

Optimisation of handshake circuits control path using STG resynthesis

{author information removed for blind review}

Abstract—The paper proposes a modification of the standard design workflow that is used in Balsa design automation system. The controllers obtained by syntax-directed mapping used in Balsa usually suffer from performance, area and power overheads because the predefined set of components is required to implement the declared protocols fully and correctly in order to be reusable in all possible circuit configurations, which results in redundancy. This redundancy can be eliminated by replacing the manually designed gate-level implementations of the high level components with the corresponding STG specifications. The STGs of individual components that form the system are then composed together to produce the final system STG that is used to synthesise an optimal implementation of the control circuit. The process is automated as a plug-in for Workcraft framework.

I. INTRODUCTION

The main obstacle for the wider spread of asynchronous systems remains to be the inherent complexity of their design. Several solutions are accepted by the industry that ease the design process through abstraction of predefined asynchronous circuit parts as standardised high level components. A designer is able to use these components as “building blocks”, and then obtain the final gate-level design through an automated mapping process. Furthermore, some of the well-known asynchronous design automation packages, such as Tangram [9], and Balsa [5], define a high-level programming-like language that is used to describe systems. The language constructs are then directly translated into a network of *handshake components* – blocks with predefined functionality that use *handshakes* to interface with other components, which are in turn mapped into a gate netlist.

Although this method greatly enhances the designer’s productivity, it has several important drawbacks, of which the control-path overhead is the most decisive. The controllers obtained by syntax-directed mapping are usually far from optimal, because the predefined components are required to implement their declared protocols fully and correctly in order to be reusable in all possible circuit configurations. However, it is often the case that a significant part of their functionality becomes redundant due to the peculiarities of the specific configuration, e.g. in many cases full handshaking between the components can be avoided.

This redundancy can be eliminated by replacing the manually designed gate-level implementation of the high level components with an equivalent STG (signal transition graph) [2] specification. The individual component STGs are then composed together to form a complete system STG [6], which is optimised using PETRIFY [1]. An optimal gate-level implementation can then be automatically produced from the STG using tools such as PETRIFY [1], SIS [8] and MPSAT [3].

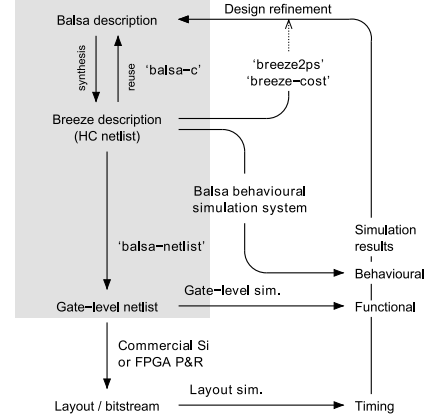


Figure 1: Balsa design workflow

Automatic synthesis becomes problematic when the size of the STG becomes large: modern synthesis tools can handle STGs of no more than 100 signals. The impact of this problem can be lessened by including STG decomposition tools [7] into the workflow, that would break the large optimised STG down into several smaller STGs that are synthesisable in reasonable time. Alternatively, the decomposition step can be carried out on the level of the handshake circuits, dividing the circuit into smaller blocks of components.

This paper proposes an automated method to include the aforementioned modification of the standard design workflow that is used in Balsa design automation system [5] using WORKCRAFT [4] framework.

II. Balsa workflow optimisation through STG RESYNTHESIS

The standard Balsa design workflow is comprised of several stages (Figure 1). The designer writes the system specification in Balsa language. It is passed to the Balsa compiler, which generates a handshake component netlist (produced in a language called Breeze) using syntax-directed mapping on the source code. Syntax-directed mapping in this context means that there is a predefined handshake component construct for every syntactic structure. The Breeze netlist is then translated into a gate-level netlist using direct mapping, this time from individual handshake components to their gate-level implementation, which is defined beforehand.

The proposed modification of this workflow is shown in Figure 2. The translation from Balsa language into Breeze netlist is retained (and is still done by the Balsa compiler), but the Breeze-netlist to gate-level-netlist mapping is replaced with the STG resynthesis flow as introduced in section I. Instead

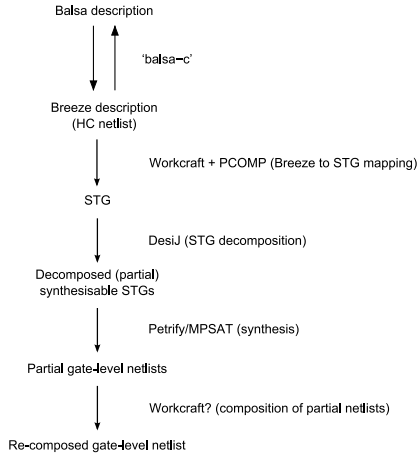


Figure 2: Modified Balsa workflow

of using Balsa tools to produce a gate-level netlist, the Breeze netlist is read by a special interpreted graph model plug-in to WORKCRAFT tool [4], which replaces the handshake components with their STG specifications and produces a composition of those STGs using PCOMP tool. If the resulting STG is small enough, the gate-level implementation may immediately be synthesised using any of the available synthesis tools.

