Software Visualisation Techniques Adapted and Extended for Asynchronous Hardware Design

Lilian Janin, Doug Edwards
School of Computer Science
The University of Manchester
Manchester M13 9PL
{lilian.janin, doug.edwards}@cs.man.ac.uk

Abstract

Asynchronous circuit design shows many similarities with software design. This is due to the modular construction style associated with asynchronous circuits, where each asynchronous module can be designed as a standalone object communicating with other modules. We propose a visualisation system for asynchronous circuit design tools, based on clustered graph visualisation and coordinated views. The novelty of our approach is to apply classical software visualisation techniques to a hardware environment and take advantage of properties specific to our asynchronous design flow. This system is based on a control and data flow graph representation of the asynchronous circuit compiled from a high-level description by a syntax-directed transparent process and transferable to the final hardware circuit by a direct synthesis process. Compared to software compilation flows, this transparent design flow offers opportunities for visualisation, with representations at different stages being easily mapped onto each other to combine their visualisation qualities. Each representation also exhibits properties based on some physical attributes of the final hardware circuit, allowing us to render some abstract properties on top of real-world-based ones. This paper shows how the handshake circuit graph is used as the underlying structure onto which properties extracted from other structures are mapped, and then how this structured graph is used as the central piece in a coordinated views environment. These visualisation techniques have been exercised by designers optimising the design of a fully asynchronous ARM processor.

Keywords: Software visualisation, coordinated views, program comprehension, asynchronous circuits

1. Introduction

Asynchronous circuits are electronic systems able to operate without global synchronisation between their different parts. They are usually designed as sets of standalone modules communicating and synchronising themselves locally with a restricted set of neighbours, forming a control and data flow graph. The aim of the visualisation system described in this paper is to provide a visualisation environment for asynchronous circuit tools, able to be used in three situations: first, as a program comprehension environment, able to represent a large circuit at different scales with the appropriate communications between the viewed objects. Second, as a debugging environment, able to help the designer locate asynchronous-specific bugs such as deadlocks. And third, as an educational environment, able to depict step by step the communication protocols between modules or the parallel activity in simple circuits.

One of the hardware description languages for describing asynchronous circuits at a high-level is the Balsa language [1]. The compilation of a Balsa description into a hardware circuit (a process called *synthesis*) is based on the handshake circuit intermediate representation, which offers excellent opportunities for visualisation:

- Handshake circuits are graphs, very amenable to visualisation.
- Balsa is based on a syntax-directed compilation, where source code constructs are transparently translated into similar components in the handshake circuit, themselves translated into sets of hardware logic. The structure of the final hardware and the intermediate handshake circuit are very similar to the original description's structure. This is an excellent opportunity for coordinated views and multi-level analysis.

- The original Balsa description, the best reference to the designer's mental image of his circuit, has a well defined, precise and hierarchical structure made of nested procedures and block statements. This provides invaluable information for building an efficient representation for circuit comprehension.
- Storey [13] defines a visualisation as *coherent* if the maintainer can construct from the given visualisation a mental model which corresponds to something in the real world. The advantage of visualising reverse engineered code of a VLSI design is that the code describes something concrete, the electronic circuit being built. It is therefore possible to use this real-world structure as an underlying base, and either transfer this structure onto the other structures being visualised, or use other sources of information to reshape (for example by clustering) the real-world structure.

This paper explains how a visualisation system based on the handshake circuit paradigm can take advantage of these opportunities for displaying fine-grained parallelism efficiently and for constructing an environment suitable for program comprehension, debugging and education purposes.

First, the handshake circuit graph is used as the underlying structure onto which properties extracted from other structures are mapped. Then, this structured handshake circuit graph, exposing the properties of the different structures, is used as the central piece in a coordinated views environment. These techniques are aimed at visualising large circuit designs (e.g. processor cores) and therefore large amounts of information.

2. Related work

The visualisation system presented in this paper is based on two main techniques: clustering information by merging multiple sources into one common structure and using coordinated views for information exploration. Other visualisation systems based on these techniques have been studied for software visualisation. This section introduces those related to this research.

2.1. Data clustering by organising knowledge from multiple sources

Clustering is the process of discovering groupings or classes in data, based on a chosen semantics. Clustering techniques have been referred to in the literature as *cluster analysis, grouping, clumping, classification,* and *unsupervised pattern recognition* [2, 8]. Two forms of clustering can be distinguished: *structure-based* clustering, which refers to clustering that uses only structural information about the graph, and *content-based* clustering, which uses the semantic data associated with the graph elements to perform clustering. An advantage of structure-based clustering is that clusters retain the structure of the original graph, which can be useful for user orientation in the graph itself. However, this class of methods often leads to the clustering of elements poorly related in their properties. Content-based clustering can yield groupings which are more appropriate for a particular application by using application-specific data and knowledge [9, 11].

By far the most common clustering approach in graph visualisation is to find clusters that are disjoint or mutually exclusive, as opposed to clusters that overlap (found by a process called *clumping*). Disjoint clusters are simpler to navigate than overlapping clusters because a visit of the clusters only visits the members once. It should be noted, however, that it is not always possible to find disjoint clusters, for instance in the case of language-oriented or semantic topologies [6]. This is one of the strengths of the techniques presented here using Balsa.

