Information and Communication Technologies (ICT) Programme

Project No: FP7-ICT-287804

D3.2: Verification procedure of static and dynamic aspects of reconfigurable designs

Author(s): Tim Todman, Peter Boehm, Tobias Becker, Wayne Luk (IMP)

Status -Version: Version 1.0 (Final)

Date: August 31, 2012

Distribution - Confidentiality: Public

Code: FASTER_D3_2_IMP_FF_20120831

Abstract:

This document constitutes deliverable D3.2, the outcome of task T3.4, reporting the procedure for the verification of static and dynamic aspects of reconfigurable designs. The procedure includes techniques developed by Imperial for verifying static and dynamic aspects of a reconfigurable design at compile time. In particular, Imperial has studied how symbolic verification, which has been shown to be a powerful verification approach for static designs, can be extended to support both static and dynamic aspects of a reconfigurable design.

Disclaimer

The FASTER Consortium is the following:

Beneficiary Number	**Beneficiary name**	**Beneficiary short name**	**Country**
1(coordinator)	Foundation for Research and Technology – Hellas	FOR	Greece
2	Chalmers University of Technology	CHT	Sweden
3	Imperial College London	IMP	UK
4	Politecnico di Milano	PDM	Italy
5	Ghent University	GNT	Belgium
6	Maxeler	MAX	U.K.
7	ST	STM	Italy
8	Synelixis	SYN	Greece

Document Revision History

Date	**Issue**	**Author/Editor/ Contributor**	**Summary of main changes**
August 27, 2012	0.9	Tim Todman	Final draft
August 31, 2012	1.0 (Final)	Tim Todman	Final

Table of contents

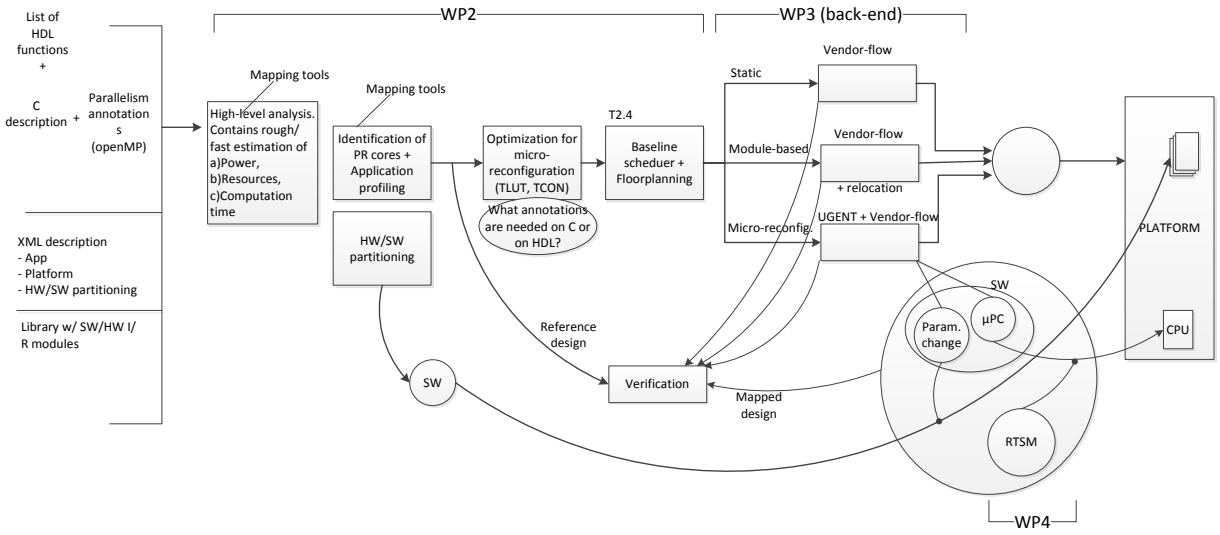

Figure 1: How verification fits into the FASTER tool flow.

1. Introduction

This document constitutes deliverable D3.2, the outcome of task T3.4, reporting the procedure for the verification of static and dynamic aspects of reconfigurable designs. The procedure includes techniques developed by IMP for verifying static and dynamic aspects of a reconfigurable design at compile time.

1.1. Task objectives

This deliverable meets the first objective of task 3.4: techniques for verifying static and dynamic aspects of a design at compile time. The document shows how we use symbolic simulation to verify the static aspects of a design, then extend the approach to verify dynamic aspects of a design. This approach will be extended in future to meet later objectives in this task, namely verifying static and dynamic aspects of a design at run time.

Figure 1 shows how the verification procedure fits with the rest of the FASTER tool flow. The verification flow checks the equivalence of a simple, unoptimized or high-level reference design with a more optimized or low-level design. The reference design could be the output of the mapping process, or even earlier in the tool flow. The optimized design coud be a scheduled, floorplanned or otherwise transformed design to improve given aspects of performance, such as speed, area or energy efficiency.

In particular, IMP has studied how symbolic verification, which has been shown to be a powerful verification approach for static designs, can be extended to support both static and dynamic aspects of a reconfigurable design.