However, for many practical cases the composed STG will become quite large. In this case, to synthesise the implementation it is necessary to insert an additional decomposition step, which may be either STG decomposition (implemented using a tool called DESIJ [7] that is automatically called from the plug-in), or handshake circuit (HC) decomposition which is supported by the plug-in directly. Therefore, the whole process is automated in the WORKCRAFT framework.

The technique allows to synthesise more efficient control circuits while at the same time preserving the benefit of rapid design methodology fundamental to Balsa. It should be noted, however, that full modelling of all Breeze components with STGs is not practical. The behaviour of most data components would be too complex to synthesise from an STG. Circuit resynthesis for such components would take too much time and would often be less effective than an already existing gate-level implementation done by an experienced designer. Subsequently, all data-related functionality in HCs is modelled outside of STG composition framework: the STG models include only control signals for the data path elements. These control signals are to be connected after the gate-level generation step to the data-path circuit that is assembled separately (its components are specified by a structural Verilog netlist). The data path is generated automatically side-by-side with the STG behaviour model.

III. SUPPORT OF BREEZE HANDSHAKE CIRCUITS AS INTERPRETED GRAPH MODEL IN WORKCRAFT

For the purpose of implementation of the design flow discussed in this paper the WORKCRAFT framework was extended with a plug-in that introduces support for Breeze HCs. The new HC model allows WORKCRAFT's convenient

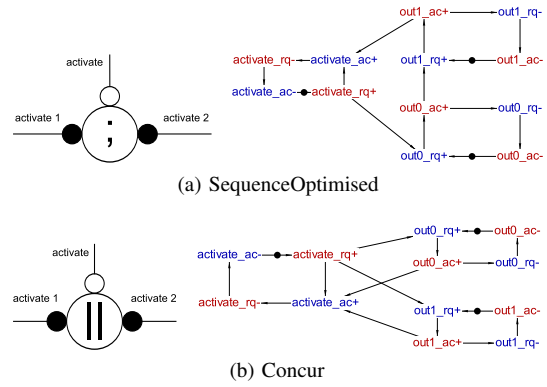


Figure 3: Pure control path handshake components and their respective STGs

visual editing tools to be applied for creation and editing of Breeze netlists. The same plug-in also performs generation of the STG behaviour model for the specified HC. The STG generation algorithm is designed to be highly customisable, with support of multiple handshake protocols and various STG implementations for each type of component. At the moment of writing, STG generation was implemented for a limited set of components using early 4-phase handshake protocol. The library of components is being expanded and will include all Breeze components with support for different handshake protocols.

IV. STG SPECIFICATIONS OF INDIVIDUAL HANDSHAKE COMPONENTS

Balsa components can be roughly divided in three groups: pure control components, data path control components and data-control interface components. We will review each group separately.

A. Pure control path components

Pure control components only control the behaviour of other components and do not carry out any data operations. These components are expected to gain the most from the new design workflow because all of their handshakes are inside the control path and such handshaking does not have to always strictly correspond to the general protocol.

The examples are Concur (Figure 3b) and SequenceOptimised (Figure 3a) components. The STGs in those figures are highly parallel specifications of these components. However, experimental results show that although such implementation might look better on paper, in practise it is sometimes better to specify traditional, more sequential behaviour. This significantly simplifies the task for synthesis tools, particularly those based on state space exploration techniques, because high parallelism often leads to early state space explosion problem. Besides that, a parallel specification suffers more from CSC (complete state coding) problems: a significant number of auxiliary signals have to be introduced to achieve CSC.

