{ if } p_2 \in P_2,\ t_2 \in T_2 \\ 0 \text{ otherwise} \end{cases}$$

$$l((t_1,t_2)) = \begin{cases} l_1(t_1) & \text{if } t_1 \in T_1 \\ l_2(t_2) & \text{if } t_2 \in T_2 \end{cases}$$

$$M_N = M_{N_1} \dot\cup M_{N_2}, \text{ i.e. } M_N((p_1,p_2)) = \begin{cases} M_{N_1}(p_1) & \text{if } p_1 \in P_1 \\ M_{N_2}(p_2) & \text{if } p_2 \in P_2 \end{cases}$$

$$\begin{aligned} In &= (In_1 \cup In_2) - (Out_1 \cup Out_2) \\ Out &= (Out_1 \cup Out_2) - (In_1 \cup In_2) \\ Int &= Int_1 \cup Int_2 \cup (In_1 \cap Out_2) \cup (Out_1 \cap In_2) \end{aligned}$$

Clearly, one can consider the place set of the composition as the disjoint union of the place sets of the components. Therefore, we can consider markings of the composition (regarded as multisets) as the disjoint union of markings of the components – as exemplified above for M_N.

We call two STGs N_1 and N_2 free from *computation interference* if there is no reachable marking $M_1 \dot\cup M_2$ in their parallel composition, where e.g. N_1 can fire an output transition under M_1, while N_2 does not enable any transition with the same label in A (as above) under M_2, i.e. the second STG is not able to receive the message sent by the first STG [14, 15].

Recall that, when building the initial Balsa-STG, we consider parallel compositions where $Out_1 \cap Out_2 = \emptyset = In_1 \cap In_2$. When constructing HS-STGs as described in Section 3.2, we always have $In_1 \cap Out_2 = \emptyset = Out_1 \cap In_2$. Then, there are no new internal signals, and the inputs and outputs are just the union of the components' inputs, outputs resp.

A most important operation for our approach is *transition contraction*; see Figure 2.2 for a

type-2 secure contraction: t has no merge place in its postset but there is a choice place in its preset. The intuition is that, whenever $M[t\rangle M'$, the transformation identifies M and M'.

Definition 4. 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* [8] if $|{}^\bullet t| = 1$ or $t^\bullet = \{p\}$ with ${}^\bullet p = \{t\}$ and $M_N(p) = 0$.

3 From behaviour expressions to STGs

In this section and in the appendix, there are a few comments (e.g. referring to 4-phase bundled data) where we assume that the reader has some acquaintance with asynchronous circuit design, see e.g. [19]. The section and the appendix can also be read when ignoring these comments.

3.1 Breeze handshake components

Breeze handshake components have ports that are connected in a net through communication channels. These channels provide the medium for exchanging handshakes and transmitting data. Each channel connects exactly two ports: one active (initiating handshakes, filled circle) and one passive (acknowledging handshakes, empty circle). When describing a component, we list the channels that the component is active for; if the component is active for C, then rC is an output and aC an input signal, and vice versa for the other, passive signals. In the scope of this paper we only consider Breeze components with the *4-phase bundled data* protocol [16].

Breeze handshake components are connected to form Breeze netlists; consider the simple example in Figure 3.1. It demonstrates a simple handshake circuit of a single place buffer. The

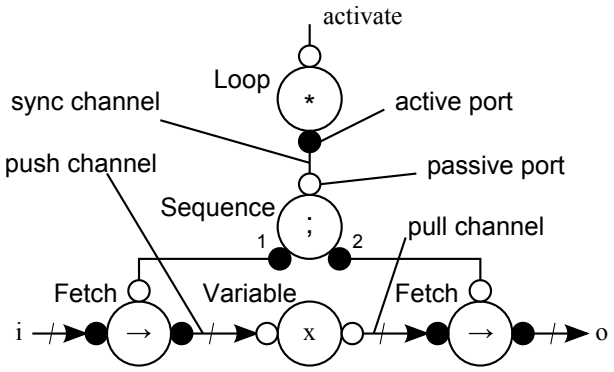


Figure 3.1: Single-place buffer

circuit is initiated by the request on the passive port of the *Loop* component. The initiating request is propagated to the Sequence component, which sequentially requests through its active ports on the left and right sides. The request on the left activates the left Fetch component, which in turn pulls data from the data input port i and pushes it to the variable[1]. Once new data was assigned to the variable x and acknowledged by the component, the data line can be safely released. Eventually, the sequence component receives an acknowledgement from the Fetch component, and after releasing its left request, continues with requesting the second Fetch component.

Note here that the variable is passive on both sides. It receives data from the left and makes sure that the data is stored before it sends and acknowledgement to the writer. However, it always acknowledges the read requests on its output port. Hence, its environment has to guarantee that the variable is never assigned a value and read at the same time. In this example, this is achieved by the sequence component, which does not begin requests on its second port before the communication on the first port has been completely finished. Similarly, it will never allow writing to the variable until the reading phase is complete.

The behaviour of each HS-component can be described using an STG. Building such an HS-STG, however, is not a straightforward task because some components have several variations depending on their parameters. In particular, many components have a variable number of ports – for instance, a sequence component may have any number of outputs. To deal with such components, we have generalized the high-level behaviour expressions of [16, 17], which we describe next.

[1] A channel or port is called *input* or *output* to indicate the flow of data.

3.2 Translation from expressions to STGs

In [16, 17], 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 respectively 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.

It was not always possible to find suitable high-level expressions; in such cases, we only give a low-level one. To increase the expressivity of the former ones, we also allow *signal* names rC or aC, we changed the translation for choice and we added the new operator *follow*.

We describe the syntax of high-level expressions together with their *expansion* into low-level ones, giving in each case the name of the high-level operator.

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

signal: $\triangle rC = rC+ \quad \triangledown rC = rC- \quad$ and
$\triangle aC = aC+ \quad \triangledown aC = aC-$

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: $\triangle(e|f) = (\triangle e; \triangledown e)|(\triangle f; \triangledown f)$
$\triangledown(e|f) = \lambda$

follow: $\triangle(e.f) = \triangle e; \triangle f$
$\triangledown(e.f) = \triangledown e; \triangledown f$

loop: $\triangle(\#e) = \#(\triangle e; \triangledown e)$
$\triangledown(\#e) = \lambda$

Given these translation rules, any high-level expression can be converted into a low-level expression. These low-level expressions are not explained in [16, 17], 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. [18] 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 (for the case that common signals are common inputs or common outputs); 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 [18]): 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 desirable to work with safe STGs

only, see e.g. [12]. 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 [18]). Again, this situation does not turn up here.

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

3.3 Changes to the old expansion rules

Our expansion rules are mostly the same as in the original Balsa manual. Based on the implementations of the HS-component in Verilog and the need to separate the control path from the data path, some new operators were added to the list. The *signal* operator allows adding individual signals. For instance, rA alone defines an isolated request signal; to indicate that rA is an output we list A as active. We may also use aA to express the acknowledgement (which is an input whenever A is listed as active); for treating data, we sometimes also have several acknowledgement signals – e.g. $aB, aB1, \ldots$ – for single request signal rA.

The new *follow* operator was convenient on several occasions to describe the control logic for various data-based HS-components. Originally, the expansion of the *choice* operator was defined by: $\triangle(e|f) = \triangle e|\triangle f$, $\triangledown(e|f) = \triangledown e|\triangledown f$
This definition always leads to inconsistency in the resulting STG – this problem was also mentioned in [13] for the following example: consider the Call component with active channel C. The behaviour is defined with the high-level expression

$$\begin{aligned} active \quad & C \\ Call \quad = \quad & \#(A:C|B:C) \end{aligned}$$

which would expand into low-level expressions as follows:

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

This results in the STG shown in Figure 3.2. It has two choice places where choices between

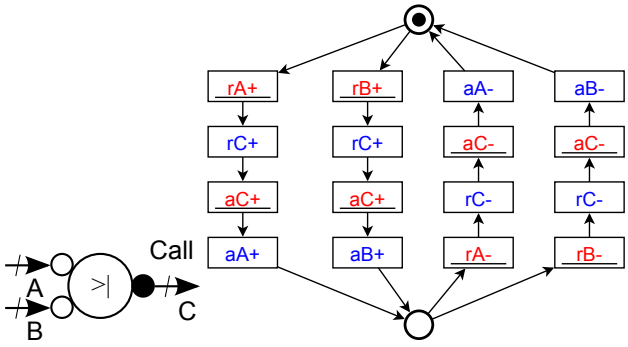


Figure 3.2: Inconsistent Call component

signals rA and rB can be made independently of each other, while in a real circuit the second choice is decided by the first one; e.g. firing $rB-$ after $rA+$ is impossible. Such an inconsistent STG is not accepted by tools like Petrify. With our expansion rule we get:

$$Call = \#((\triangle(A:C); \nabla(A:C)) \,|\, (\triangle(B:C); \nabla(B:C)))$$

and the resulting STG has just one choice place.

Unfortunately, there are cases where our expansion does not give the desired behaviour. The control path of the Case component is defined in [16, 17] by:

$$Case = \#(A : (B|C))$$

The new and the old expansion are:

$$Case_{old} = \#(rA+; \triangle(B|C); aA+; rA-; \nabla(B|C); aA-)$$
$$Case_{new} = \#(rA+; \triangle(B|C); \nabla(B|C); aA+; rA-; aA-)$$

The second expansion avoids inconsistency, but actually $aA+; rA-$ should be in between the set and the reset phase of B (or C resp.). To describe this, we resolved to just considering a low-level expression for the correct behaviour:

$$\#(rA+; ((\triangle B; aA+; rA-; \nabla B) \,|\, (\triangle C; aA+; rA-; \nabla C)); aA-)$$

Possibly, a suitable high-level expression could be found if a different expansion for enclosure were used, maybe just for enclosure in combination with choice. The *Case* component is discussed in more detail in the appendix, where we also present an interface to the data path. This is an example where one request can be answered by one of several acknowledgements.

3.4 Scalable HS-components

As mentioned in the introduction, some HS-components are scalable; the respective high-level expression in [16] would contain an indexed term like $(B_0; C; D_0)||\ldots||(B_{n-1}; C; D_{n-1})$. To have compact (and closed) expressions for such cases, we introduce new operators, e.g. : an expression $\#||e$ for the *repeated parallel* composition $\#||$ is always accompanied by a scale set *scaled*, containing some channel names appearing in e. For the translation into a low-level expression, also a scale factor *scale* is given. Now, as a first step, $\#||e$ with $scale = n$ is expanded to a repeated parallel composition where the first operand is e and the others are replications of e; in the i-th replication ($i = 1, \ldots, n-1$), each channel name appearing in the scale 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 scale set $\{B, D\}$. For scale 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* "#;". The repeated operators are also defined for low-level expressions, which have to be scaled before instantiation in the same way – see the treatment of DecisionWait below.

We can redefine the Call component as scalable (where we also give the scale factor for the expansion presented):

$$
\begin{array}{rcl}
scale & & 2 \\
scaled & & A \\
active & & B \\
Call & = & \#(\#|(A : B)) \\
\triangle Call & = & \#((rA+; rB+; aB+; aA+ \\
 & & ; rA-; rB-; aB-; aA-) \\
 & | & (rA1+; rB+; aB+; aA1+ \\
 & & ; rA1-; rB-; aB-; aA1-))) \\
\triangledown Call & = & \lambda
\end{array}
$$

3.5 Some more HS-components

The full list of components is available in the Appendix. Here we present some examples to demonstrate how the expressions are used.