This deliverable concerns verification at compile time, however, the techniques and methods

described can extend to run time. IMP intends to extend and modify this work later in the project to address run time verification.

1.2. Document overview

The rest of this document is organised as follows: section 2 gives a technical overview of the verification flow, showing the approach used, and giving definitions of static and dynamic aspects. Section 3 shows the inputs and outputs of the verification flow. Section 4 shows the procedure for verifying static aspects at compile time, with an example illustrating the approach, while section 5 shows the procedure for verifying dynamic aspects at compile time. Finally, section 6 concludes and shows the planned future work on this task.

2. Technical Overview

The procedure verifies the equivalence of two designs, where equivalence means that they have the same output for each corresponding input. The two designs could be a simple or straightforward implementation of a specification, and an optimized, possibly reconfiguring version. Our procedure works by using symbolic simulation plus equivalence checking, using a number of tools developed by IMP alongside some external tools.

Whilst the implementation uses Maxeler design input, the approach is general and can extend to other hardware design inputs, such as more traditional Verilog and VHDL languages used for hardware description. Future work in this project may address verification of these other design inputs.

Rather than relying on numerical or logical simulation, our approach combines *symbolic simulation* with *equivalence checking* [1], Symbolic simulation applies symbols rather than numbers or logic values to the design, the outputs being functions of these symbolic inputs. For example, symbolically simulating an adder with inputs a and b would result in $a + b$. However, for larger designs it is harder to distinguish different but equivalent outputs ($b + a$ instead of $a + b$) from incorrect ones. The equivalence checker tests whether or not the outputs of transformed designs are equivalent to those of the reference design.

Our approach has the advantages that:

- We simulate and validate word-level designs, assuming that arithmetic operations have already been validated by other techniques, with the advantage that any mismatches between source and target are found at word level, rather than at bit level, which can be hard to read for application designers;
- Symbolic simulation means users need not worry about covering full input and output ranges, as with numerical simulation;
- The equivalence checker automatically compares symbolic input and output to ensure the optimized design preserves the behaviour. Unlike numerical simulation, it can reject false negatives due to different but equivalent output that would differ using simple-minded numerical comparison;