If clustering is performed by recursively applying the same clustering process to groups discovered by a previous clustering operation, the process is referred to as *hierarchical clustering* [8]. Hierarchical clustering can be used to induce a hierarchy in a graph structure that might not otherwise have a hierarchical structure. Michaud et al. have described Shrimp [7], which gathers various sources of information together (source code artifacts and relationships, architectural abstractions, documentation and history information, metrics and analysis information) and visualise them together. The difference between their work and the one presented here is that they limit their study to static information, while dynamic simulation data is also used here to deduce some clusters. Hierarchical clustering is applied here to the originally flat handshake circuit.

2.2. Information exploration with coordinated views

In VLSI design, designers are used to working with raw data: source files opened in a standard editor, simulation traces displayed as wave forms, etc. Keeping these standard views is important, and the only way to bring designers to use new visualisation techniques is to provide them as optional but easily accessible elements. One way to achieve this is to use coordinated views in order to link well-understood items from the standard views to structures coming from the new visualisation techniques.

Favre [3] describes such a collection of views representing the same information in different manners. In his research, a component-based software is represented by three techniques: first, by representing the network graph of its components. Then, by displaying a list structure containing the object-oriented hierarchical structure. And finally, by showing the source code. Other views are also available to display either the internal representation or the external representation of a selected component. The drawback of this study is the lack of communication between views, restricted to the selection of a component.

When dynamic information is visualised and various views are representing the program's state during a given timestep, these views need to be synchronised to represent the same timestep. The Vista architecture [14] is an example of such an organisation where every view is connected to, and controlled by, a server process (called Visualization Manager). In this architecture however, views do not send any feedback to the visualization manager. They therefore do not communicate with each other either.

Software debuggers like DDD [15] or VIPS [12] follow the same idea of sharing the time variable between views: the source code view indicates the currently executed line, while another view can show the current value of some data. But DDD goes further by adding some real communication between views: from the source code view, one can select a variable and choose to add it in the other view for tracing. The other way around, other views such as the stack view and the thread view can send some feedback and cause changes in the source code view.

The best reference about collaborating views is Shrimp [7], showing some real similarities with this thesis's work. Four sources of information are collected and represented in multiple views, some of them being network graphs representing the structure of the system. They implement a technique called *control integration*, which implies the ability for one tool to control another tool, either by directly activating a functionality or by event notification. The difference with this work is that Shrimp is targeted at exploring static information. The execution of the visualised Java program is not represented.

3. Handshake circuit visualisation

Handshake circuits are used as an intermediate format during the Balsa synthesis process. The difficulty with handshake circuits compiled from Balsa descriptions is their large size: it is not uncommon for handshake circuit graphs to have thousands of edges and vertices, usually referred to as "huge graphs" in the literature. Specific methods for structuring the graph before visualisation are therefore needed. Trying to reproduce the designer's mental image of his circuit is generally a good starting point.

The following references to the designer's mental image can be obtained from the Balsa development flow:

- The written Balsa source code.
- The generated handshake circuit – a good designer will be able to anticipate the generated circuit when writing Balsa code.
- The execution/simulation trace: During the design of a circuit, the designer always anticipates the amount of information flowing on the different channels/buses and bases his circuit's architecture on this information.
- The visualisation software interface with the user: Human interaction can be used to designate important regions of the circuit and to correct software guesses about the architecture, etc.

All these elements coming from the designer's mind to create the circuit are gathered together to structure the raw handshake circuit graph, hopefully making for an intelligible representation.

3.1. Static multiscale visualisation

Static visualisation is based on clustering techniques to structure large handshake circuit graphs into more manageable groups, by analysing and combining the sources of information available: the Balsa source code, the compiled handshake circuit and the simulation trace.

Combining multiple sources of information offers the following benefits:

- This information can be used to cluster some components from one source by using the information from another source. Clustering eventually allows the number of elements to be processed at a time to be reduced by processing them by group instead of individually.
- Sources usually used to visualise at different scales are combined to make a graph viewable at any scale.
- More clues are available to reconstitute the designer's mental image of the circuit.

A static view of the handshake circuit is constructed from the hierarchical graph obtained after clustering. This graph view enjoys the above-mentioned benefits.

Starting from the raw handshake circuit graph, this section shows how clusters of handshake components can be formed by using other related sources of information and how they can be used for understanding and debugging asynchronous circuits, in particular from the point of view of concurrency visualisation.

These methods correspond to static allocations of the groups: although they may need some information taken from the simulation in order to be determined, the clusters do not change dynamically during the simulation.

3.1.1. Description-based clustering. The network of handshake components is derived from the high level description of the circuit in Balsa, which itself is organised by structural information such as procedures, functions, instruction blocks and local variables. Given the close relationship (due to transparent syntax-directed translation) between the Balsa description and the generated handshake circuit, it is logical to try to transfer this high-level structure onto the lower-level handshake circuit in order to partition this huge network into more manageable chunks.

A Balsa description contains procedures and local sub-procedures inside these procedures (as well as the other local structures: functions, instruction blocks and local variables), resulting in nested groups and subgroups in the graph of the handshake circuit. Experiments on large circuits show that functional grouping applied to procedures and functions usually divides a graph of n elements into groups of about $\sqrt{n}$ elements. Reasonably large networks of ten thousand handshake components are therefore divided into groups of about hundred elements. However, this number is an average: many groups are smaller than a hundred elements, while a few groups can reach thousand items or more, which is still too large for an efficient visualisation.