An example for scaling and for communication with the data path is the *Variable* component, which allows storing information in its local memory elements. For better understanding,

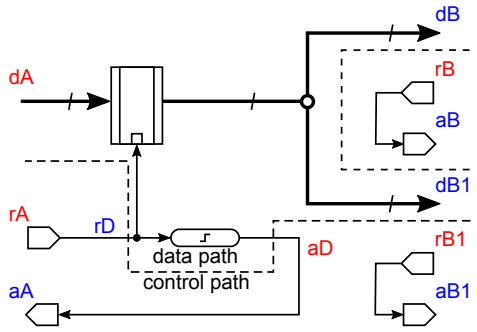


Figure 3.3: Gate-level implementation of Variable with two readers

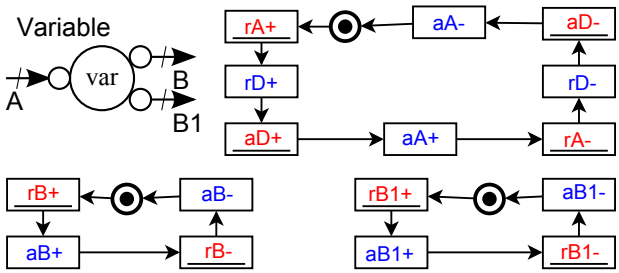


Figure 3.4: Variable with two read channels

its gate-level model with support for two *readers* B and $B1$ is presented in Figure 3.3. The component uses the additional active channel D (the other channels are passive): 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 using D as interface. 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 3.4:

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

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 *DecisionWait* component is another interesting example where a low-level expression is

required. The original specification was (phrased as one of our high-level expressions):

$$DecisionWait = \#(A : \#|(B : C))$$

This requires the behaviour to start with $rA+$, i.e. it does not allow channel B to initiate communication before channel A, which is wrong. Based on the knowledge of the actual implementation, we have determined that the behaviour expression should be as follows:

$$\begin{array}{rcl}
scaled & & B, C \\
active & & C \\
scale & & 2 \\
DecisionWait & = & \triangle\#(\#|(B : C)) \\
 & || & \#(rA+;\#|(\triangle C; aA+; rA-; \triangledown C); aA-) \\
\triangle DecisionWait & = & \#((rB+; rC+; aC+; aB+ \\
 & & ; rB-; rC-; aC-; aB-) \\
 & & |(rB1+; rC1+; aC1+; aB1+ \\
 & & ; rB1-; rC1-; aC1-; aB1-)) \\
 & || & \#(rA+; ((rC+; aC+; aA+; rA-; rC-; aC-) \\
 & & |(rC1+; aC1+; aA+; rA-; rC1-; aC1-)) \\
 & & ; aA-) \\
\triangledown DecisionWait & = & \lambda
\end{array}$$

In this example, the parallel composition has common signal C on both sides, which causes synchronization over transitions labelled $rC+$, $rC-$, $aC+$, and $aC-$, see Figure 3.5.

In this section we have described how we modified and improved the high- and low-level expressions of Bardsley [16]. The translator we have implemented proved to be very useful for studying and discussing different STG models. For example, one can choose different options in the BALSA approach resulting in different behaviours of the HS-components, which obviously would correspond to different behaviour expressions and STG models. More generally, we believe that our languages and the translator can also be used for constructing STGs from sources other than BALSA.

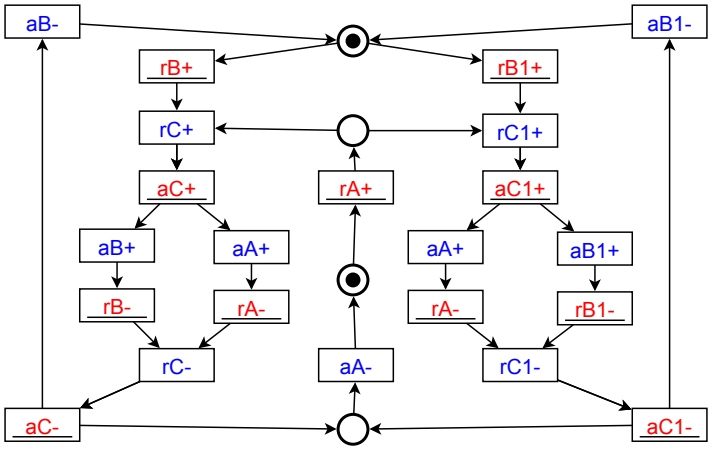


Figure 3.5: DecisionWait STG, scale factor 2

4 Constructing the Balsa-STG

4.1 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 [15]: first, we can *enforce injectivity*; we do not define this here, but it is essentially the inverse of path-splitting defined below. 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 (cf. p. 6), since this guarantees that N' has a clear meaning as explained in [8]; also, we know then a lot about behaviour-preserving reduction operations, see below. For the purpose of our proof, we do not have to distinguish internal and output signals; so all transitions in N' are labelled with (edges of) local or input signals. We know that N' is deterministic since the Breeze-STGs are. Furthermore, no transition in conflict with another one under a reachable marking is labelled with a local signal (*output persistence*) as argued in the introduction. N is derived from N' by lambdarizing all transitions with label in some set of local signals, e.g. the set of internal signals.

Lemma 5. *Whenever $M_N[w_1\rangle M_1$ and $M_N[w_2\rangle M_2$ with $l(w_1) = l(w_2)$, there are v_1, v_2 and M with $M_1[v_1\rangle M \wedge M_2[v_2\rangle M \wedge l(v_1) = \lambda = l(v_2)$.*

Proof. If $|w_1| + |w_2| = 0$, all markings equal M_N and all sequences λ. So 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 a local signal, 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 $M'[t\rangle M$; 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' had local signals, 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 6. *Let N' be an output persistent deterministic STG, and let N be derived from N' by lambdarizing all transitions with label in some set of local signals. Then 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 5. 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 $M_2[v_2\rangle M[t\rangle$. □

4.2 Possible problems with clustering

Let us contrast the observations and the result of Section 4.1 with the problems that may arise in the clustering approach, which considers only pure control components in resynthesis [20]. This approach localizes a number of "islands" of pure control logic, which are converted into a cluster-STG each.

This approach may lead to a problem when such a cluster is to be synthesized. In Figure 4.1 a simple Breeze structure is shown, where the Case component on the left chooses between two of its output ports. Both of these ports are connected to two parallelizers igniting the activity on the DW component that follows.

If in this example one tries to make a cluster out of pure control components, the data-based Case component would be separated from the rest, which forms a cluster. The Case component guarantees the exclusiveness of its outputs; however, this is lost in the cluster. The parallel composition of the three remaining handshake components produces an STG with computation interference, where the DecisionWait component is not ready to accept simultaneously

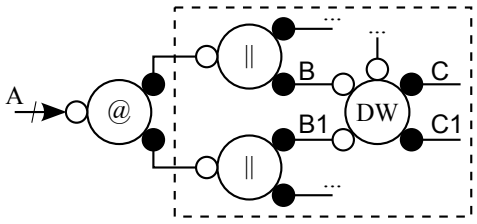


Figure 4.1: Non-SI circuit

both requests from channels B and $B1$ (Figures 4.1 and 3.5). Formally, in this STG, a marking can be reached where both, $rB+$ and $rB1+$, are enabled. After $rB+$ is performed the lower parallelizer has its output $rB1+$ enabled, but it is not enabled in DecisionWait (cf. Figure 3.5) and hence not in the parallel composition. This is computation interference.

Additionally, when the communicating transitions on these channels are lambdarized and contracted, the resulting STG has a free choice between its output transitions $rC+$ and $rC1+$, where one transition disables the other. Hence, this STG is also not output-persistent.

An additional effort is needed to avoid these problems when using clustering. One may enforce cluster borders on the channels that are expected to be mutually exclusive (like B and $B1$); however, this approach may lead to a large number of small clusters. Bearing in mind that Balsa handshake components are quite optimal on their own, this probably destroys all benefits from resynthesis.

4.3 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 [8], 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 too 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.

4.4 Removing redundant places based on a subgraph

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

Definition 7. A place $p \in P$ is *structurally redundant place* [21] 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:

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 8. *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 4.2 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

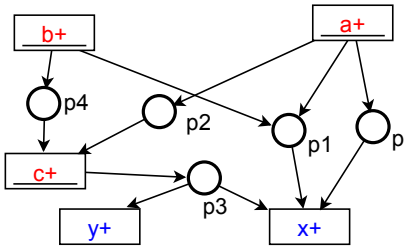


Figure 4.2: Checking place p for redundancy

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: $p^\bullet = \emptyset$ that 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 $p^\bullet$.

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.

4.5 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.

4.5.1 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 4.3). 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

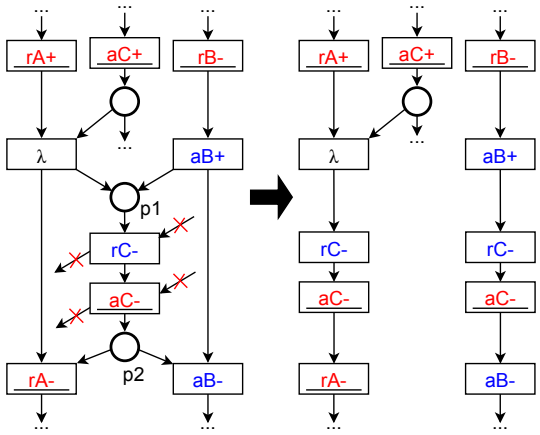


Figure 4.3: Shared-path splitting

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.

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.

4.5.2 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 4.4. 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 4.3, the dummy transition would have several places in both, its preset and postset, and

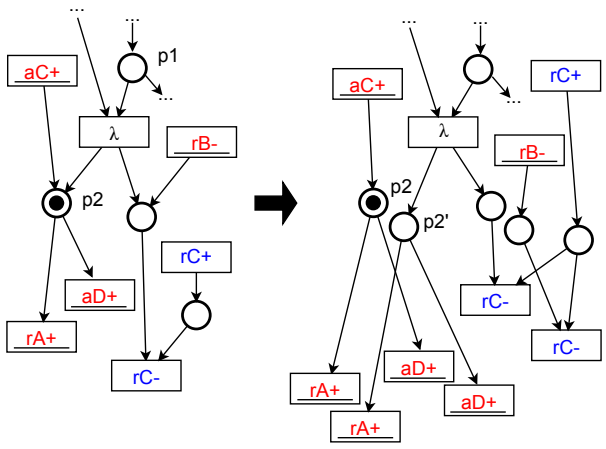


Figure 4.4: Merge-place splitting

contracting it would not be safeness preserving. Hence, merge-place splitting is more suitable as an "emergency plan" when other methods fail.

5 A strategy for STG-reduction with splitting

Based on experiments, we propose the strategy shown in Figure 5.1 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 does not "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

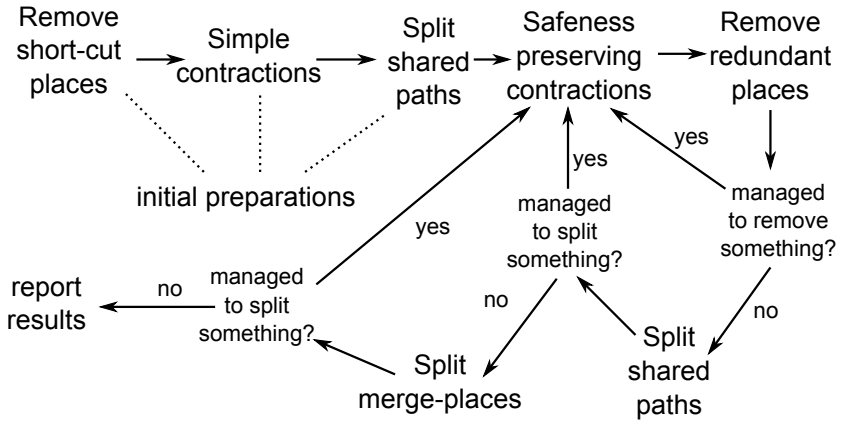


Figure 5.1: Elect reduction strategy

in our examples[2]. 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).

6 Experimental data

For testing our methods, we have chosen 5 reasonably large *realistic* Breeze file samples (Table 1). 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 6.1a). We measured the overall time for the complete 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 once. For depth

[2]In the case of the History Unit example (see below), we also had to use a new structural operation, an admissible operation in the sense of [22].