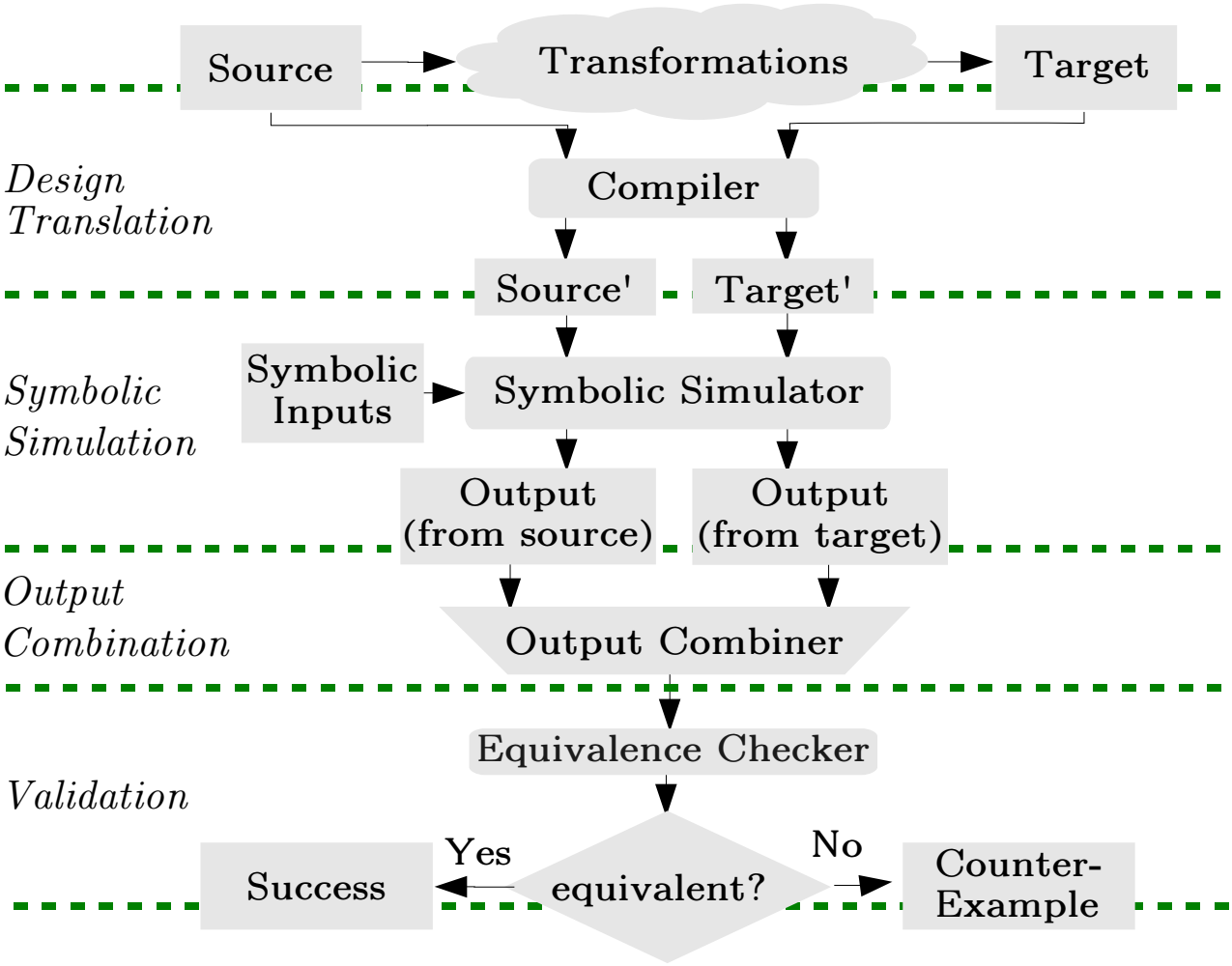

Figure 2: Abstract verification design flow.

- The user can still use numerical simulation to (a) compare with other simulators, such as those from FPGA vendor tools, those from higher-level tools such as Maxeler's MaxCompiler simulator, or results of software implementing the same design, (b) simulate only part of a design symbolically to eliminate data-dependent values, allowing application to larger problems, or to isolate problematic corner cases.

The general idea to verify the compliance of an implementation with a specification using symbolic simulation and checking for equivalence of the symbolic outputs of the simulation, is not new; the application of this methodology to reconfigurable hardware/software codesigns however is.

2.1. Overall verification flow

The core of our approach is to use a straightforward software design as a golden reference and verify its equivalence with optimized designs containing software and hardware.

Figure 2 shows our general verification flow. We refer to the reference model as *source model* and to the optimised model, which is being checked for equivalence, as *target model*. Note that the assumption of a straightforward software implementation as source model is merely a logical one as

this simplifies the correct specification of codesigns. The general verification process is not limited to this setup, but either design could be software, hardware, or a combination of the two.

Given source and target designs, our verification approach works in four modular phases:

1. *Design Translation*: source and target designs are translated to an intermediate representation suitable for successive processing. This ensures that the application/input-specific part of the process is kept as small as possible.

2. *Symbolic Simulation*: both designs are executed symbolically using the same inputs using a symbolic simulator for our intermediate language. We do support basic interface mappings in case the interfaces of source and target design do not match anymore.

3. *Output Combination*: to support weaker notions of equivalence than strict bisimulation-equivalence, such as stutter-equivalence among others, we pre-process the two output traces of the symbolic simulator in an output combiner before passing to the equivalence checker.

4. *Validation*: an equivalence checker compares the (processed) symbolic outputs from source and target designs. The result is either a successful verification, or a counterexample with symbolic inputs leading to different outputs.

Partitioning the workflow in these four steps, especially keeping the design translation small and straightforward, increases the modularity and generality of the approach. As detailed in the next section, we target C software and Maxeler hardware kernels, but the system can easily be extended to support other input languages, such as Verilog, as hardware design inputs. The difference between Maxeler hardware and more traditional hardware design languages such as Verilog is that Maxeler designs are written using Maxeler's class library and extensions to the Java programming language to allow designers to express hardware data paths and control paths at a high level, rather than by using the lower-level features in languages like Verilog. For example, state machines can be expressed directly in Maxeler's domain-specific language, rather than build from registers and transfer functions as in Verilog.

In order to handle realistic systems, we made a design decision of modelling at word level rather than bit level. This provides a very convenient abstraction and helps controlling the infamous state-explosion problem when dealing with hardware. However, to model using a word level abstraction, basic operator implementations are assumed to be correct. While this seems a strong limitation of the system, we can still integrate bit-level correctness results for components in our system using a modular verification approach since operator implementations are purely combinatorial.

Because of our design choices, our approach has some restrictions, though these can be mitigated: 1. we assume synchronous hardware with a single global clock; 2. some data must be numeric, not symbolic, for example array sizes; 3. despite using symbolic inputs, data-dependent control flow can still cause state-space explosion. Using numeric values for data that cannot be abstracted to symbolic values rarely poses a problem in actual case studies: for example there is usually only a limited set of useful image sizes which can be verified using concrete instantiations. Data-dependent control flows can be partitioned by the user to verify different modes separately.

As we are modelling scheduled software and hardware, outputs arrive at specific cycles and reordering or offsets will appear to be different output. If the target design produces results in a different order to the source design, the user must reorder the target design's results in software.

Interaction between software and hardware is modelled as a simple application processing interface (API) in software that the allows the software to send data to and from the hardware, and to reconfigure the hardware. Section 5 shows more details of the interaction between software and hardware.

Finally, our approach can symbolically compare hardware and software implementations of same algorithm—often the translation between the two is a source of bugs.

2.2. Definition of static and dynamic aspects

The static and dynamic aspects to be verified are defined as follows:

- Static aspects are those that can be checked purely at compile-time;
- Dynamic aspects must be checked at run-time

In this deliverable, the static aspects we check are the static equivalence of two static, non-reconfiguring designs: that they have the same output for each corresponding input. The dynamic aspects we check are the equivalence between a static design and a run-time reconfiguring design; section 5 shows our approach to verifying reconfigurable designs.