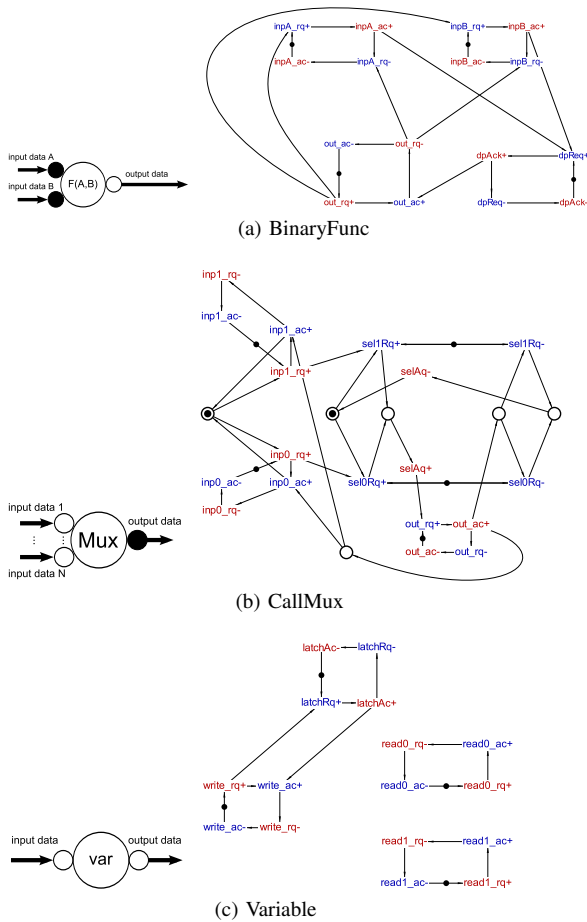


Figure 4: Pure control path handshake components and their respective STGs

B. Data path control components

This group of components is used to control the corresponding data path components that execute predefined operations on data. These operations are far too complex for automated synthesis, but the control path part can still be optimised using STG resynthesis, which makes it reasonable to separate data and control signals. The signals that control the data path are in this case specified as the input and output signals of the component's STG. Because the data path blocks are outside this specification, their handshake protocols must be implemented strictly and thus cannot be optimised. This, however, does not prevent the optimisation of handshakes that belong to the same component but interface with other control path components.

BinaryFunc (Figure 4a), CallMux (Figure 4b), Variable (Figure 4c) are good examples of the data path control components.

C. Data-control interface components

Data-control interface components provide conversion of data to control signals or vice versa. For example, the While component (Figure 5a) analyses the input data to decide whether it should end its operation and conclude the activation handshake, or to continue activating the output handshake.

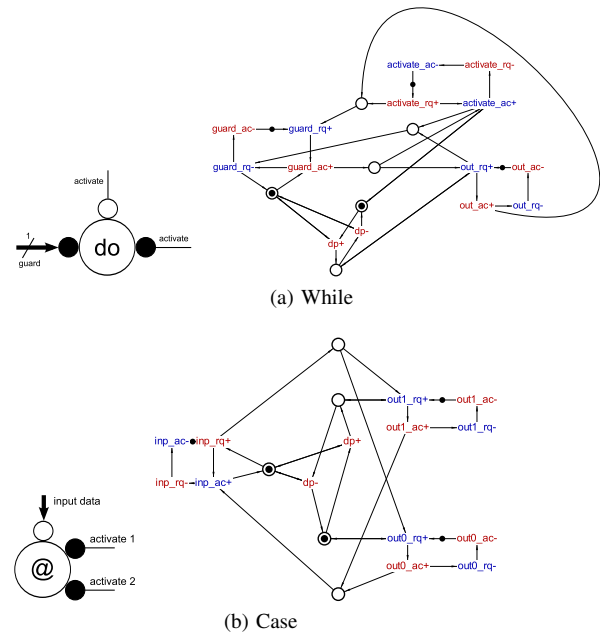


Figure 5: Data-control interface components and their respective STGs

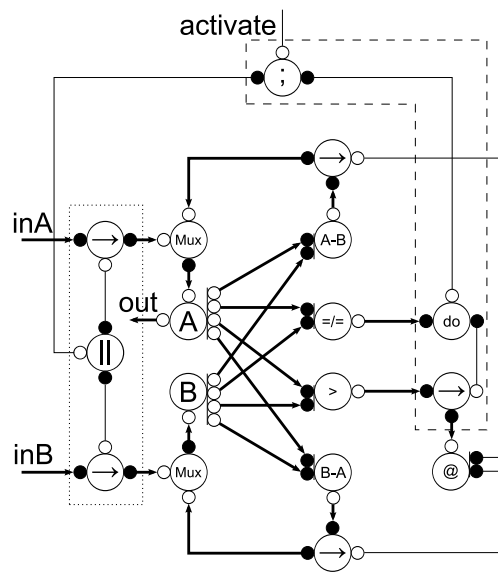


Figure 6: Breeze Handshake Circuit model of a GCD block

Case component (Figure 5b) handles the data in a very similar way, however it has an arbitrary bus width, so for bus widths of more than one bit a decoder that resides in the data path could be used to reduce the STG complexity. These components STGs can become quite complex and the strict behaviour of their data-path handshakes must be preserved.