Table 1: Balsa benchmarks, initial sizes

	|arcs|	$\|T\|$	$\|P\|$	$\|\lambda\|$	|arcs|	$\|T\|$	$\|P\|$	$\|\lambda\|$
	initial				after preparations			
HU	6457	2913	2887	1881	3154	1436	1357	141
EX	9141	4021	4331	2970	4808	2098	2341	508
CP0	5207	2306	2381	1584	2724	1277	1279	160
RB	7335	3237	3286	2101	4312	2038	1986	134
D	4977	2057	2161	1473	3660	1697	1763	270

Table 2: Final results when all optimization options are on

	|arcs|	$\|T\|$	$\|P\|$	$\|\lambda\|$	$\|P\|$	$\|\lambda\|$
					no init	
HU	2875	1315	1212	15	1212	15
EX	3408	1576	1645	7	1581	44
CP0	2370	1119	1086	2	1086	2
RB	4126	2013	1926	7	1926	7
D	2577	1234	1220	5	1164	22

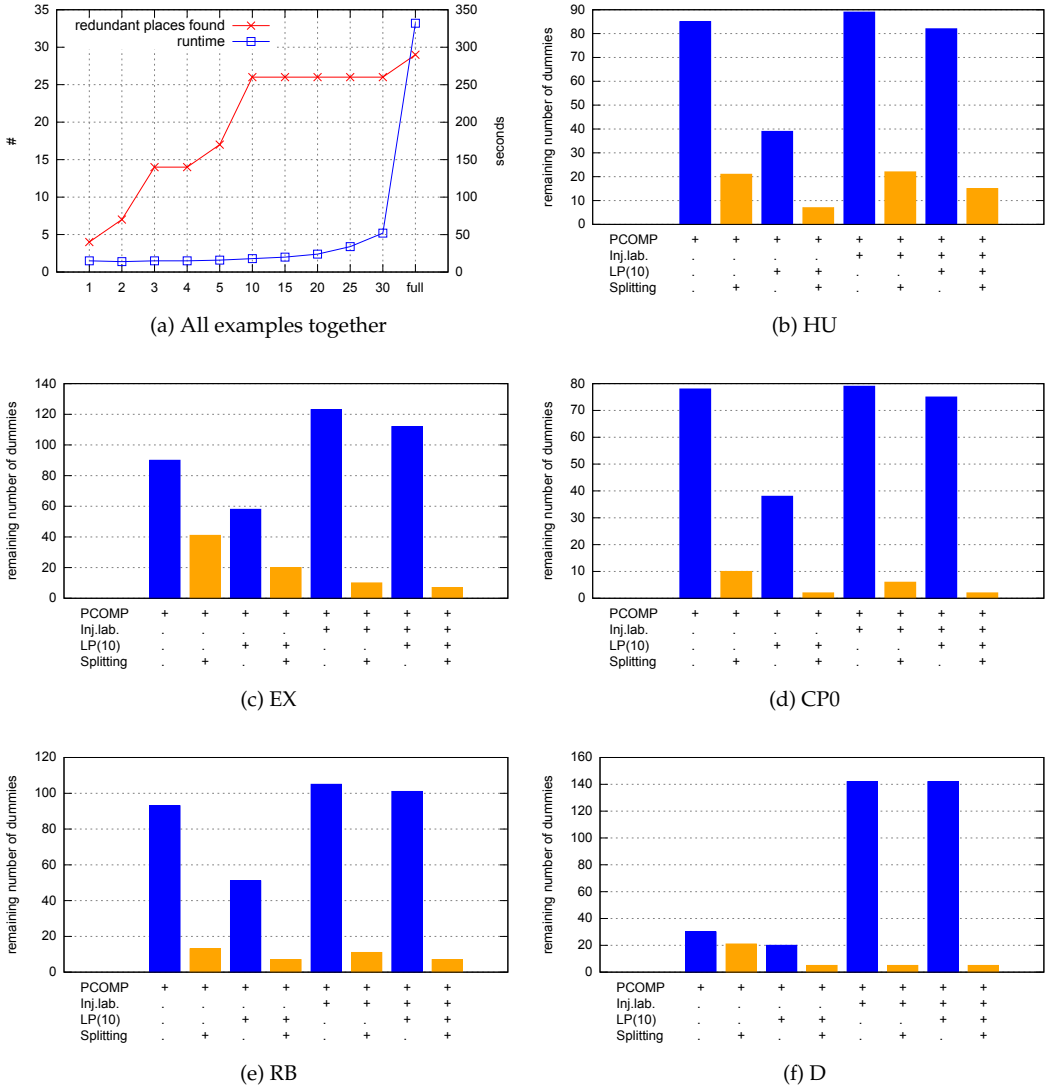
5, this number increases to 172, and to 353 for depth 15. The second factor is the size of the inequations; on larger subgraphs the solver becomes slower.

Figure 6.1a 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 [15]); so we always used this. Figures 6.1b-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: it usually helps. Table 2 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.

6.1 Variations of HS-STGs

Sometimes, it is not completely clear what the HS-STG for an HS-Component of Balsa should look like. As an example we discuss the FalseVariable component, which in the Appendix is defined as follows (cf. Figure 6.2):



(a) All examples together

(b) HU

(c) EX

(d) CP0

(e) RB

(f) D

Figure 6.1: a) Total time spent on different depths b)-f) Remaining dummies

$$
\begin{aligned}
\mathit{scaled} \quad & C \\
\mathit{active} \quad & B \\
f \quad = \quad & \#(A : (rB+; \#||(\#C); aB+; \triangledown B)))
\end{aligned}
$$

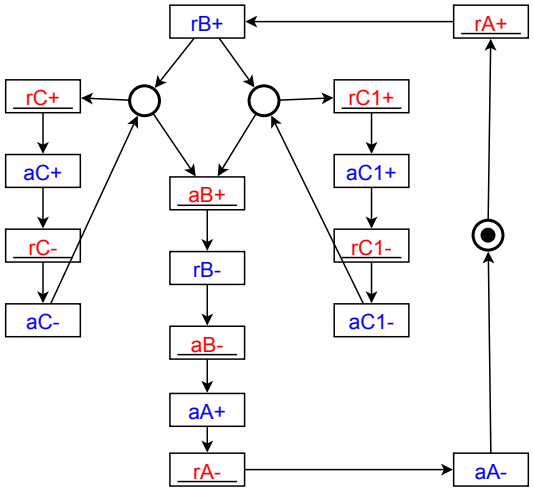


Figure 6.2: Original FalseVariable

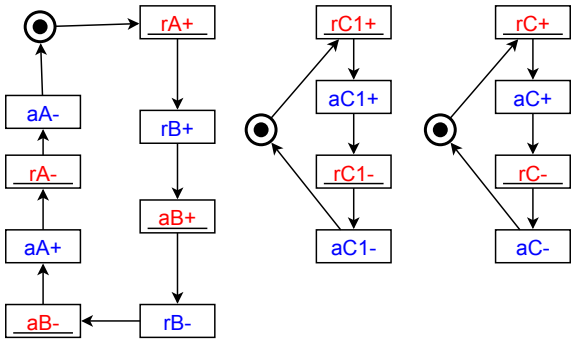


Figure 6.3: FalseVariable with SELEM and separated channels

This STG describes the requirements on the environment of FalseVariable: the readers (the C channels) are only allowed to read the variable after the component signals (via channel B) that the data is available. In the context of Balsa design, this requirement will be satisfied and, as an alternative, one could separate channels C resulting in (cf. Figure 6.3):

$$\begin{aligned} \mathit{scaled} \quad & C \\ \mathit{active} \quad & B \\ f \quad = \quad & \#(A : (\triangle B; \triangledown B))||(\#||(\#(C))) \end{aligned}$$

Since this variation creates fewer arcs, one could expect that it helps to contract more trans-

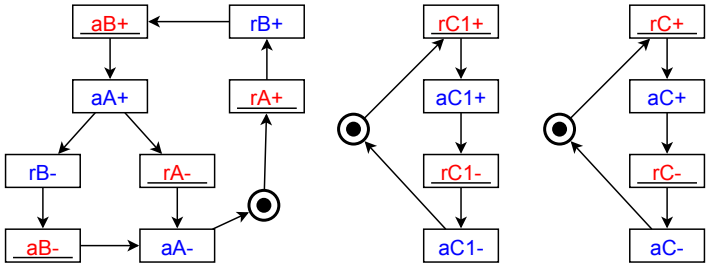


Figure 6.4: FalseVariable with TELEM and separated channels

itions when constructing the Balsa-STG. We have made some experiments showing that separating the channel C from the component has no visible effect (the same amount of arcs and transitions). Presumably the main reason for such a result is that optimized parallel composition and LP-solver based redundant-place removal are efficient enough and can automatically remove these arcs when possible. We also considered a third variation, which exchanges the SELEM (cf. A.1) by a more concurrent so-called TELEM (cf. Figure 6.4):

$$
\begin{aligned}
&\mathit{scaled} && C \\
&\mathit{active} && B \\
&f && = \#(rA+;rB+;aB+;aA+;((rB-;aB-)||rA-);aA-)||(\#||(\#(C)))
\end{aligned}
$$

The use of TELEM sometimes resulted in a few more arcs but no other differences otherwise. Table 3 presents the results for the different variations of the FalseVariable component. Here we have also looked at some additional smaller examples that confirm the general impression.

7 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 behaviour expressions. We took special care to separate control and data path, inserting communication signals for putting them together again in the end.

We modified and improved the languages of these expressions and implemented a translator from high- to low-level expressions and to STGs. This proved to be very useful for studying and discussing different STG models. We believe that our languages and the translator can also be used for constructing STGs from sources other than BALSA.

	old		SELEM		TELEM	
Example	#arcs	#transitions	#arcs	#transitions	#arcs	#transitions
GCD	504	233	504	233	504	233
BMU	1081	463	1081	463	1081	463
GlobalWinner	474	189	474	189	474	189
HistoryUnit	6457	2913	6457	2913	6465	2913
Arb1	318	135	318	135	322	135
Arb2	670	289	670	289	674	289
Shift	1902	810	1902	810	1914	810
AAU	2198	965	2198	965	2204	965
MEM	2423	1083	2423	1083	2427	1083
CP0	5207	2306	5207	2306	5221	2306
DeCode	4977	2057	4977	2057	4981	2057
RegBank	7335	3237	7335	3237	7341	3237
EX	9141	4021	9141	4021	9163	4021

Table 3: Effect of changing the definition of the FalseVariable component

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 6.1b-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.

Acknowledgements. 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] S. Golubcovs, W. Vogler, and N. Kluge, "Stg-based resynthesis for balsa circuits," in *Application of Concurrency to System Design (ACSD), 2013 13th International Conference on*, pp. 140–

149, 2013.

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

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

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

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

[11] 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.

[12] 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.

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

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

[15] 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.

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

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

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

[19] J. Sparsø, "Asynchronous circuit design - a tutorial," in *Chapters 1-8 in "Principles of asynchronous circuit design - A systems Perspective"*, pp. 1–152, Boston / Dordrecht / London: Kluwer Academic Publishers, dec 2001.

[20] F. Fernández-Nogueira and J. Carmona, "Logic synthesis of handshake components using structural clustering techniques," in *PATMOS 2008*, Lect. Notes Comp. Sci.5349, 188–198. Springer, 2009.

[21] 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.

[22] W. Vogler and B. Kangsah, "Improved decomposition of signal transition graphs," *Fundamenta Informaticae*, vol. 78, pp. 161–197, 2007.

A List of HS-components

In this section we present a list of components and explain their functionality. For some components we also show their gate-level implementation and clarify how their control path and data path parts are separated. In these implementations we also use signals like dA, which are only concerned with data path and do not appear in the STGs. Compared to the conference version [1] we have changed some of the HS-STGs. One of the main reasons for this is that we improved the communication between the control and the data path. Further experiments will be needed to validate our implementations.

A.1 SELEM element

In BALSA the SELEM element [16] (the sequence element) is a widely used primitive. It is used to completely enclose a handshake inside another handshake, and its behaviour can be described with the following expression:

$$\begin{array}{rcl} active & & B \\ SELEM & = & \#(rA+; \triangle B; \triangledown B; aA+; rA-; aA-) \end{array}$$

A.1.1 TELEM element

The TELEM encloses handshakes according the to following rules:

$$\begin{array}{rcl} active & & B \\ TELEM & = & \#(rA+; \triangle B; aA+; (rA- || \triangledown B); aA-) \end{array}$$

This component is an alternative to the SELEM element, it has an earlier acknowledgement propagation, which allows more concurrent behaviour and does not require an additional internal signal for complete state coding. However, its use is not allowed when subsequent components prohibit any overlapping of the incoming handshakes (such components, for instance, are Call and DecisionWait).

Figure A.1 demonstrates the way SELEM is implemented in the gate-level model. Here the output of the complex gate CG is an internal signal that resolves a CSC conflict. It is initialized to 1 and behaves as a C-element with its output inverted.