3. Verification flow: inputs and outputs

This section details the inputs to, and outputs from the verification flow.

3.1. Design inputs

Design inputs to the verification flow consist of:

- C programs, for the software part
- Maxeler MaxJ kernel designs, for the hardware part

To make the verification flow more feasible, we restrict the form of C and MaxJ inputs, and add extensions to allow the designs to be stimulated by symbolic inputs.

3.2. Extended C language

Our approach allows software parts of designs to be described in a C-like language with some restrictions: there is no standard library and no dynamic memory allocation; these restrictions still allow useful software from some application domains, such as embedded software and digital signal processing, to be verified. We could extend our approach to allow dynamic memory allocation,

but with numeric allocation sizes only. We extend C with one statement and one expression to respectively allow symbolic inputs to and outputs from the design.

We specify the syntax of the verification inputs using Extended Backus-Naur Form (EBNF). Our variant of EBNF has the following conventions:

- lower-case identifiers denote non-terminals;
- upper-case identifiers denote text literals such as strings and integers;
- quoted elements denote keywords and punctuation;
- round brackets group elements together;
- square brackets denote optional items
- $X*$ means zero or more occurrences of X, where X is a terminal, non-terminal, or bracketed expression;
- curly brackets also denote zero or more occurrences of the elements they enclose.

The following shows the syntax of our software input in EBNF:

```
gd = d
   | t ID '(' d (',' d)* ')'
   | t ID '(' d (, d)* ')' s
s = 'if' '(' e ')' s ['else' s]
  | 'for' '(' e ';' e ';'e ')' s
  | 'while' '(' e ')' s
  | e ';'
  | '{' d* s* '}'
  | 'write' '(' STRING ',' expr ')'
d = t ID {',' ID}
e = e bop e
  | uop e
  | INT | FLOAT | STRING
  | e '.' e
  | e '(' [e (',' e)*] ')'
  | e '[' e ']'
  | 'read' '(' STRING ')'
bop = '*'|'/'|'+'|'-'|'%'|'=='|'!='|'<'|'<='
    |'>'|'>='|'<<'|'>>'|'&'|'^'|'&&'|'|'|'+='
|'-='|'*='|'/='|'<<='|'>>='|'&='|'|='|'^='
uop = '*'|'!'|'~'|'-'
t = 'int' | 'float' | 'bool'
```

The grammar uses standard C operator precedence rules to avoid ambiguity. We extend C with the `write` statement and `read` expression, which respectively allow symbolic outputs from and symbolic inputs to the design. For our prototype, this offers a simple way to stimulate the design with symbolic inputs; a more practical implementation could overload existing C input/output (IO) facilities, such as files, with symbolic connections.

Our approach compiles from software to symbolic simulation by adapting a syntax-directed scheme for compiling to hardware [2], which works by translating expressions to combinatorial hardware and statements to a one-hot encoded state machine. Whilst this means the simulation is effectively scheduled, with one assignment statement per simulation cycle, this also makes it easy for the designer to calculate when the outputs corresponding to a symbolic input will emerge.

3.3. Extended MaxJ language

The hardware design description is based on a simplified version of the Maxeler MaxJ streaming hardware description language. The Maxeler system allows the user to describe streaming hardware designs as Java programs written using their Java class library and Java language extensions. The user writes a program which, when run, builds the dataflow graph of the streaming design, compiles the graph into a HDL (Hardware Description Language) implementation, and calls FPGA vendor tools to compile the HDL into a bitstream.

The streaming hardware design consists of a data path reading from one or more stream inputs, one per cycle, and producing one or more stream outputs, also one per cycle. Scalar inputs are set once per run of the stream and are constant for each run. A control path controls the operation of the design.

Our hardware input language corresponds to the data path of a Maxeler description, plus the control path using counters. There are some restrictions: the control path is not symbolic, and models numeric counters only. This means the approach can cover all numeric states but that if the bounds of the counters are set at compile time, the design must be verified for each parameter value. The advantage is that symbolic output is faster, and the symbolic output is easier for the user to read – symbolic simulation of the control path leads to large symbolic expressions that cover all the control decisions made up to that point. The disadvantage is that each control path must be checked separately, and the design cannot be so easily parametrised – for example, for a 1D convolution, each size of window must be verified separately; in practice, this is not such a limitation, since only a few sizes of window are useful. We do not currently model Maxeler's state machines; these could be modelled like the counters, so that the state is numeric, rather than symbolic.

Our approach is not restricted to Maxeler or streaming hardware, and could model other streaming design inputs, or more general HDLs such as VHDL or Verilog.