Some experiments have been made about using every level of instruction block (i.e. blocks of instructions contained between language keywords, such as the division: if <block> then <block> else <block> end) in the clustering process. This unfortunately results in a huge, and thus difficult to manage, quantity of nested groups where the original circuit gets divided into small groups of usually less than five elements: the clusters themselves are wasting the entire visualisation area and are as difficult to organise as the original flat graph. This technique has also been applied while limiting the clustering depth or by setting a minimal number of elements per group, but with limited success.

In practice, groups based on procedures appear to be the most useful ones during visualisation. These groups improve the readability of the graph in a general manner by highlighting a structure to which the designer is familiar. This is an important part of the mental image reconstitution.

3.1.2. Circuit-based clustering. Circuit-based clustering uses the handshake circuit information to group components together. Due to the transparent compilation of Balsa, the information present in the handshake circuit can always be found in more or less the same form in the original Balsa description, but some characteristics are better reflected by handshake components and channels than others. Three types of clustering methods are exposed here, the first one being also detectable at the description level, and the last one being highly specific to handshake circuits.

Design pattern clustering: The term *design pattern* is used here with the same meaning as in object-oriented software development. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context [4].

This idea of exploiting design patterns for asynchronous circuit design with Balsa has not been thoroughly exploited yet. However, in the context of thread visualisation, one particular design pattern appears in every medium-to-large circuit. It describes the way concurrent data-processing asynchronous blocks are designed in order to be initialised when the circuit is switched on and subsequently run forever, processing inputs and generating outputs. Each block is made of an initialisation part followed by either some calls to other blocks following the same pattern, or a forever loop containing the functional body of the module, as represented in Figure 1. The successive blocks are used to perpetrate the activation signal down to the various functional bodies, and are constituting an activation tree. Functional bodies executed inside their forever loops are at the leaves of the tree. Variable and channel initialisations are also at the leaves of the tree, but typically have a less important role.

When compiled into a handshake circuit, this pattern appears as recognisable sets of components, shown in Figure 1b:

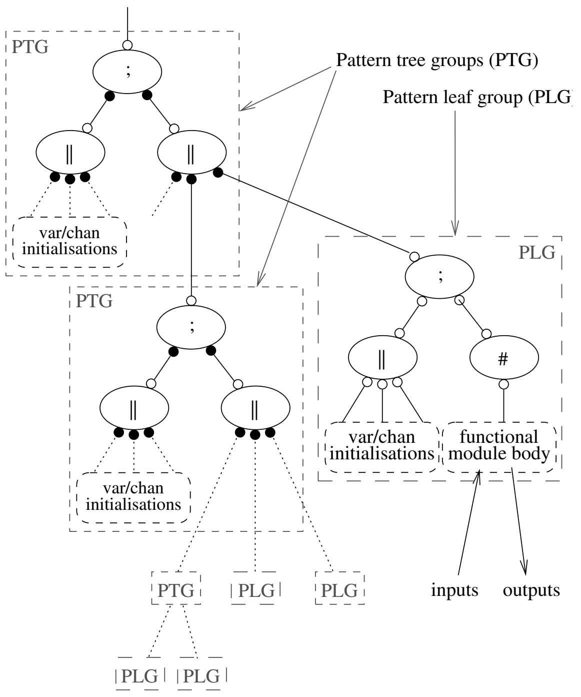
- Both the pattern_tree and the pattern_leaf groups are starting with a 2-output Sequence handshake component, whose first output activates a Concur component used for variable and channel initialisations.
- In the case of the pattern_tree group, the second output is used to propagate the activation signal to the next modules via a Wirefork handshake component.
- In the case of the pattern_leaf group, the second output is connected to a Loop handshake component (forever loop) controlling the module's functional body, and processing input/outputs.

When detected as being part of a pattern, the sets of components can be treated specifically. In the current

```
procedure Pattern_tree (inputs; outputs) is
local -- local variable definitions
begin
    -- parallel initialisation of local variables and channels
    variable := 0 ||
    channel := 0 ;
    -- sequentially followed by sub-procedure calls
    SubProcedure1a (inputs, outputs) ||
    SubProcedure1b (inputs, outputs)
end

procedure Pattern_leaf (inputs; outputs) is
local -- local variable definitions
begin
    -- parallel initialisation of local variables and channels
    variable := 0 ||
    channel := 0 ;
    -- sequentially followed by forever loop
    loop
        -- code executed forever, waiting for inputs,
        -- processing it, and sending result to outputs
    end
end
```

a. Balsa description level



b. Handshake circuit level

Figure 1. Circuit initialisation design pattern

application, the components part of the initialisation and of the forever_loop sets are clustered and the wirefork tree can be shown in the handshake circuit in a different colour. The initialisation clusters and the wirefork tree are clearly only used during the initialisation phase of the circuit. The ability to make them invisible during the rest of the simulation accounts for a better readability of the graph.

Parallel composition clustering: Preparing for the visualisation of concurrent threads at the handshake circuit level involves handling those handshake components responsible for the creation of new threads: Fork, Wirefork and Concur components. Their roles are identical: forking one thread into *n* new threads. Only their implementations differ: Wirefork components do not expect the newly created threads to ever finish, while Fork and Concur components differ in the way they interleave the return-to-zero (RTZ) phase present in some asynchronous handshake protocols (Fork components wait for every thread to acknowledge its completion before sending every RTZ request at once, while Concur components send the RTZ request to each thread individually directly after it acknowledged).