V. AN EXAMPLE: GCD CONTROLLER

We have chosen the GCD controller (Figure 6) to demonstrate how the proposed technique applies to real-life circuits. The GCD controller is a good research example because it has components from every group described in section IV and its complexity does not allow omitting of the STG decomposition

Component type	MPSAT cost	PETRIFY cost	Best
BinaryFunc	21	27	21
Case	13	13	13
Fetch	17	13	13
Concur	16	16	16
Variable	13	18	13
Sequence	13	13	13
CallMux	25	33	25
While	17	17	17
TOTAL	305	333	285

Table I: Costs of individual components

MPSAT		PETRIFY	
Synthesised block	Cost	Synthesised block	Cost
seq+concur+2xfetch	35	var+2xBF	52
fetch+var+2xBF	49	var+2xBF	52
fetch+var+2xBF	49	2xfetch+case	29
fetch+case	23	fetch+while	29
while	17	seq+concur+fetch	29
callmux	25	callmux	33
callmux	25	fetch	13
		callmux	33
TOTAL	223	TOTAL	270

BEST CHOICE	
Synthesised block	Cost
var+callmux+2xBF	63
fetch+while	29
fetch+case	21
seq+concur+2xfetch	35
callmux	25
fetch+var+2xBF	47
TOTAL	220

Table II: Cost of optimally split full GCD circuit

step, which is an important part of the proposed workflow. All available synthesis tools failed to synthesise a circuit from the fully composed STG model of GCD controller. This proves that the decomposition is a necessary step lacking which the synthesis of a practical circuit is not likely to succeed.

Decomposition on the level of STG can be replaced with decomposition on the level of handshake components. Such decomposition can be done simply by partitioning the input handshake circuit into blocks, trying to minimise the number of handshakes between blocks, and applying the synthesis process to each block separately. While working with the GCD example it was found that decomposition on the level of handshake components can be done easier and is guaranteed to be successful, whereas decomposition on the STG level is a complex task, which requires additional third-party tools.

VI. EXPERIMENTAL RESULTS

For the evaluation of the proposed method effectiveness, each individual handshake component was synthesised separately and its cost (in logic equation literals) estimated. Then, parts of the GCD handshake circuit were synthesised from the STG composition, and the cost of this implementation was compared to the sum of costs of individual components implementations. For synthesis, two tools were used: MPSAT and PETRIFY.

The process of circuit synthesis using this approach was completely automated.

In Table I, the costs of each standalone handshake component, synthesised from the STG specifications, are shown. The results are shown for both applied synthesis tools.

In Table II, the cost of fully synthesised GCD controller is shown. The cost was derived for synthesis carried out by each tool individually, and for the best mix of HC parts produced by both tools, selected on lowest total cost basis (in Figure 6, two such parts are highlighted).

It can easily be seen from the tables that the cost improvement for the GCD circuit was approximately 28%.

VII. CONCLUSIONS

The methodology presented in this paper aims to improve the existing design workflow of Balsa asynchronous synthesis system. A workflow modification was presented that introduces automated synthesis based on STG composition into the traditional Balsa workflow that allows to lessen the control path overheads. The technique was implemented as a plugin to the WORKCRAFT framework and the whole process is automatic. A GCD controller was used as a test example that produced promising preliminary results.

Acknowledgement: {information removed for blind review}

REFERENCES

- [1] Petrify tool. reference withheld for blind review. 1997.
- [2] Reference withheld for blind review. 1998.
- [3] MPSat tool. Reference withheld for blind review. 2004.
- [4] Workcraft tool. Reference withheld for blind review. 2009.
- [5] D. A. Edwards and A. Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45 (1):12–18, jan 2002.
- [6] Marco A. Pena and Jordi Cortadella. Combining process algebras and petri nets for the specification and synthesis of asynchronous circuits. In *ASYNC '96: Proceedings of the 2nd International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 222, Washington, DC, USA, 1996. IEEE Computer Society.
- [7] Mark Schaefer. DesiJ - A Tool for STG Decomposition . Technical Report tr-11-2007, University of Augsburg, October 2007.
- [8] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Sis: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.
- [9] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The vlsi-programming language tangram and its translation into handshake circuits. In *EURO-DAC '91: Proceedings of the conference on European design automation*, pages 384–389, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.