The following shows the syntax of our hardware design input in EBNF:

```
start = 'stream' '{' (s | d)* '}'
s = 'if' '(' e ')' s ['else' s]
  | 'for' '(' e ';' e ';' e ')' s
  | 'while' '(' e ')' s
  | e';'
  | e '<==' e';'
  | '{' d* s* '}'
d = t ID (',' ID)*
e = e bop e
  | uop e
  | INT | FLOAT | STRING
  | e '.' e
```

```
  | e '(' [e(',' e)*] ')'
  | e '[' e ']'
bop = '*'|'/'|'+'|'-'|'%'|'=='|'!='|'<'|'<='
 |'>'|'>='|'<<'|'>>'|'&'|'|'|'^'|'&&'|'||='
 |'+='|'-='|'*='|'/='|'<<='|'>>='|'&='|'|='|'^='
uop = '*'|'!'|'~'|'-'
t = 'int' | 'float' | 'boolean' | 'HWVar'
```

To compile from hardware to symbolic simulation, we use the following scheme:

- The design is first interpreted to evaluate any metaprogramming features (conditional compilation, unrolling) used in the design;
- The interpreted design is compiled to a dataflow graph (DFG);
- The DFG is compiled into input for the symbolic simulator – generally this is a one-to-one mapping, except for counters, which must be built from simulator primitives.

3.4. Symbolic inputs

Symbolic inputs to the designs to be verified consist of textual symbols. Each symbol represents any possible value that can be applied to the inputs, occur internally within the design, or appear at the outputs.

3.5. Verification outputs

The verification process yields a message showing whether or not the source and target designs could be verified as equivalent for all symbolic inputs. There are three possible outcomes:

- Success: if the output is sat on all cycles, the corresponding symbolic outputs match each other for all symbolic inputs, and the two designs are equivalent;
- Unknown: if the designs are too large to verify, the result can be unknown for one or more cycles;
- Failure: if one or more corresponding outputs definitely do not match, then verification has failed, and the designs are not equivalent.

In the unknown case, the designs are too large to verify. The user can decide to partition their design into smaller components, and verify the equivalence of those pieces. Each individual piece must be recompiled for the symbolic simulator, but in practice this takes a few seconds per piece.

In the failure case, the verification flow reports the differing symbolic outputs, and the cycle they occurred on, which the user can use for debugging the cause of the mismatch.

4. Verifying static aspects at compile time

This section shows our procedure for verifying static aspects of a design at compile time, by verifying the equivalence of two non-reconfiguring designs with each other.

4.1. Procedure for static aspects

The procedure for static aspects runs as follows:

- The user chooses source and target designs, for example the source designs could be an unoptimized design, and the target could be an optimized version;
- The user chooses the number of symbolic inputs to apply, using the number of counter states in the Maxeler design – since we only verify designs with known, numeric bounds on their counters, this can be calculated easily from the design;
- Our compiler compiles both designs into separate inputs to the symbolic simulator;
- The verification toolflow runs the symbolic simulator on both designs with the symbolic inputs;
- The user parameterizes the output combiner to suit both designs – parameters include startup latency and cycles per output
- The output combiner compiles both symbolic outputs from the symbolic simulator into a single input to the equivalence checker;
- The equivalence checker checks the equivalence of both designs for every cycle, with possible results of success (all corresponding outputs march for each input), unknown (the design is too large to check), or failure (one or more corresponding outputs definitely do not match).

This toolflow uses the following tools:

- Our own compiler from Maxeler MaxJ [3] design descriptions to the Rebecca symbolic simulator format;
- The Rebecca symbolic simulator [4];
- Our own output combiner, compiling symbolic output from the Rebecca simulator to input to the equivalence checker;
- The Yices satisfiability modulo theories solver [5], used here as an equivalence checker;
- Our own tools to script these components together.

Figure 3 shows the inputs and outputs to the process, and how the tools work together.