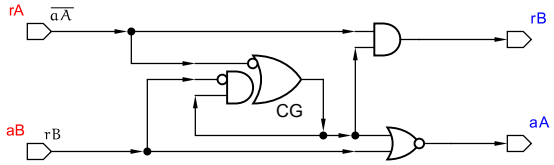


Figure A.1: SELEM element

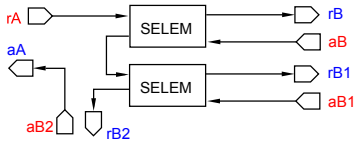


Figure A.2: Sequence for three channels

A.2 Activation driven control components

A.2.1 Sequence

This component provides a sequential handshakes for each of its active ports B; cf. Figure A.2.

$$
\begin{aligned}
\mathit{active} \quad & B \\
\mathit{scaled} \quad & B \\
\mathit{scale} \quad & 2 \\
f \quad &= \quad \#(A : (\#; B)) \\
\triangle f \quad &= \quad \#(rA+; rB+; aB+; rB-; aB-; rB1+; aB1+; aA+; rA-; rB1-; aB1-; aA-) \\
\triangledown f \quad &= \quad \lambda
\end{aligned}
$$

The implementation can be constructed of SELEM and TELEM elements (currently DesiJ supports SELEM components only). The Balsa compiler automatically chooses whether to use SELEM or TELEM, which is shown in the instance parameters. Notably, it uses only $n-1$ SELEM/TELEM components to support n channels, and this is the reason why the last channel is expanded differently in expressions with sequential composition (ie., $\triangle(B; B1; B2) = \triangle B; \triangledown B; \triangle B1; \triangledown B1; \triangle B2$, and $\triangledown(B; B1; B2) = \triangledown B2$).

A.2.2 Concur

The component encloses its concurrent active ports B with its passive port A; cf. Figure A.3.

The basic definition is:

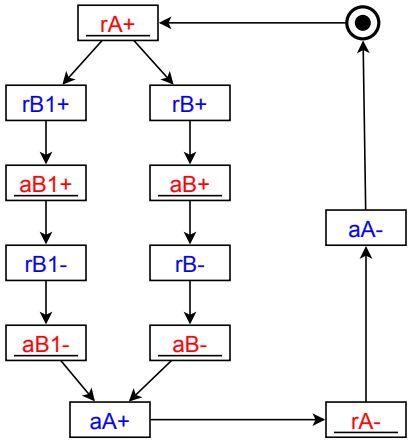


Figure A.3: Concur, scale factor 2

$$\begin{aligned} active &\quad B \\ scaled &\quad B \\ scale &\quad 2 \\ f &= A : (\#||B) \end{aligned}$$

A.2.3 Fork

Fork has one input channel and two or more output channels. It propagates the requests incoming on the input port A to the output ports $B, B1, ...$ Once the acknowledgement is received from each of the output channels, it acknowledges its input channel request. The subsequent reset phase is similar.

$$\begin{aligned} active &\quad B \\ scaled &\quad B \\ scale &\quad 2 \\ f &= \#(A : \#, (B)) \end{aligned}$$

The output for each request rB is implemented by a fork of wires from rA, while the acknowledgement aA is gathered by a tree of C-elements receiving aB.

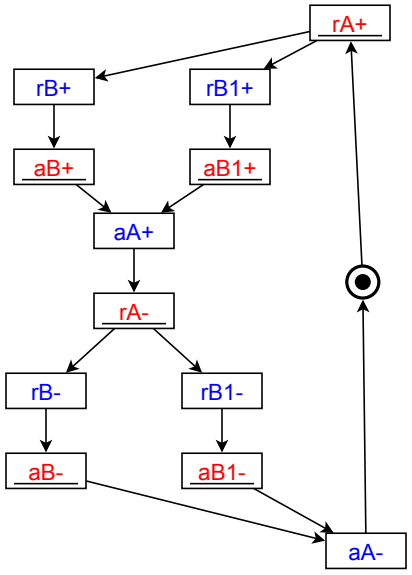


Figure A.4: Fork

A.2.4 WireFork

WireFork receives a request signal and forks it to several outputs (as the name suggests, it is simply a forking wire that broadcasts the incoming activation). This component is different because it does not receive any acknowledgements and does not produce an acknowledgement on its input channel's acknowledgement wire. In circuits it is only used with Loop and other components that do not acknowledge their activator.

$$
\begin{aligned}
\mathit{active} &\quad B \\
\mathit{scaled} &\quad B \\
\mathit{scale} &\quad 3 \\
f &= rA+;\#||(rB+) \\
\mathit{scaled}: &\quad rA+;(rB+;rB1+;rB2+)
\end{aligned}
$$

A.2.5 Loop

Once activated, this component constantly produces handshakes on its output channels (Figure A.5). Since this is an endless loop, the component never acknowledges its activator:

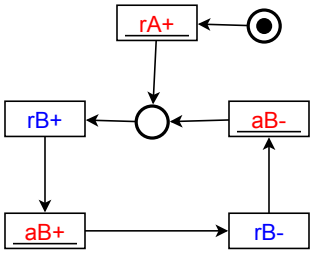


Figure A.5: Loop

$$
\begin{aligned}
active \quad & B \\
f &= A : \#(B) \\
\triangle f &= rA+; \#(rB+; aB+; rB-; aB-) \\
\triangledown f &= \lambda
\end{aligned}
$$

A.3 Channel termination components

A.3.1 Continue and ContinuePush

Continue and ContinuePush have one passive input port A producing acknowledgements. The implementation is a simple wire connection from input signal rA to output signal aA. The component simply acknowledges all of the requests and does nothing with the data received:

$$f = \#(A)$$

A.3.2 Halt and HaltPush

Halt can receive a request, but does not acknowledge it, hence it blocks the component communicating with it:

$$f = rA+$$

The HaltPush does the same, but it also accepts some data bits from the connected channel.

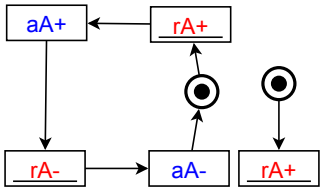


Figure A.6: Continue and Halt

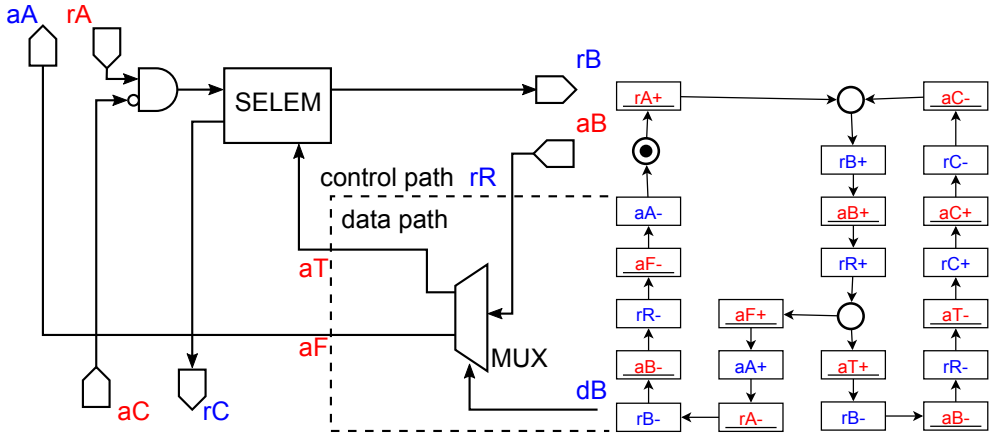


Figure A.7: While

A.4 Control to datapath components

A.4.1 While

The component, once activated on its passive port A, repeatedly starts handshakes on C as long as some Boolean condition evaluates to true on port B (Figure A.7).

When the request on rR fires, the demultiplexer MUX interprets the input data dB and provides a dual-rail input for the control path (aT and aF); cf. Figure A.7 l.h.s. If $aT+$ is fired, the component produces a complete handshake on port C. If $aF+$ fires, the component finishes the handshake on port A and awaits the next request to start a new loop iteration.

The component is described with the low-level expression:

$$\begin{aligned} &\mathit{active} && B, C, R, T, F \\ &f && = \#(rA+; \triangle B; rR+; \#(aT+; \triangledown B; rR-; aT-; C; \triangle B; rR+); aF+; aA+; rA-; \triangledown B; rR-; aF-; aA-) \end{aligned}$$

A.4.2 Fetch

The basic implementation of this component consists of three control wires and some data wires passing from port B to port C. The initial request on the passive port A tells the component to fetch data from the input port B and to propagate it to the output port C. Hence, the behaviour can be described with the following expression:

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

A.4.3 FalseVariable

This component consists of the data writer port A, the signal port B, and one or more reader ports C. Here "False" means there is no implicit memory gate storing the data bits; this is useful to preserve area. Instead, data is directly forked to each of the output ports C. Channel B signals the readers, that the data is available, which allows the readers to use the data line several times completely independently of each other. The port B is acknowledged after the data has been read and is no longer required. The main difference from the Variable is that the writer's channel is not decoupled from the readers because the write port has to keep valid data.

The component receives the "write" request $rA+$ providing the data bits for the readers. Then the component signals the environment on its "signal" port B, stating to the readers that the data is available now. During this time, some number of read requests may arrive on those channels. Once B is acknowledged, no further reading is expected from the environment. The channel B is reset and the acknowledgement $aA+$ follows. Finally, the trace rA- aA- finishes the transaction.

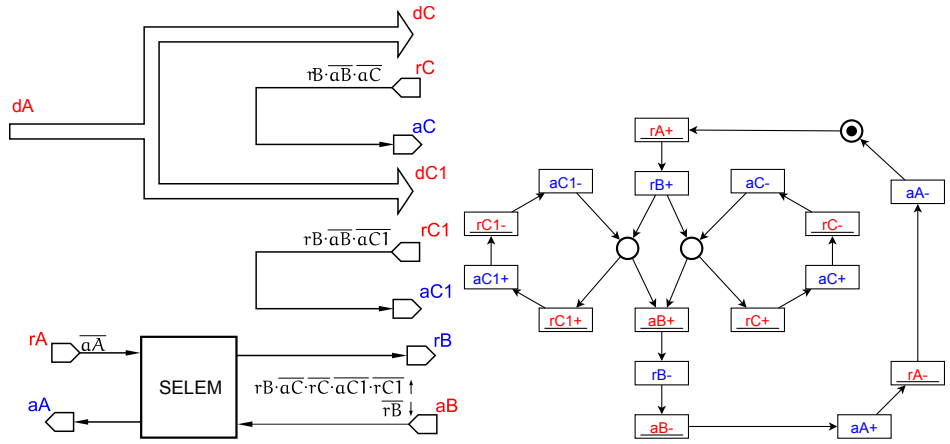


Figure A.8: FalseVariable

$$
\begin{aligned}
\mathit{active} &\quad B \\
\mathit{scaled} &\quad C \\
\mathit{scale} &\quad 2 \\
f &= \#(A : (rB+; \triangle(\#||(\#C)); aB+; \triangledown B)) \\
\triangle f &= \#(rA+; rB+ \\
&\quad ; (\#(rC+; aC+; rC-; aC-)||\#(rC1+; aC1+; rC1-; aC1-)) \\
&\quad ; aB+; rB-; aB-; aA+; rA-; aA-) \\
\triangledown f &= \lambda
\end{aligned}
$$

The implementation and the STG for this component are shown in Figure A.8. Note how the behaviour of the input signals is constrained. The input rC is only allowed to request when rB is high. Also, signal aB is only allowed to acknowledge when none of the readers is requesting data.

It was particularly difficult to find a suitable f for this component. The f given is not really allowed, since it is a high-level loop of a high-level enclosure where the second operand is a low-level expression. We hope it gives a fairly clear compact notation for the expansion also provided.

A.4.4 ActiveEagerFalseVariable

The component expects an initial activation request on channel A. When this arrives, the data is pulled from the writer port B. Afterwards, the port C signals to the readers D that the data is available for reading. There may be several reads from ports D. Once all reading is complete, the acknowledgement on channel C is ignited. Consequently, the reset phase follows: rC- aC- rA- aB- (Figure A.9).

$$
\begin{aligned}
\mathit{active} \quad & B, C \\
\mathit{scaled} \quad & D \\
\mathit{scale} \quad & 2 \\
f \quad = \quad & \triangle\#(A : (B.\#||(\#(aD)))) \\
|| \quad & \#(rA+; rC+; \triangle(\#||(\#D)); aC+; \triangledown C; aA+; \triangledown A)
\end{aligned}
$$

Observe that the first operand of $||$ only considers aD, while the second also considers rD.