In practice, unfortunately, clustering threads by starting from the components that created them results in a degradation of the readability of the visualisation. This is explained by the programming style of the designers, who use parallel statement most of the time in one of these two situations:

- either the newly created block contains only a few statements, in which case it is inlined between other code statements,
- or the newly created block is larger than a few statements, in which case it is described as a procedure.

Almost never a large block is kept inlined in a parallel composition statement. For this reason, clustering these threads results either in clustering only a few components, or in clustering an already clustered procedure.

Variables clustering: Two functional groupings providing a better graph readability concern variables.

The first situation is when a Balsa variable is compiled into one Variable handshake component. Variable components have, by design, only a single write port. When the Balsa code contains more than one writer to the same variable, a tree of CallMux (data merges) components is used to combine writers from all sources into one. In the visualisation, this tree has no reason to belong to the same group as one of the writers. It should then appear next to the Variable component, hence the grouping. The result of this grouping does not have any significant consequence on the number of groups and the number of components inside groups. However, the components grouped together by this method have a strong relationship, which highly improves the visualisation, as such groups may transparently be reduced to single elements representing variables with many write ports.

Sometimes, Balsa variables are distributed into more than one Variable component. This is for processes needing only a few bits of a variable to avoid reading the whole variable (and therefore holding this resource). When the whole variable is required, a tree of Combine components is used to reconstitute the data from its parts. In the same way as with the writers' tree, this tree of components may be clustered with the Variable component for increased readability.

3.1.3. Simulation-based clustering. Simulation-based clustering uses the simulation trace to determine a relationship score for each couple of components, based on the amount of control and data sent between these components during the simulation. This score, proportional to the activity in the handshake channels, is used as a weight for the corresponding handshake circuit graph's arcs. The more activity in a channel during simulation, the shorter the associated arc and the closer the linked components. By using graph layout algorithms able to take weights into account, this technique groups the communicating components together without any explicit clustering. This results in a *loose* clustering scheme where highly communicating components can still be taken apart by other predominant clustering techniques.

As with every trace-based technique, this technique is dependent on the test data used during the simulation (e.g. in the case of a Balsa-described processor being simulated, the trace will be dependent on which application is chosen to run on the simulated processor). The designer has the choice of using this clustering technique or not. Sometimes, an unbiased visualisation may be more appropriate than a clearer but biased visualisation.

3.1.4. Additional user specifications. Additional user specifications are largely used in algorithm visualisation, where the original description often needs to be instrumented with visualisation instructions. This makes the process of representing the designer's mental image much easier, as the designer himself provides the instructions. However, it also requires the designer to understand the code before visualising it, making this technique infeasible for large systems or tasks involving program discovery.

This present research is aimed at automating as many tasks as possible in order to relieve the designer of some repetitive interactions. However, first experiments concluded that, although graphs generated using the above-mentioned techniques are appropriately structured, designers would have chosen different symbols to represent specific electronic components. For example, the ALU (arithmetic logic unit) inside a processor is always represented by a well-known symbol, which is of course impossible for a software to guess without any prior knowledge.

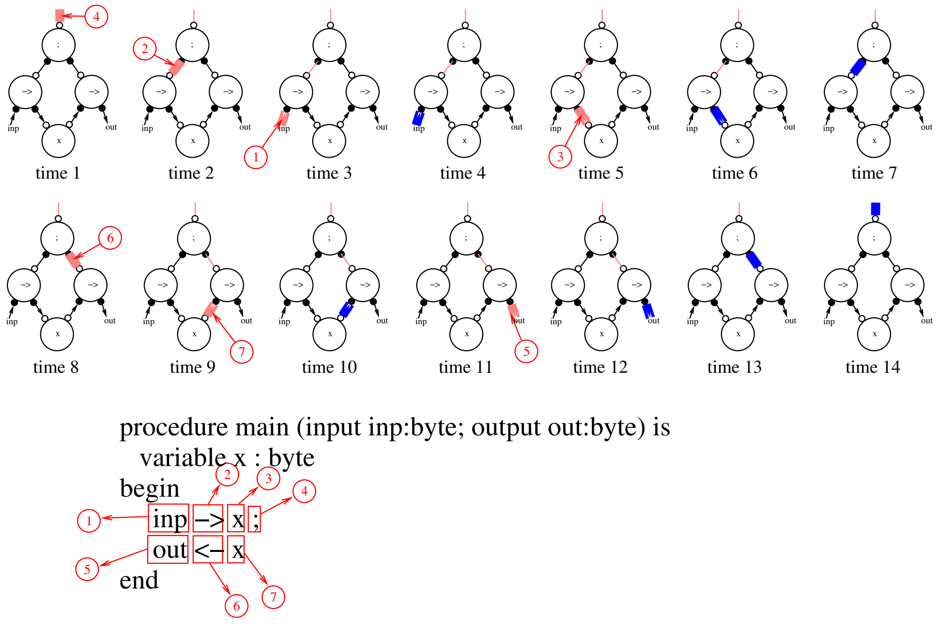
An interface for designers to teach this knowledge to the software is therefore needed. One possibility is to let designers assign shapes to procedures. This is useful but only manageable for small designs or small numbers of shape-assigned procedures. This also becomes quickly a tedious process when designers decide to assign a shape to every type of buffers or multiplexers they described as distinct procedures.

3.1.5. Multiscale structure. As stated above, clustering techniques not only have the beneficial effect of separating a huge number of handshake elements into fewer manageable groups. They also have the important consequence of transferring the structure of each analysed source of information onto the handshake circuit. The resulting hierarchical structure exhibits the advantages of each of the source structures.