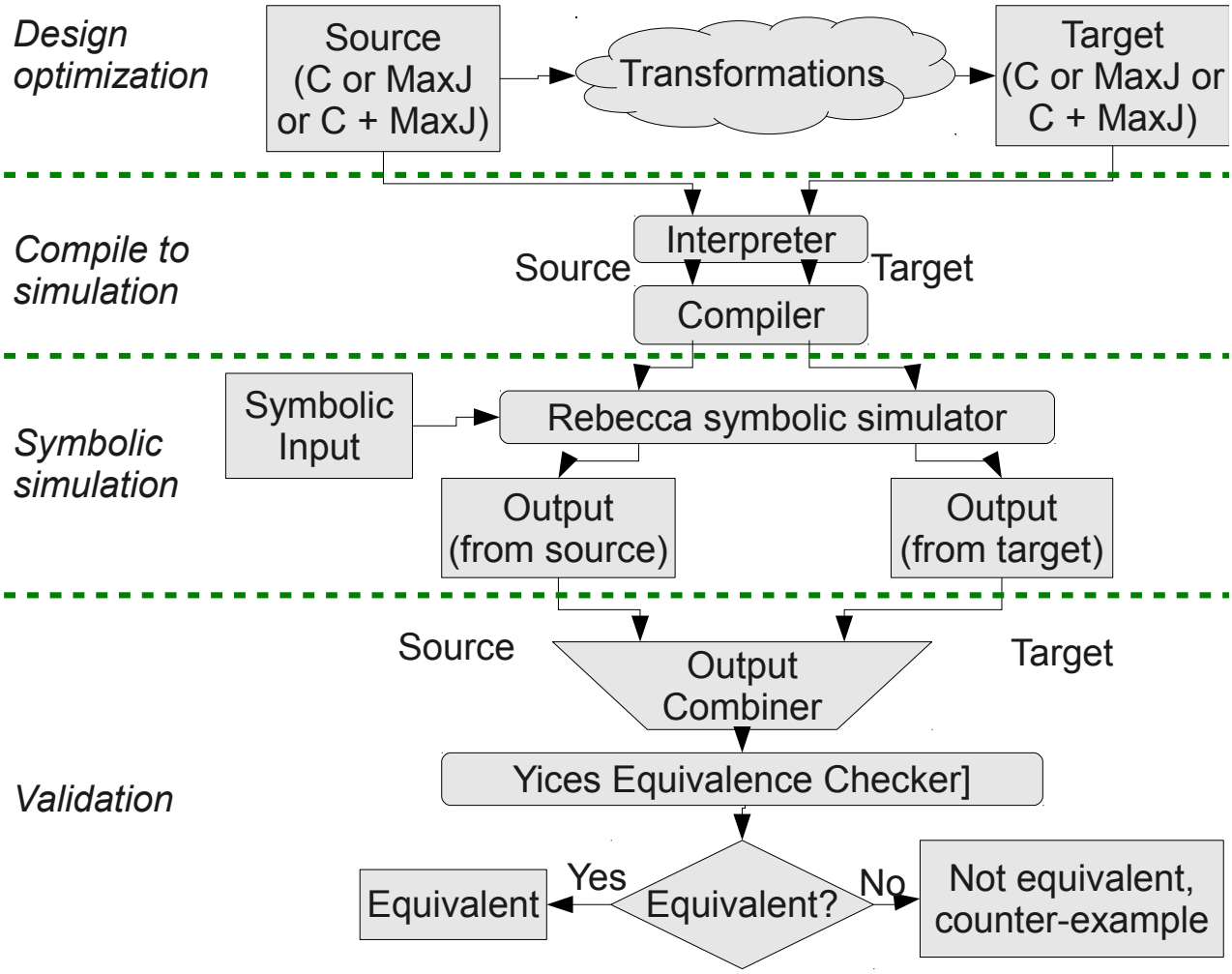

Figure 3: Concrete verification design flow, showing how the tools interact with each other.

4.2. Worked example

The procedure is illustrated by an example comparing unrolled and rolling-sum versions of a one-dimensional moving average kernel. The following is a basic MaxCompiler kernel implementation (for clarity, we only show the kernel core without surrounding declarations; line numbers are not part of the code):

```
1:HWVar x = io.input("x", scalarType);
2:HWVar sum = constant.var(scalarType, 0);
3:for (int i = 0; i < W; i++)
4:  sum += stream.offset(x, -i);
5:io.output("z", sum / W, scalarType);
```

Respectively: line 1 declares a stream input named x; the type scalarType is parameterizable, in our example it is single-precision floating point, lines 3-4 implement the summation, summing the current stream input with previous ones using stream offsets and line 5 outputs the sum divided by the window size W to an output stream named z.

Note that the loop in lines 3-4 runs at compile time, effectively unrolling the summation and resulting in W adders in total. We compare this design with one implemented as a rolling sum, which saves $W - 2$ additions by saving partial sum results between outputs:

```
1:HWVar inp1=io.input("x",scalarType);
2:HWVar temp=scalarType.newInstance(this);
3:HWVar sum=temp+inp1;
4:temp <== stream.offset(sum,-1);
5:HWVar temp2=sum-stream.offset(sum,-(W+1));
6:io.output("z",temp2 / W,scalarType);
```

where respectively: line 1 is as before, line 2 creates an unconnected graph node which will be used for feedback, line 3 adds the current input to the rolling sum, line 4 connects the sum result back to the unconnected node, line 5 subtracts the input from $W + 1$ cycles previously, line 6 is as before.

Our compiler translates this into the following simulator input, for window width $W = 3$:

```
1: D 0 .2 .3
2: D 0 .1 .2
3: add <.3,.2>   .4
4: add <.4,.1>   .5
5: div <.5,3>   .7
6:Directions - in ~ out
7:Wiring - .1 ~ .7
```

where nodes are numbered, so `.1` is node 1, and each line comprises the name of a simulator primitive and its input and output nodes, for example `D 0 .2 .3` is a register with input node 2, output node 3. Respectively: lines 1-2 implement a register chain to implement the stream offsets, lines 3-4 implement the unrolled adder tree, line 5 divides by constant 3, and lines 6-7 give the external connections, input `x` and output `z` compiled into nodes 1 and 7 respectively.

To simulate symbolically, we generate symbols to apply to the inputs. We let the user decide how many cycles to simulate; for Maxeler designs controlled by counters, the design should be simulated for the total number of counter states. Simulation results are as follows:

```
0 - a_0 ~ ((0 + a_0) / 3)
1 - a_1 ~ (((0 + a_0) + a_1) / 3)
2 - a_2 ~ (((a_0 + a_1) + a_2) / 3)
```

This shows three cycles of symbolic output for symbolic input stream `a_0`, `a_1`, `a_2`.