Here the term "Active" means that the component has a dedicated port A for igniting its functionality. "Eager" means that, when the activation request arrives, it immediately sends a signal to start reading on port C even before the data has been pulled from channel B. The component itself acknowledges each of the readers D only when the data is ready. This functionality is provided by the asymmetric C-elements AC and AC1 (Figure A.9).

During resynthesis we have noticed that this component may have a slightly simpler implementation (than the one provided in the Balsa library). The asymmetric C-elements AC and AC1 can be safely replaced by simple AND gates.

The STG in Figure A.9 has a rather nasty structure, which does not work well when reducing dummies in the Balsa-STG. On the one hand, it utilizes the concurrent activation of $aB+$ and $aC+$; the usefulness of this optimization depends on the delay from $rC+$ to $rD+$. On the other hand, it adds some performance penalty for each of the read requests $rD+$, as it has to trigger the asymmetric C-element on each of the read transactions.

There is an alternative less concurrent implementation (Figure A.10). Since this gives better results when we construct the Balsa-STG, we have used it in our experiments.

A.4.5 Case

The Case component decodes input data lines to activate one of the output requests, see Figure A.11. To preserve speed independence, the data channels must not change while the rA signal is high, otherwise the data lines are free to change at any moment. In the gate-level

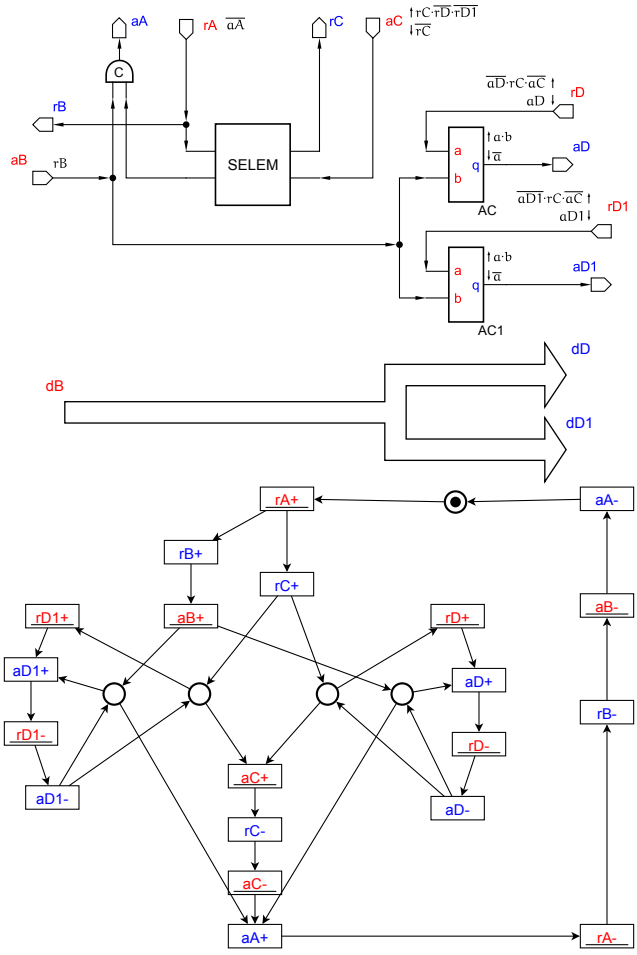


Figure A.9: ActiveEagerFalseVariable

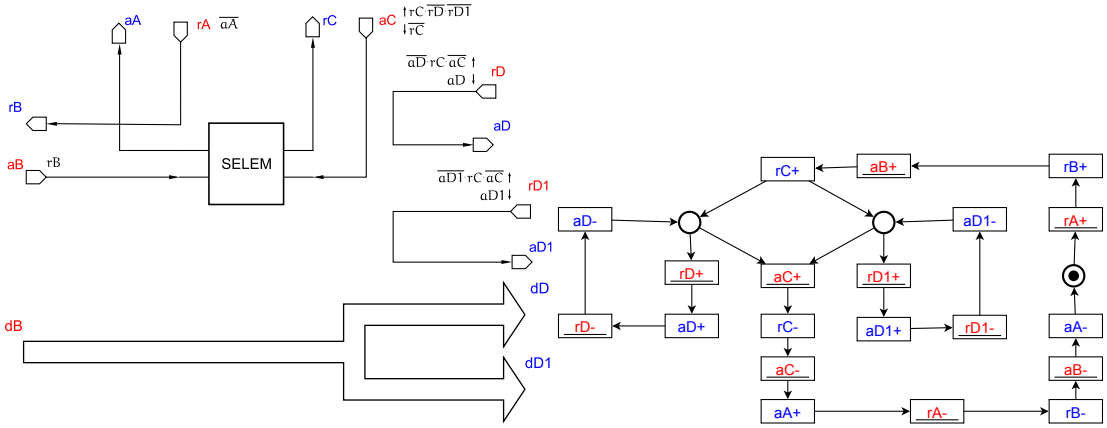


Figure A.10: ActiveEagerFalseVariable (simplified)

implementation, this constraint is shown with the equations like $dA = \begin{cases} \uparrow & \overline{rA} \\ \downarrow & \overline{rA} \end{cases}$. The decoder consists of demultiplexers and C-elements and converts the input data into one of the output requests. The implementation of the decoder may vary for each individual Case component. In the example provided, $dA = 0$ results in a handshake on B, $dA = 1$ results in a handshake on $B1$, etc.

To avoid modelling the decoder in the STG, we consider it being part of the data path. The separation in the STG is implemented with signals rC and aD.

$$
\begin{aligned}
\mathit{active} &\quad B, C, D \\
\mathit{scaled} &\quad B, D \\
\mathit{scale} &\quad 4 \\
f &= \#(rA+; rC+; \#|(aD+; \triangle B; aA+; rA-; rC-; aD-; \triangledown B); aA-)
\end{aligned}
$$

A.4.6 NullAdapt

NullAdapt produces an output request on A for each incoming data push request on B.

The implementation uses a SELEM element connecting input and output ports. This decouples the reset phase of A from the handshakes on B. It is defined as follows:

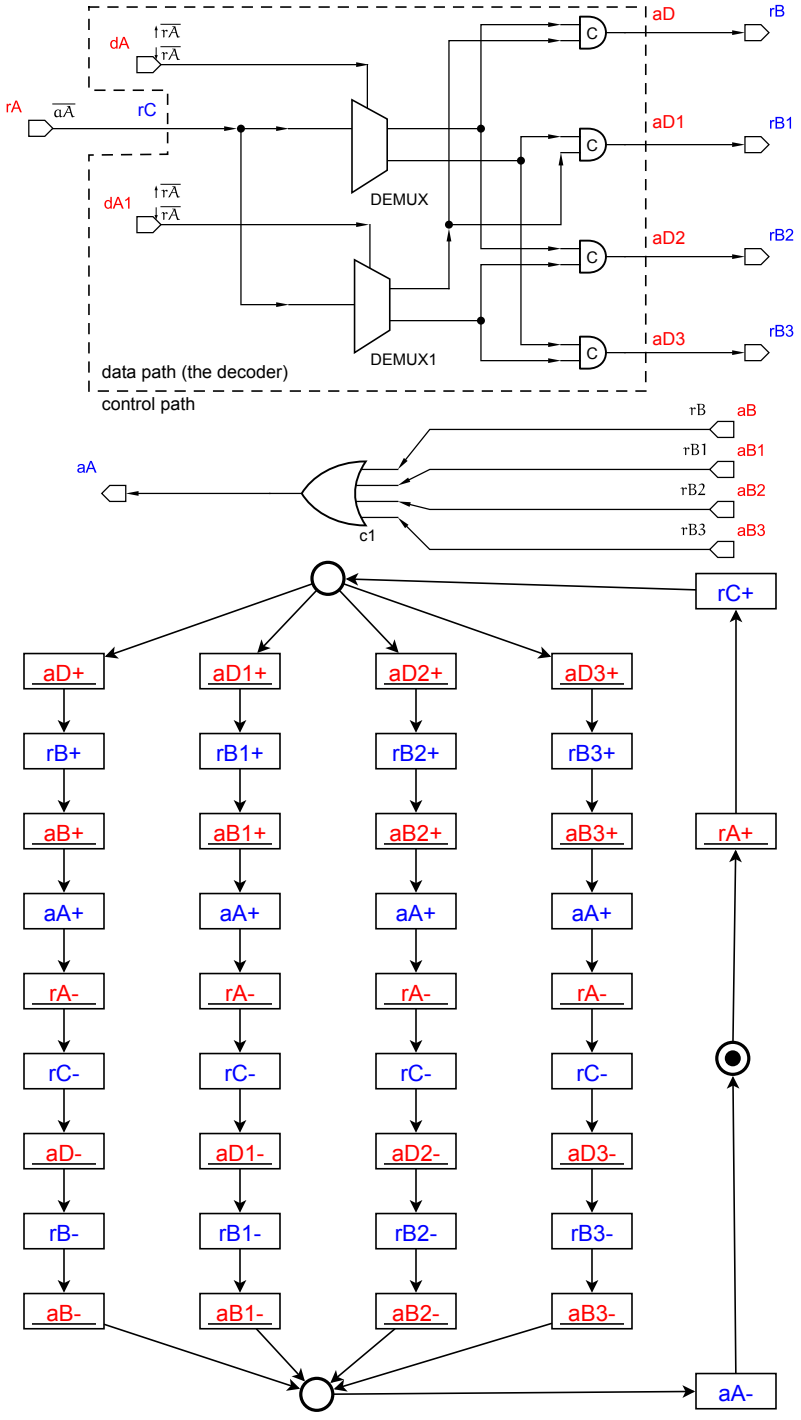


Figure A.11: Case component for 2-bit data converted into 4 output channels

$$
\begin{aligned}
active &\quad A \\
f &= \#(B : (\triangle A; \triangledown A)) \quad \text{(see comment in A.4.3)} \\
\triangle f &= \#(rB+; rA+; aA+; rA-; aA-; aB+; rB-; aB-) \\
\triangledown f &= \lambda
\end{aligned}
$$

A.4.7 Encode

This component converts each input request into the index of the respective A port. A request on port B signals that the index can be found on the data lines. Essentially, this component has the reverse functionality of *Case*. The implementation of the encoder may depend on each particular instance, in the example in Figure A.12 the request $rA+$ is encoded as 0, the request $rA1+$ as 1, etc. For correct operation, the component requires fully exclusive requests among the A ports, meaning that an input channel cannot request while some other input channel is in the middle of a handshake.

All push channels implement the broad bundled data protocol, so *Encode* has to hold valid output data until the moment the receiving side resets with the transition $aB-$. This protocol is enforced with the OR gates receiving inputs rE, ..., $rE3$ as shown in Figure A.12. There is no acknowledgement of transitions rE, ..., $rE3$; however, each change of rE is eventually followed by one of rC, and rC is always acknowledged by aD.