One important implication comes from the fact that different sources of information are usually used to visualise the structure and behaviour of a circuit at different scales (or level of detail): the hierarchy of Balsa procedures gives a high-level representation of the description, data and control flows can show an intermediate level, while traced events happen on handshake channels at a low-level. Therefore, the structure obtained after clustering can be visualised at any scale and always shows useful information: from the global view of the circuit to the lowest level, the main components of the circuit can be distinguished, followed by the (sometimes recursive) high-level implementations of the modules, and on until the detailed implementation of the modules, precise enough to visually understand their behaviour.

3.2. Dynamic visualisation

The previous section introduced techniques to build a static representation of the circuit structure by clustering the components of the circuit in an easily readable way. Based on the simulation trace, the role of the ani-



```
procedure main (input inp:byte; output out:byte) is
  variable x : byte
begin
  inp -> x ;
  out <- x
end
```

Figure 2. Step by step animation of a 1-place buffer

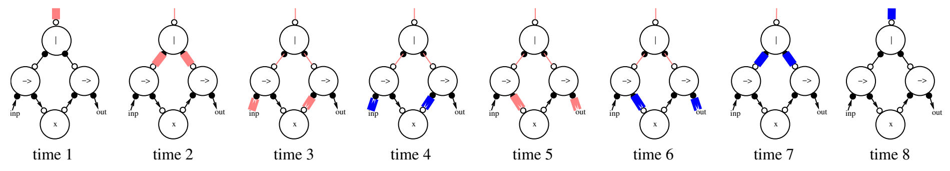


Figure 3. Step by step animation of a hypothetical parallel circuit

mation module is to add further information to the static picture in order to represent data and control flows, and the changing activity of the components during simulation.

This is achieved by marking the handshake circuit graph with colour annotations: each component or channel state is represented by a colour, and the circuit is animated as the simulation system updates the states of the components and channels.

Figure 2 shows a step-by-step animation of the activity present in a very small part of a processor: a 1-place buffer used to store a value in a buffer and transmit it to the next processing stage. The associations between handshake channels and pseudo-source code are represented, thus indicating the meaning of each coloured event. In this animation, a 2-phase protocol is used where requests are represented by a thick red (light gray on black&white prints) highlighting scheme and acknowledgments by a thick blue (or dark gray) one. Thin red channels represent channels which have been activated at a previous timestep and have not been acknowledged yet.

By following the same scheme, concurrent flows and their interactions can be easily observed. For example, Figure 3 shows the same circuit as above, but where the Sequence component has been replaced by a Fork component. It should be noted that the Balsa compiler would not generate such a circuit unless forced (by the designer) to do so: the concurrent read and write accesses to the variable are non-deterministic.

The advantage of such an animation system is its ability to show all the information available from the Balsa description and from the execution of the simulation of the system, and then let the user decide what he wants to focus on. Debugging is made easier through the visualisation of the parallel activity: every thread of execution of a simulation can be shown simultaneously, and the observer can focus on one specific thread, observe its activity, and can easily observe its merging with another thread or its splitting into two threads.

This animation system also provides some interesting debugging features for deadlocks and livelocks. When a deadlock situation arises, the program stops, leaving the guilty components in a specific colour and the trace of the components before them in another colour, making it less difficult to debug. In a livelock situation, the colours can be observed circling in an endless loop, but while this identifies the components involved it hardly indicates the entry condition.

Another advantage with viewing synthesised electronic circuits rather than compiled software is the absence of "shared memory". In software programs, an instruction at one particular memory address can be executed by multiple concurrent threads. On the opposite, in a hardware electronic circuit, the same transistor cannot be controlled by two "threads" simultaneously. The benefit of this situation is that concurrent threads never overlap in a handshake circuit (or lower level) view.

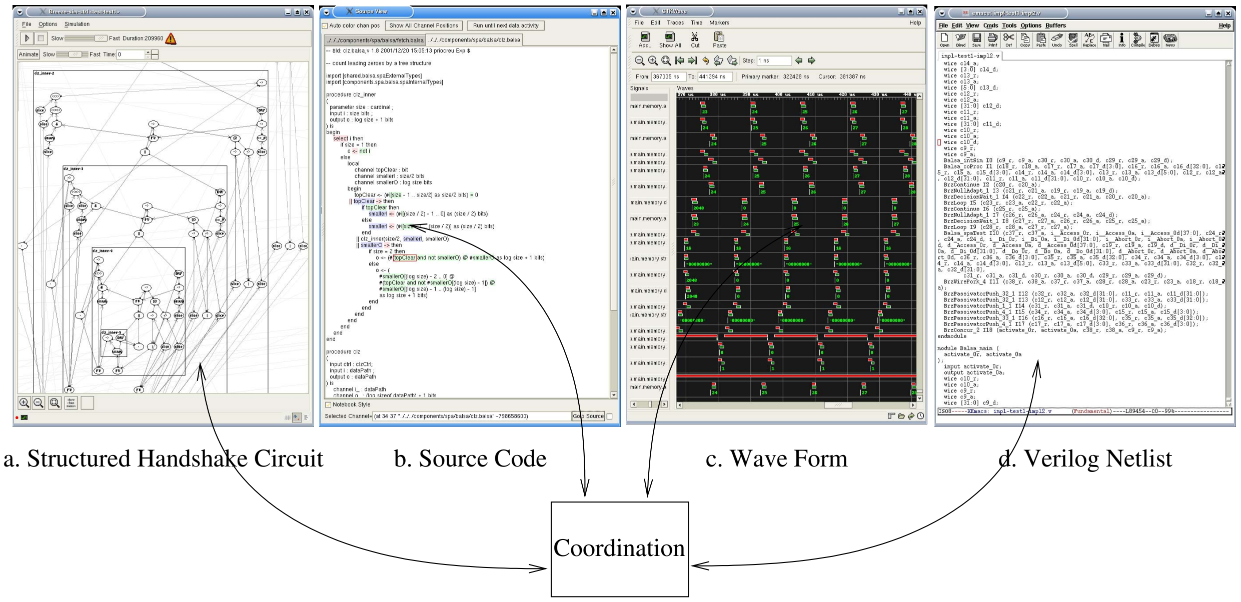
Finally, the one-to-one correspondence between the Balsa description and the visualised handshake components makes it easy to link any error located on the visualised circuit with its corresponding location in the Balsa description.