Finally, our output combining tool compiles the source and target symbolic simulation outputs into input for the equivalence checker. For our simple example, this looks like (window width $W = 3$):

```
1: (echo "1: ")
2: (assert (forall (a_0::int)
   (=(/(+ 0 a_0)3) (/(-(+ 0 a_0)0)3))))
3: (check)
```

where each simulation clock cycle gives rise to three statements: an echo statement, which prints the clock cycle, an assert statement, which declares symbolic variables used in the statement and compares the output expressions, and a check statement, which checks the previous statement for equivalence. Note that the language uses Lisp-style s-expressions, so each expression is enclosed by brackets and contains the operator and its operands in sequence, thus $a_0 + a_1$ becomes $(+a_0 a_1)$.

5. Verifying dynamic aspects at compile time

Our approach to modelling dynamic aspects of designs at compile time is now described.

5.1. Modelling dynamic aspects

The dynamic aspect we model is reconfiguration: the design can reconfigure itself to perform a new function.

Our model for software-hardware codesign is a software host with an application programming interface (API) for loading and running streaming hardware designs. Our model and the API calls are *synchronous*, meaning that the software halts until the hardware returns. While this limits parallelism, this can verify simple hardware-software partitioning which may help to eliminate some bugs in a more general, asynchronous design (here asynchronous means the hardware and software run in parallel). Future work on this task may remove this limitation. The API contains the following calls:

- `load`: loads a streaming hardware design compiled with our hardware compiler;
- `run`: runs a previously-loaded hardware design for a given numeric cycle count, with one or more input or output arrays, which must match stream inputs and outputs on the hardware design;
- `set_scalar`: sets a scalar hardware input to a given value, which will apply to the hardware design on the next call to `run`.

5.2. Runtime Reconfiguration

Our approach is extended to support run-time reconfiguration by adding to our API a call that allows the user to load multiple streaming hardware designs and switch between them by writing to a particular scalar input.

Our approach compiles reconfiguration to symbolic simulation using the concept of *virtual multiplexers* to represent different configurations; the multiplexers switch out inactive parts of the design [6].

Figure 4 shows how virtual multiplexer-demultiplexer pairs model reconfiguration. Each hardware element between the multiplexers represents a mutually exclusive partial configuration, with control inputs switching between the configurations. The non-reconfigurable parts of the design connect

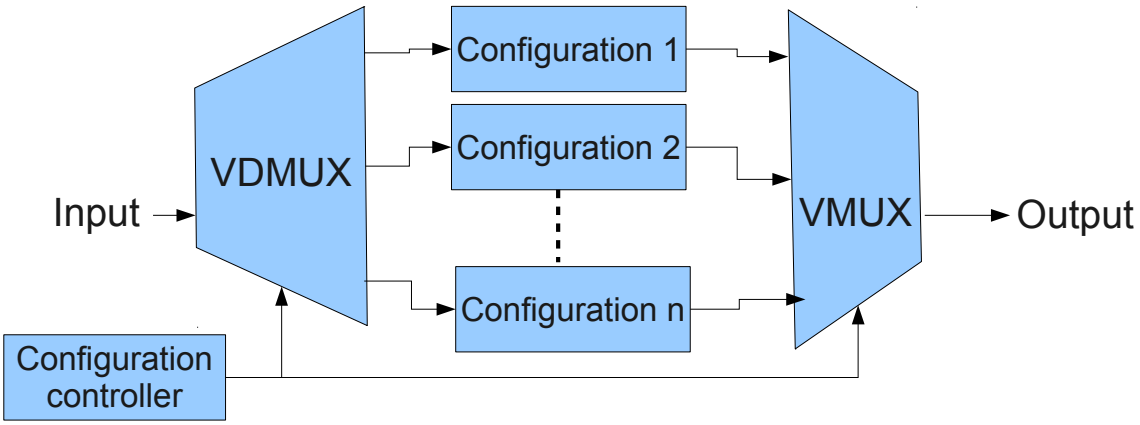

Figure 4: Using virtual multiplexer (VMUX) and demultiplexer (VDMUX) pairs to model run-time reconfigurable designs.

to the multiplexer data inputs and outputs. Figure 5 shows the timing semantics of our model of reconfiguration: the new configuration is loaded as soon as the control input to the virtual multiplexers changes.

5.3. Design inputs: extensions for dynamic aspects

Our extension adds one statement to the grammar of Maxeler's Java dialect:

```
s -> ... | reconfigure_if ( e ) s else s
```

where `s` and `e` are non-terminals representing statements and expressions, respectively, and `reconfigure_if` is a new keyword. The new statement resembles the if-statement, except that it leads to run-time decisions, rather than compile-time decisions in Maxeler kernels. Note that, unlike the conventional C or Java if-statement, the else clause is compulsory. The meaning of:

```
reconfigure_if (e) s1 else s2
```

is: `e` must be a run-time expression of boolean type; when `e` is true, the FPGA is configured to perform `s1`, otherwise it is configured to perform `s2`. To implement this, the tools must (a) compile the expression into a runtime reconfiguration controller, such as a soft processor on a Xilinx device, (b) compile `s1` and `s2` into separate bitstreams that can load on demand depending on the value of `e`. Multiple `reconfigure_if` statements can be composed to model multiple alternative configurations.