The expression for Encode can be specified quite easily with the "follow" operator:

$$
\begin{aligned}
active &\quad B, C, D, E \\
scaled &\quad A, C \\
scale &\quad 4 \\
f &= \#(\#|(A : (rC.aD.rB.aB.rE)))
\end{aligned}
$$

A.5 Pull datapath components

A.5.1 Adapt and Slice

The Adapt component transfers data from A while adjusting the number of bits in the channel. It has a trivial control part: rB is connected to rA and aA is connected to aB, there is no communication between the data and the control path:

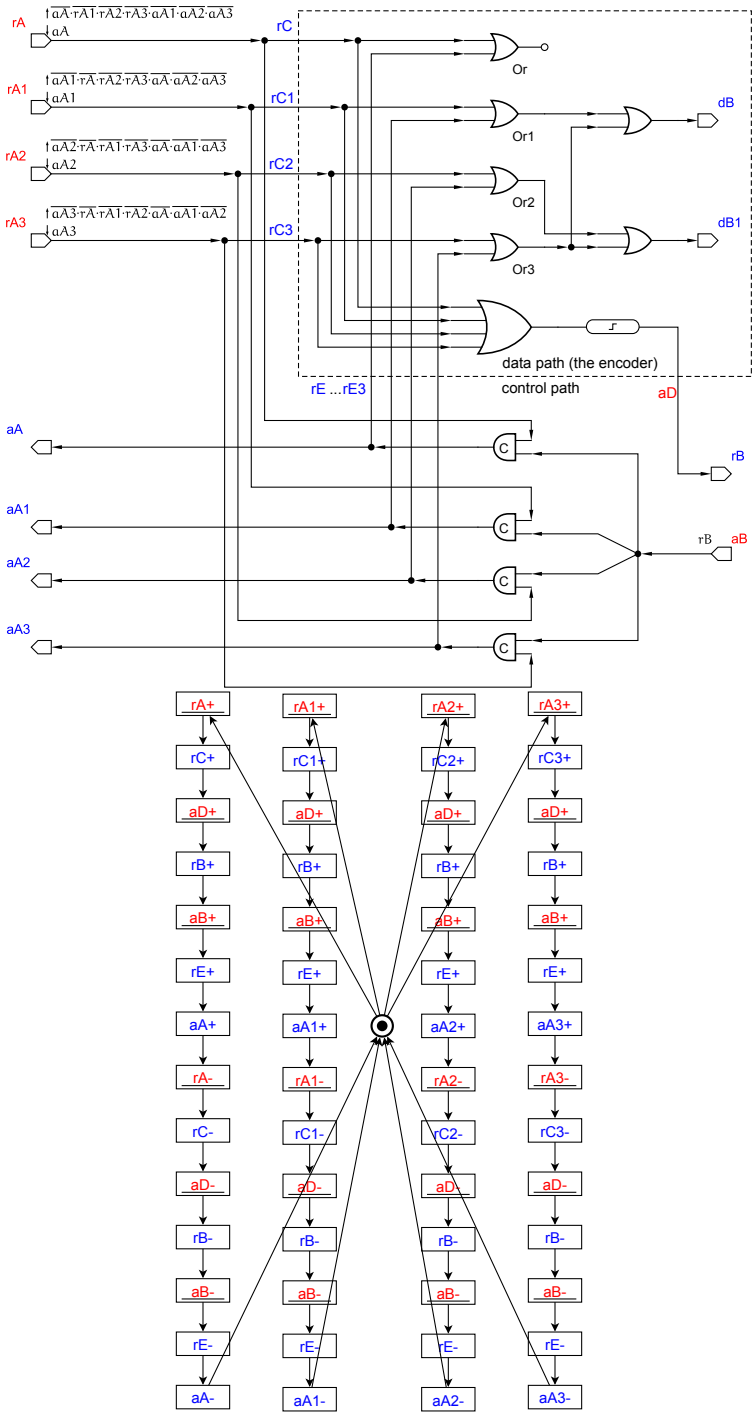


Figure A.12: Encode, the original specification

$$
\begin{aligned}
active \quad & B \\
f &= \#(A : B) \\
\triangle f &= \#(rA+; rB+; aB+; aA+; rA-; rB-; aB-; aA-) \\
\triangledown f &= \lambda
\end{aligned}
$$

Slice is used to extract particular bit wires from the incoming data bits. It has the same control logic implementation as Adapt.

A.5.2 Constant

In this component all data signal values are fixed to either Vdd or Gnd (no transitions on data lines). No computation takes place on the data lines, hence the acknowledgement signal aA is directly connected with the request signal rA as in the component "ContinuePush".

$$
\begin{aligned}
active \quad & A \\
f &= \#(A) \\
\triangle f &= \#(rA+; aA+; rA-; aA-) \\
\triangledown f &= \lambda
\end{aligned}
$$

A.5.3 Combine

This is a simple composition of data signals. Two input ports n and m bits wide form one output channel with width $n + m$. The control part contains a C-element, which is used to propagate the request to the output port. There is no interaction between data lines and the control signals.

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

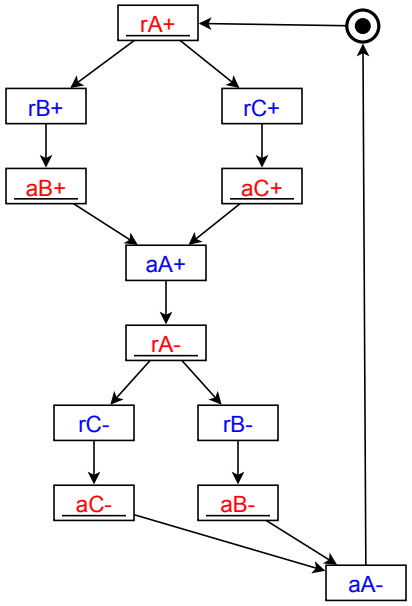


Figure A.13: Combine

A.5.4 CombineEqual

This component composes its output data line from multiple input data lines of equal width. There may be n input ports with m data bits each, the output port is then $n \times m$ bits wide. The expression is:

$$\begin{aligned} \mathit{active} &\quad B \\ \mathit{scaled} &\quad B \\ f &= \#(A : (\#, (B))) \end{aligned}$$

The control part simply broadcasts the initial request from the output port to each of the input ports. Then, the acknowledgements of the input ports are combined using a tree of C-elements. Hence, the output acknowledgement aA rises when each of the input acknowledgements is raised, and similarly the falling edges of all the latter lowers the output acknowledgement.

A.5.5 CaseFetch

This component propagates data on request from one of the input channels C. When requested, it finds pulls index data from port B, decodes the index and then pulls data from C and propagates it to the requester A. D to H form interface to the data path:

$$
\begin{aligned}
\mathit{active} &\quad B, C, D, E, F, G, H \\
\mathit{scaled} &\quad C, F, G \\
\mathit{scale} &\quad 4 \\
f &= \#(rA+; ((D.B); rE+; \#|(aF+; \triangle(C.rG.aH); aA+; rA-; rE-; aF-; \triangledown(C.rG.aH)); aA-))
\end{aligned}
$$

Once the component receives the initial request from the output port, it requests its index channel B to figure out, which of the C channels should request the data (Figure A.14). The SELEM component ensures that the communication with the index channel finishes completely before propagating the request to rC. This decouples the data provider on the index channel B and the data providers on the C channels.

Notice that this component has to have signal aD explicitly acknowledging work of the data latches. This is required because the channel B might get completely lambdarized (for instance, when it is connected to a read port of some Variable), resulting in a trace: $rA+; rD+; rD-; rE+;$ where rD resets immediately after if was set. Such a behaviour can be regarded as a glitch, which may be completely ignored by the memory latches, therefore the explicit acknowledgement wire aD must be present.

A.5.6 BinaryFunc, BinaryFuncConstR, and UnaryFunc

BinaryFunc receives a pull request from the output port A. It propagates the request to its input ports B and C, pulls data from them and outputs the result of its combinational logic function out to port A:

$$
\begin{aligned}
\mathit{active} &\quad B, C, D \\
f &= \#(A : ((B, C).D))
\end{aligned}
$$

The BinaryFuncConstR does the same, however, one of its inputs is a constant, which needs no dedicated handshake. Therefore, the control part is a simple enclosure. We still add a delay element here:

$$
\begin{aligned}
\mathit{active} &\quad B, D \\
f &= \#(A : (B.D))
\end{aligned}
$$

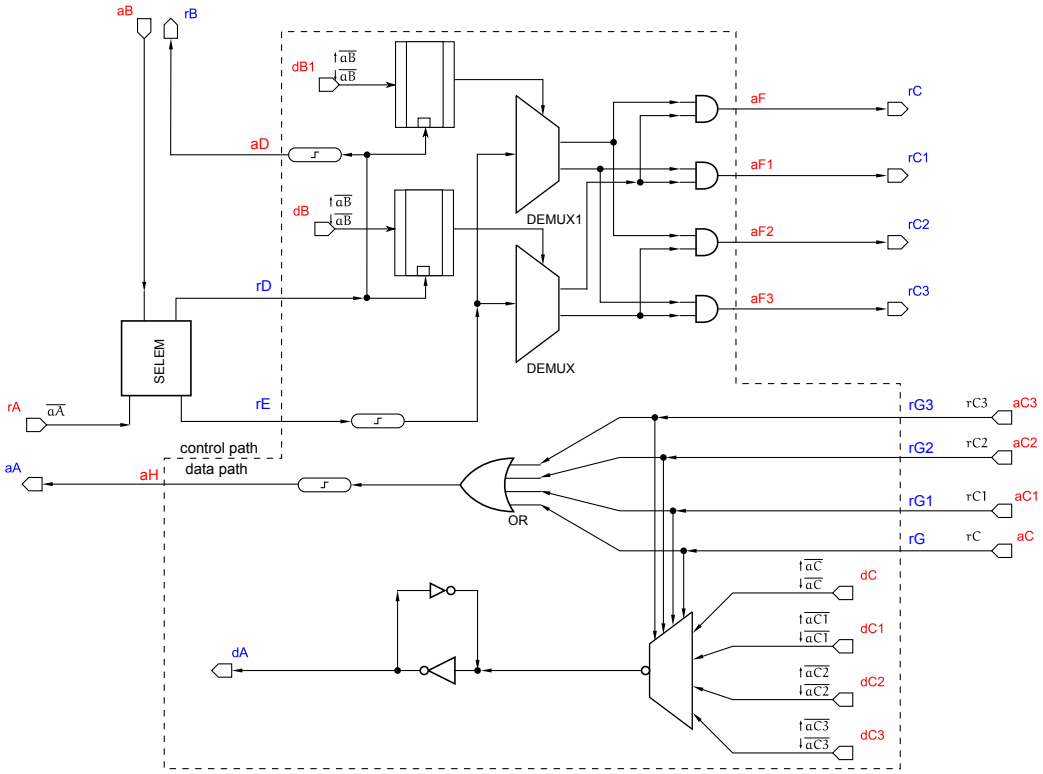


Figure A.14: CaseFetch

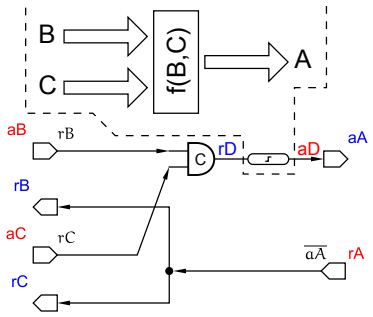


Figure A.15: BinaryFunc

The UnaryFunc is similar to BinaryFuncConstR, however, it does not need any extra information about the constant (such as the number of bits or the sign bit presence). It has the same control logic as BinaryFuncConstR.

A.6 Connection components

A.6.1 ForkPush

ForkPush is Fork with data wires attached. It broadcasts its input data on port A to each of the output ports B. When each of the output ports is acknowledged, the acknowledgement propagates to the input port:

$$
\begin{aligned}
\mathit{active} \quad & B \\
\mathit{scaled} \quad & B \\
\mathit{scale} \quad & 2 \\
f \quad = \quad & \#(A:\#,(B)) \\
\triangle f \quad = \quad & \#(rA+;((rB+;aB+)||(rB1+;aB1+));aA+;rA-;((rB-;aB-)||(rB1-;aB1-));aA-) \\
\triangledown f \quad = \quad & \lambda
\end{aligned}
$$

A.6.2 Call

This component has a number of passive input channels. As soon as one of these channels is activated, the component propagates its request to the only input channel. The environment guarantees that no more than one channel is active at a time. It has a straightforward specification:

$$
\begin{aligned}
\mathit{active} &\quad B \\
\mathit{scaled} &\quad A \\
\mathit{scale} &\quad 2 \\
f &= \#(\#|(A:B)) \\
\triangle f &= \#((rA+;rB+;aB+;aA+;rA-;rB-;aB-;aA-)\,|\,(rA1+;rB+;aB+;aA1+;rA1-;rB-;aB-;aA1-)) \\
\triangledown f &= \lambda
\end{aligned}
$$

[3]

A.6.3 CallMux and CallDemux

The CallMux has the functionality of Call, however, it also pushes the data incoming from its input ports. The request signals $rA,\ldots,rA3$ are propagated to the data channel through $rS,\ldots,rS3$ (Figure A.17) to configure the associated multiplexer. The responsibility of the data path here is to acknowledge rS signal with aD when the output data has been settled to a stable value.