4. Coordinated views

Until now, the visualisation system has been used to visualise multiple sources of information together in a single view. The benefits of such a view have been presented. However, in order to be useable and intuitive, a visualisation system must also take into consideration what the user wants to see. Most of the time, the user/designer wants to continue to use the same style of design he has always used. In the case of asynchronous design with Balsa, the views usually consist of a text editor containing the source code, and a waveform viewer to analyse the results of the simulation. In order to make these views even more useful, a collaboration scheme is suggested in this section to track the visualised elements and navigate efficiently from one view to another.

The main view (Figure 4a) is accompanied by a number of other views, representing the designed system at other levels:

- **Source code view** (Figure 4b): This view is a text viewer showing the Balsa source code. The designer's preferred text editor can also be used. Although this is the simplest of all the views when used individually, this is the most difficult to link bi-directionally to the rest, as simple text viewers are not generally designed to display anything but text or to forward keyboard/mouse events.
- **Waveform view** (Figure 4c): GTKWave [5] is an ex-


Source View
Auto color chan pos
Show All Channel Positions
Run until next data activity
../../../components/spa/balsa/fetch.balsa
../../../components/spa/balsa/clz.balsa
Notebook Style
Go to Source
GTKWave
emacs: impl-test1-impl2.v
Coordination

```
-- $Id: clz.balsa,v 1.8 2001/12/20 15:05:13 priocreu Exp $

-- count leading zeroes by a tree structure

import [shared.balsa.spaExternalTypes]
import [components.spa.balsa.spaInternalTypes]

procedure clz_inner
(
  parameter size : cardinal ;
  input i : size bits ;
  output o : log size + 1 bits
) is
begin
  select i then
    if size = 1 then
      o <- not i
    else
      local
        channel topClear : bit
        channel smallerI : size/2 bits
        channel smallerO : log size bits
      begin
        topClear <- (#i[size - 1 .. size/2] as size/2 bits) = 0
       || topClear -> then
          if topClear then
            smallerI <- (#i[(size / 2) - 1 .. 0] as (size / 2) bits)
          else
            smallerI <- (#i[size - 1 .. (size / 2)] as (size / 2) bits)
          end
        || clz_inner(size/2, smallerI, smallerO)
        || smallerO -> then
            if size = 2 then
              o <- (#topClear and not smallerO) @ #smallerO as log size + 1 bits)
            else
              o <- (
               #smallerO[(log size) - 2 .. 0] @
               #(topClear and not #smallerO[(log size) - 1]) @
               #smallerO[(log size) - 1 .. (log size) - 1]
               as log size + 1 bits)
            end
          end
        end
      end
    end
  end
end

procedure clz
(
  input ctrl : clzCtrl;
  input i : dataPath ;
  output o : dataPath
) is
  channel i_ : dataPath
  channel o_ : (log sizeof dataPath) + 1 bits
```

Selected Channel= (at 34 37 "../../../components/spa/balsa/clz.balsa" -798658600)

```
  wire c14_a;
  wire [3:0] c14_d;
  wire c13_r;
  wire c13_a;
  wire [5:0] c13_d;
  wire c12_r;
  wire c12_a;
  wire [31:0] c12_d;
  wire c11_r;
  wire c11_a;
  wire [31:0] c11_d;
  wire c10_r;
  wire c10_a;
  wire c10_d;
  wire c9_r;
  wire c9_a;
  Balsa_intSim I0 (c9_r, c9_a, c30_r, c30_a, c30_d, c29_r, c29_a, c29_d);
  Balsa_coProc I1 (c18_r, c18_a, c17_r, c17_a, c17_d[3:0], c16_r, c16_a, c16_d[32:0], c1
5_r, c15_a, c15_d[3:0], c14_r, c14_a, c14_d[3:0], c13_r, c13_a, c13_d[5:0], c12_r, c12_a
, c12_d[31:0], c11_r, c11_a, c11_d[31:0], c10_r, c10_a, c10_d);
  BrzContinue I2 (c20_r, c20_a);
  BrzNullAdapt_1 I3 (c21_r, c21_a, c19_r, c19_a, c19_d);
  BrzDecisionWait_1 I4 (c22_r, c22_a, c21_r, c21_a, c20_r, c20_a);
  BrzLoop I5 (c23_r, c23_a, c22_r, c22_a);
  BrzContinue I6 (c25_r, c25_a);
  BrzNullAdapt_1 I7 (c26_r, c26_a, c24_r, c24_a, c24_d);
  BrzDecisionWait_1 I8 (c27_r, c27_a, c26_r, c26_a, c25_r, c25_a);
  BrzLoop I9 (c28_r, c28_a, c27_r, c27_a);
  Balsa_spaTest I10 (c37_r, c37_a, i__Access_0r, i__Access_0a, i__Access_0d[37:0], c24_r
, c24_a, c24_d, i__Di_0r, i__Di_0a, i__Di_0d[31:0], i__Abort_0r, i__Abort_0a, i__Abort_0
d, d__Access_0r, d__Access_0a, d__Access_0d[37:0], c19_r, c19_a, c19_d, d__Di_0r, d__Di_
0a, d__Di_0d[31:0], d__Do_0r, d__Do_0a, d__Do_0d[31:0], d__Abort_0r, d__Abort_0a, d__Abo
rt_0d, c36_r, c36_a, c36_d[3:0], c35_r, c35_a, c35_d[32:0], c34_r, c34_a, c34_d[3:0], c1
4_r, c14_a, c14_d[3:0], c13_r, c13_a, c13_d[5:0], c33_r, c33_a, c33_d[31:0], c32_r, c32_
a, c32_d[31:0],
      c31_r, c31_a, c31_d, c30_r, c30_a, c30_d, c29_r, c29_a, c29_d);
  BrzWireFork_4 I11 (c38_r, c38_a, c37_r, c37_a, c28_r, c28_a, c23_r, c23_a, c18_r, c18_
a);
  BrzPassivatorPush_32_1 I12 (c32_r, c32_a, c32_d[31:0], c11_r, c11_a, c11_d[31:0]);
  BrzPassivatorPush_32_1 I13 (c12_r, c12_a, c12_d[31:0], c33_r, c33_a, c33_d[31:0]);
  BrzPassivatorPush_1_1 I14 (c31_r, c31_a, c31_d, c10_r, c10_a, c10_d);
  BrzPassivatorPush_4_1 I15 (c34_r, c34_a, c34_d[3:0], c15_r, c15_a, c15_d[3:0]);
  BrzPassivatorPush_33_1 I16 (c16_r, c16_a, c16_d[32:0], c35_r, c35_a, c35_d[32:0]);
  BrzPassivatorPush_4_1 I17 (c17_r, c17_a, c17_d[3:0], c36_r, c36_a, c36_d[3:0]);
  BrzConcur_2 I18 (activate_0r, activate_0a, c38_r, c38_a, c9_r, c9_a);
endmodule

module Balsa_main (
  activate_0r, activate_0a
);
  input activate_0r;
  output activate_0a;
  wire c10_r;
  wire c10_a;
  wire c9_r;
  wire c9_a;
  wire [31:0] c9_d;
ISO8-----XEmacs: impl-test1-impl2.v      (Fundamental)----L89454--C0--99%----------------
```

a. Structured Handshake Circuit   b. Source Code   c. Wave Form   d. Verilog Netlist

Figure 4. Coordinated views

ternal program used to display waveforms of the handshake channels. It is directly and bi-directionally linked to the handshake circuit visualisation system, which provides the user with an interface to select which channels are to be displayed in GTKWave. In return, GTKWave can be used to select some channels and periods of time of interest over which some processing actions can be executed by the visualisation system, such as calculating the power consumption of a sub-circuit over a certain period of time.

- **Verilog description** (Figure 4d): Verilog code is generated from the Balsa description and corresponds to a direct translation from the handshake circuit netlist.

In each of these views, a subset of the whole information set (source code, handshake circuit, simulation trace) is represented. The challenge is to link together the different representations of a same item in different views. This link is used for synchronising views when a component's properties are changed in one of them. More importantly, it also allows the user to track elements efficiently between one view and another, e.g. to jump from a source code statement to the corresponding channel in the handshake circuit view.

The visualised elements are categorised into three groups:

- The *simple* elements originally contained in the Breeze and simulation trace files: procedures, ports, channels, components, time and events. They are visualised in most of the views.
- The *compiled* elements: data and control flows, states. These are generated after analysis of the simple elements.
- The Balsa statements, present only in the original Balsa source code and visualised only in the source code view.

A subset of these elements is used as links between views:

- **Handshake channels:** Handshake channels are first generated in Breeze files. They are directly and fully visualised in most of the views and are therefore the preferred means for going from one view to another. They are the most fine-grained components of handshake circuits, and as such provide efficient ways of manipulating and associating the different views at a low level of the design. In addition to themselves being the link between different views, handshake channels contain the source code position of the Balsa statement they have been compiled from. This allows every view containing channels to report references to the original source code, necessary for correctly reporting errors to aid the debugging process.
- **Handshake components:** Handshake components are the base execution blocks of handshake circuits. They can usually be logically associated with Balsa operations. The link between a handshake component and the corresponding source code is implemented by the intermediate of handshake channels. Components are connected to channels, among which one channel can always be seen as 'more special' than the others: Variable components have a unique write port, BinaryFunctions components have a unique output, most components have a unique activation port, etc. The association between a component and the source code is made through this special channel.
- **Procedures:** Procedures are described in Balsa and compiled as groups of handshake channels and components. Their coarse grain allows the user to manipulate large circuits at a high level before going into the details of handshake channels and components.
- **Simulation time:** Simulation time is an easy parameter to deal with: a single number allows to specify the current simulation time in a view and see the correspondence in other views. The only slight difficulty is due to the asynchronous nature of the circuits being designed, which let events happen at any time instead of at regular intervals of time as would be the case with synchronous circuits.
- **Source code elements:** Source code elements (or statements) constitute the special case of this section: they are not represented directly but are "cross-referenced" by other elements via their position in the source code files. Balsa source code statements are first referenced in Breeze files as channel positions. This work extends this to any component that can be associated with a channel, such as handshake components and simulation events. The source code position is the invisible medium of the association between Balsa statements and visualised elements.