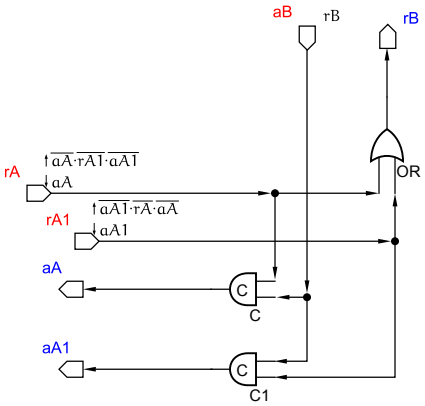
$$
\begin{aligned}
\mathit{active} &\quad B,S,D \\
\mathit{scaled} &\quad A,S \\
\mathit{scale} &\quad 2 \\
f &= \#(\#|(A:(rS.aD.B)))
\end{aligned}
$$

The CallDemux is the same as CallMux, with the exception that the data is propagated in the opposite direction: from the input port B to one of the output ports A, its data broadcasting is implemented as a simple wire fork, without interaction with the control path.

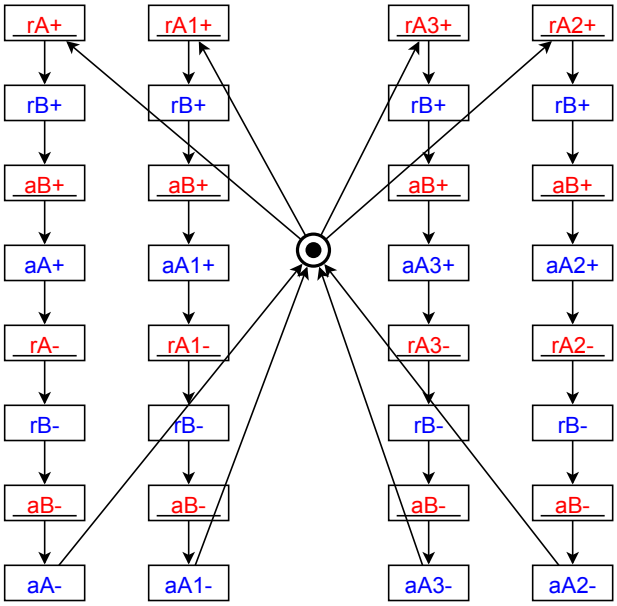
A.6.4 Passivator and PassivatorPush

The Passivator gathers requests from all passive ports and then acknowledges all the requesters. Its implementation is a simple C-element tree, which acknowledges each input port A via wire fork. To express this component, we use an custom output signal aB:

[3] The resynthesis of the circuit implementation has shown that the C-elements can be replaced with simpler latches or the asymmetric two-input C-elements.



(a) Call component circuit



(b) Call component STG

Figure A.16: Call

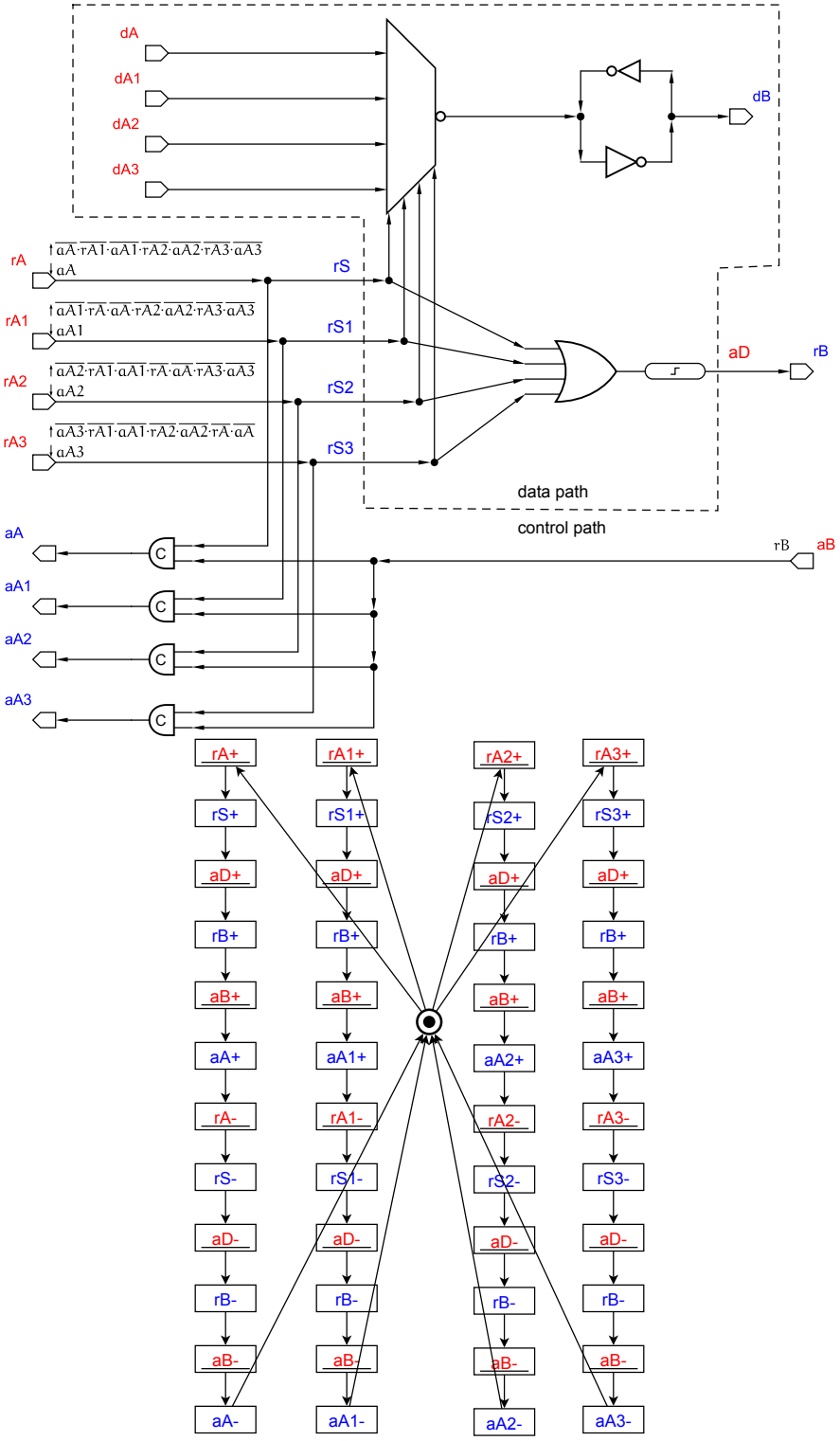


Figure A.14: CallMux

$$
\begin{array}{rcl}
\mathit{scaled} & & A \\
\mathit{scale} & & 4 \\
f & = & \#||(\#(A:(aB)))
\end{array}
$$

Theoretically, this component can be created without the additional signal aB; however, this produces a large number of places. An experiment has shown that with scale factor 9 this breaks PETRIFY.

The *PassivatorPush* accepts requests from all channels and acknowledges every one, while the data is propagated from port B to the A-ports:

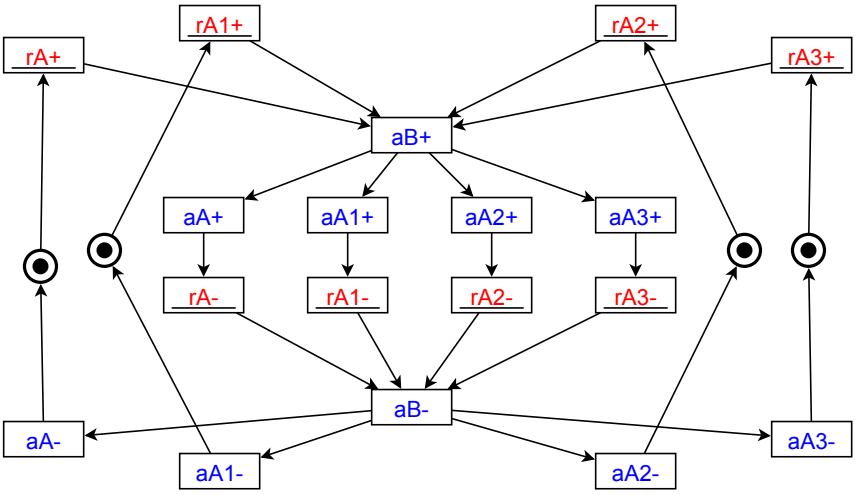
$$
\begin{array}{rcl}
\mathit{scaled} & & A \\
\mathit{scale} & & 4 \\
f & = & \triangle(\#(B))||(\#||(\#(rA+;aB+;aA+;rA-;aB-;aA-)))
\end{array}
$$

A.6.5 Synch and SynchPull

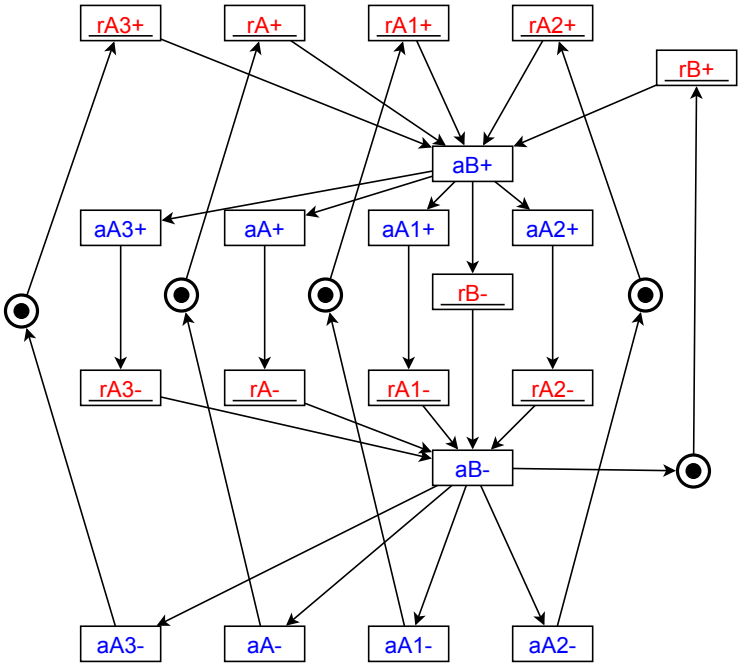
The synchronization component waits until each of the passive ports receives a request, then it produces a request on its only output port. After the output port is acknowledged, it acknowledges each of the requesters. The reset phase follows in a similar fashion. Its implementation is trivial, using a tree of C-elements.

$$
\begin{array}{rcl}
\mathit{active} & & B \\
\mathit{scaled} & & A \\
\mathit{scale} & & 2 \\
f & = & \#||(\#(A:B)) \\
\triangle f & = & \#(rA+;rB+;aB+;aA+;rA-;rB-;aB-;aA-) \\
 & || & \#(rA1+;rB+;aB+;aA1+;rA1-;rB-;aB-;aA1-) \\
\triangledown f & = & \lambda
\end{array}
$$

The *SynchPull* component has the same implementation; however, it additionally pulls data along the channels. There is no communication between the data and the control paths.