Views communicate through a *Coordinator* process, which gathers and broadcasts events happening to the linking elements (e.g. element selection and modification). This keeps all the views synchronised and allows the user to navigate from one view to another by selecting an element as the central piece.

5. Results

These techniques have been developed and integrated in the Balsa framework. The visualisation system has been used by designers optimising the design of SPA, a Balsa-described ARM compatible processor

[10]. This processor is the largest circuit synthesised with Balsa so far, and the SPA description is compiled into a handshake circuit comprising around ten thousand elements.

The clustering techniques are able to structure efficiently this graph for visualisation. After clustering, the static handshake circuit structure can be viewed at different levels of detail, with different information appearing automatically at different scales.

The dynamic colour-based representation makes it easy to visualise concurrent threads on the handshake circuit, and the coordinated views enable the user/designer to navigate rapidly between the different views by selecting some elements detected during thread visualisation.

Our current problem lies in that no particular method has been found for representing concurrent threads in the source code view. This therefore limits source code debugging to a single thread at a time, even though multiple concurrent threads can be inspected simultaneously in the handshake circuit graph view.

6. Conclusions

A visualisation system oriented towards program comprehension has been presented. It is able to merge and represent in a single view different sources of information related to handshake circuits: the original Balsa source code, the compiled static handshake circuit and the dynamic simulation trace. This results in a very useful graph structure, viewable at any level of detail and showing the evolution of the control flows present in the circuit during the simulation.

The visualisation system is also based on a structure of coordinated views. In order to provide a familiar environment to the designer, the usual *source code*, *wave form* and *post-synthesis Verilog* views are integrated in the environment. Their usefulness is improved by a collaboration scheme allowing the user to track elements from one view to another. The navigation between the various sources of information is considerably enhanced. This enables, in particular, an efficient tracking of the control flows from the simulation trace to the source code or to the post-synthesis Verilog structure, leading to an easy and precise comprehension of the handshake circuit's structure.

The current system is a result of regular evaluations and feedback from the original designers of the SPA processor and from the new team optimising it.

7. References

[1] Edwards, D.A., Bardsley, A., “Balsa: An Asynchronous Hardware Synthesis Language”, The Computer Journal, 2002, Vol. 45(1), pp. 12-18.

[2] Everitt, B., Cluster Analysis, First edition, Heinemann Educational Books Ltd, 1974, Fourth edition, ISBN 0340761199, 2001.

[3] Favre, J.-M., Cervantes, H., “Visualization of Component-based Software”, Laboratoire LSR-IMAG, University of Grenoble, France, 2002.

[4] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[5] GTKWave Electronic Waveform Viewer.
URL: http://www.cs.man.ac.uk/apt/tools/gtkwave/

[6] Herman, I., Melançon, G., Marshall, M.S., “Graph Visualization and Navigation in Information Visualization: A Survey”, IEEE Transactions on Visualization and Computer Graphics, 2000, Vol. 6(1), pp. 24-43.

[7] Michaud, J., Storey, M.-A., Müller, H., “Integrating Information Sources for Visualizing Java Programs”, Proc. International Conference on Software Maintenance, IEEE, 2001, pp. 250-259.

[8] Mirkin, B., Mathematical Classification and Clustering, Kluwer Academic Publishers, 1996.

[9] Mukherjea, S., Foley, J.D., Hudson, S., “Visualizing Complex Hypermedia Networks through Multiple Hierarchical Views”, Human Factors in Computing Systems, CHI’95 Conference Proceedings, ACM Press, 1995, pp. 331-337.

[10] Plana, L.A., Riocreux, P.A., Bainbridge, W.J., Bardsley, A., Garside, J.D., Temple, S., “SPA - A Synthesisable Amulet Core for Smartcard Applications”, Proceedings of Async'2002, Manchester, April 2002, pp. 201-210.

[11] Risch, J.S., Rex, D.B., Dowson, S.T., Walters, T.B., May, R.A., Moon, B.D., “The STARLIGHT Information Visualization System. Proceedings of the IEEE Conference on Information Visualization”, IEEE CS Press, 1997, pp. 42-49.

[12] Shimomura, T., Isoda, S., “VIPS: A Visual Debugger for List Structures”, Proc. Computer Software and Applications Software, 1990, pp. 530-537.

[13] Storey, M.-A., “A Cognitive Framework For Describing and Evaluating Software Exploration Tools”, Ph.D. Thesis, Computing Science, Simon Fraser University, Canada, 1998.

[14] Tuchman, A., Jablonowski, D., Cybenko, G., “Run-Time Visualization of Program Data”, Proc. IEEE Conference on Visualization, October 1991, pp. 255-261.

[15] Zeller, A., Lutkehaus, D., “DDD - A Free Graphical Front-End for UNIX Debuggers”, SIGPLAN Notices, 1996, Vol. 31(1), pp. 22-27.