(a) Passivator



(b) PassivatorPush

Figure A.18: Passivator components

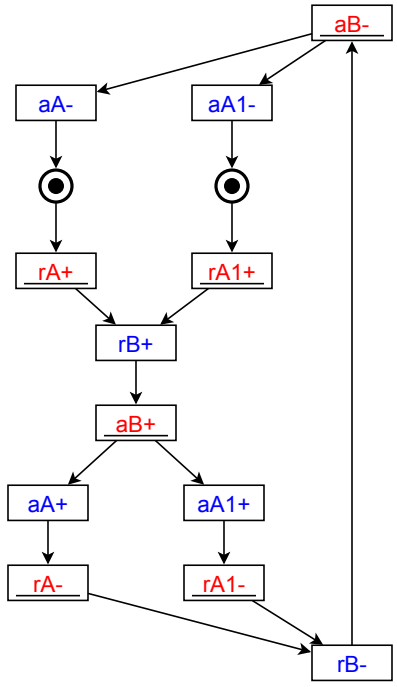


Figure A.19: Synch

A.6.6 SynchPush

This component has an additional active output port C. The data is actively provided on passive port A .

$$
\begin{aligned}
\textit{active} \quad & C \\
\textit{scaled} \quad & B \\
\textit{scale} \quad & 2 \\
f \quad = \quad & \#(A:C)||(\#||(\#(B:C))) \\
\triangle f \quad = \quad & \#(rA+;rC+;aC+;aA+;rA-;rC-;aC-;aA-) \\
\quad || \quad & \#(rB+;rC+;aC+;aB+;rB-;rC-;aC-;aB-) \\
\quad || \quad & \#(rB1+;rC+;aC+;aB1+;rB1-;rC-;aC-;aB1-) \\
\nabla f \quad = \quad & \lambda
\end{aligned}
$$

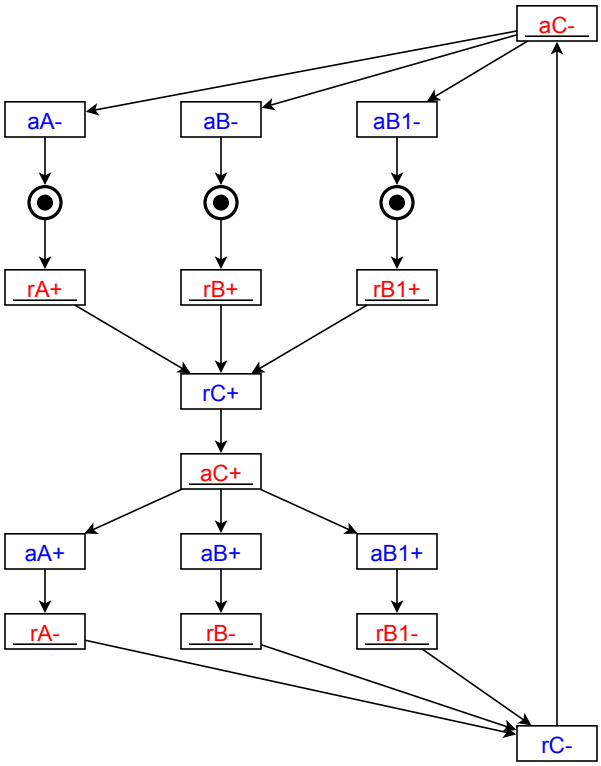


Figure A.20: SynchPush

A.6.7 DecisionWait

The DecisionWait component listens to its passive input ports B and the activator port A. Once A and one of the B ports produce a request, the component activates the corresponding C port (depending on which B the request arrived on). We can define the behaviour using parallel composition:

$$\begin{aligned} &\mathit{active} && C \\ &\mathit{scaled} && B,\, C \\ &f &=\ & \#(\#|(B:C))||\,\#(\#|(A:C)) \end{aligned}$$

As it can be seen from the Verilog implementation of the component (Figure A.21a), the arbitrary request order of channels A and B is supported. This circuit is speed-independent, provided channels B and $B1$ are mutually exclusive. The corresponding STG is shown in Figure A.21.

A.6.8 Split

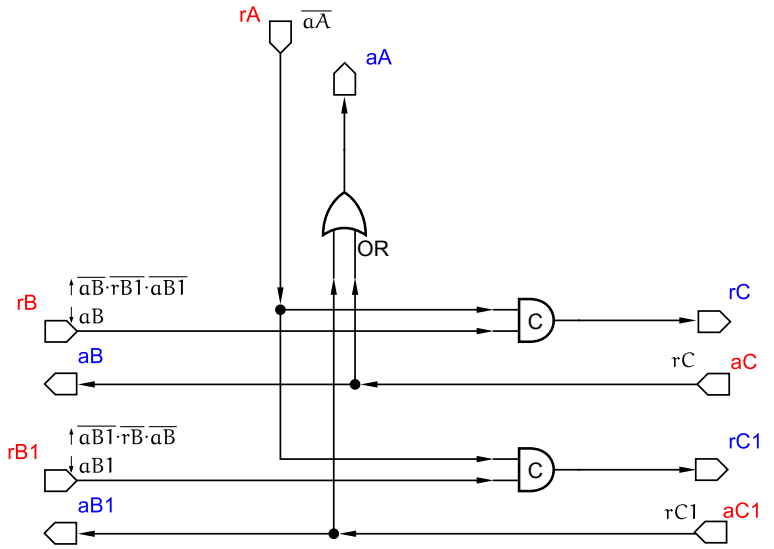
This component, according to given parameters, splits its input channel A into two output channels B and C. The control logic is the same as in two output Fork component.

$$\begin{aligned} &\mathit{active} && B, C \\ &f &=\ & \#(A:(B,C)) \end{aligned}$$

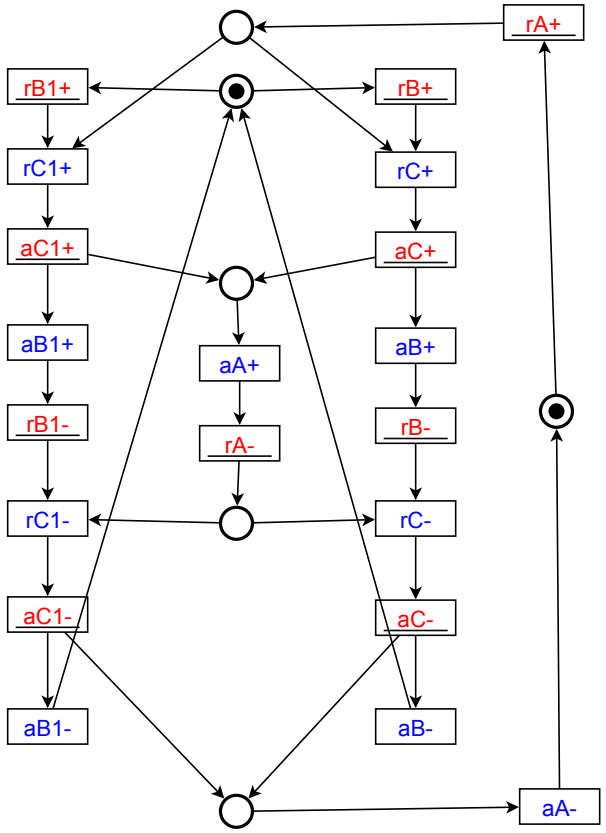
A.6.9 Arbiter

Arbiter has two passive ports A, B and two active ports C, D. The port A encloses C and the port B encloses D (Figure A.23). Because of the MUTEX element in the circuit, which cannot be synthesized, two additional channels E and F were added to place MUTEX outside the synthesized part of the STG:

$$\begin{aligned} &\mathit{active} && C, D, E, F \\ &f &=\ & \#(A:(E.C))||\#(B:(F.D))||\#(aE\,|\,aF)||\#(C\,|\,D) \end{aligned}$$



(a) Verilog implementation (environment ensures B and $B1$ are completely exclusive)



(b) STG

Figure A.21: DecisionWait component, scale factor 2

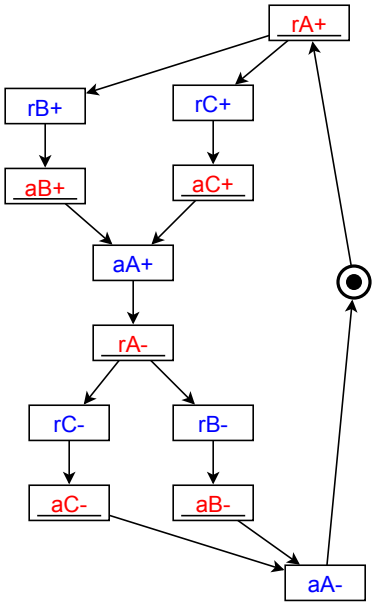


Figure A.22: Split

A.6.10 Variable, InitVariable

Variable supports a single write channel and several read channels. It uses transparent data latches to store the data received from the write channel A. It has a delay line to make sure the data is written to the latches before it acknowledges the write request on channel D. Independently of each other, the readers can request and read the data stored. It is assumed that the environment of the component ensures all read requests on port B are exclusive with all write requests on the reader ports A:

$$\begin{aligned} \mathit{scale} &\quad 2 \\ \mathit{scaled} &\quad B \\ f &= \#(A : D)||(\#||(\#B)) \end{aligned}$$

The *InitVariable* is only different because it has some initial value stored before the first write request arrives on A.

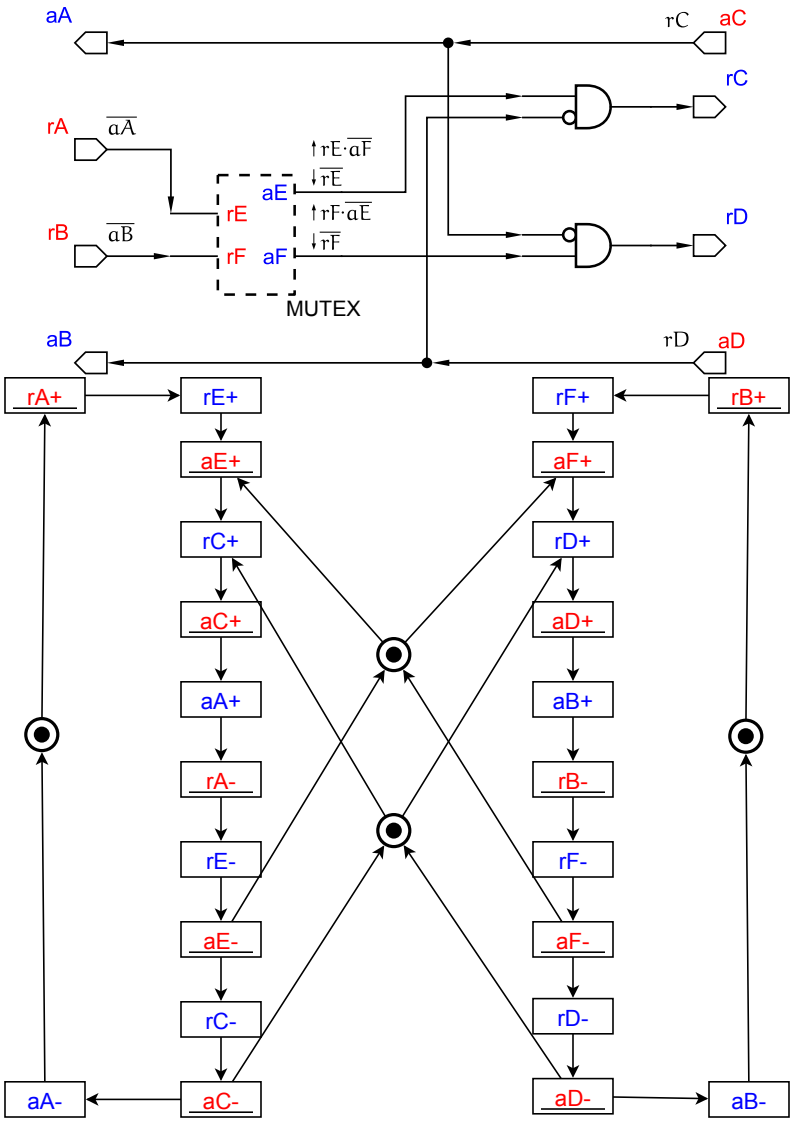


Figure A.23: Arbiter

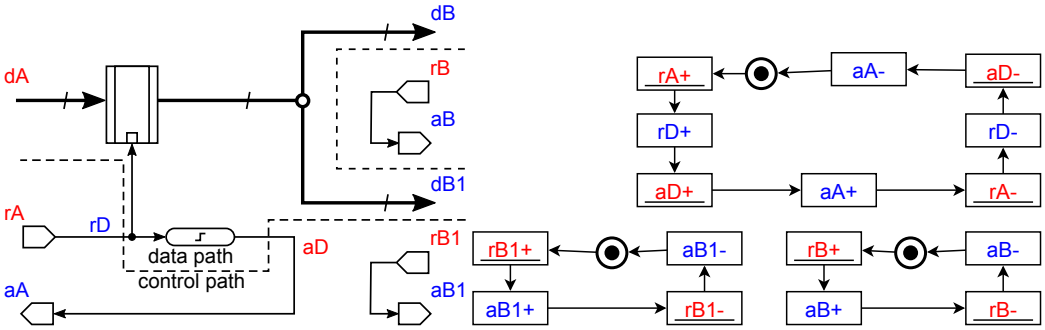


Figure A.24: Variable with two read channels