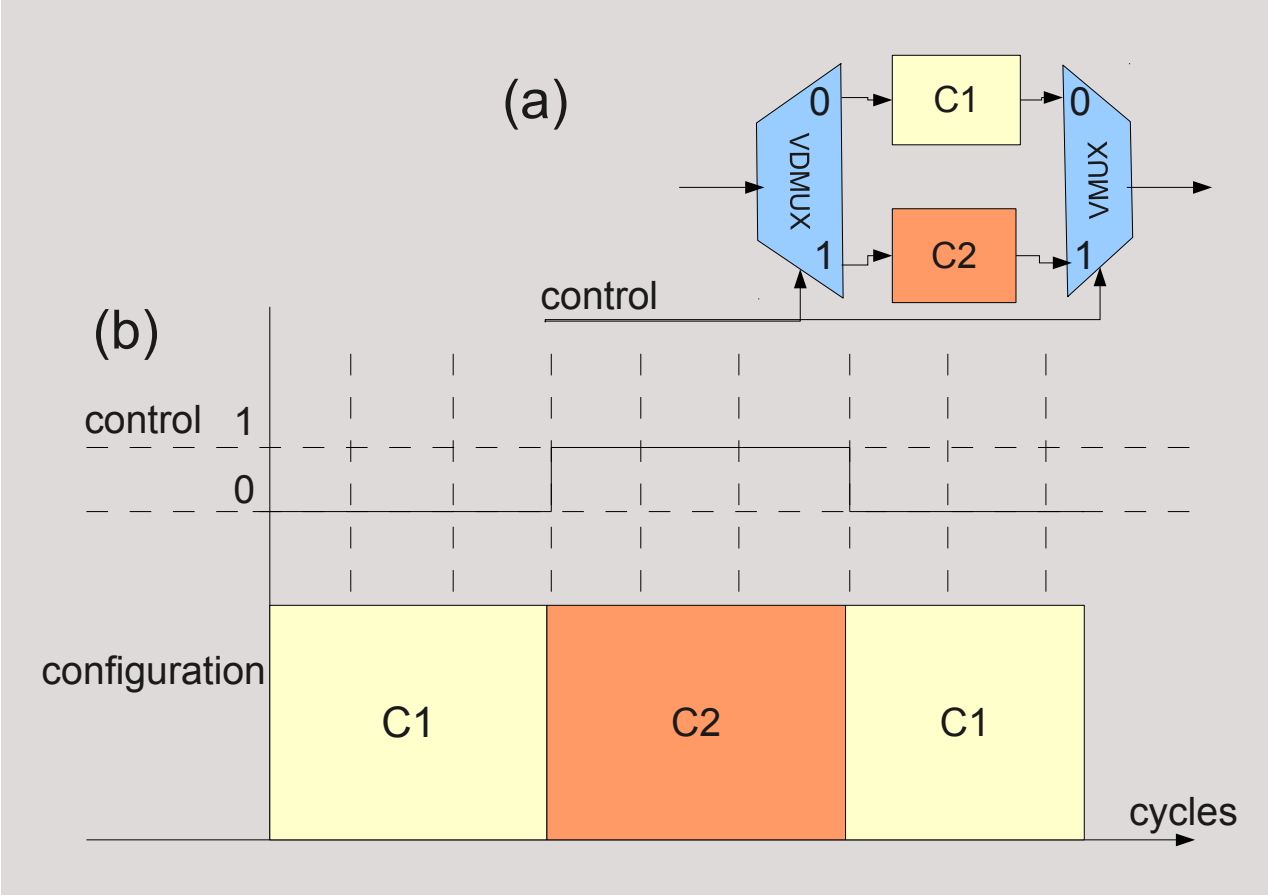

Figure 5: (a) two mutually-exclusive configurations connected by a multiplexer (VMUX)-demultiplixer (VDMUX) pair; (b) Timing of reconfiguration modelled by virtual multiplexers; when the control line of the multiplexer-demultiplexer pair changes, the corresponding design is configured on the same cycle.

5.4. Procedure for dynamic aspects

The procedure for verifying dynamic aspects at compile time checks the equivalence of two, possibly reconfiguring, designs with each other. The procedure is the same as for verifying static aspects at compile time, except that design inputs for reconfigurable designs consist of reconfiguring Maxeler kernels with the reconfiguration controlled by C software using our simple API. The user can verify (a) the equivalence of a static (non-reconfiguring) designs with reconfiguring design, or (b) two reconfiguring designs. The user must ensure that sufficient symbolic inputs are provided to capture all the reconfuring behaviour.

5.5. Worked example

The following code shows a run-time reconfigurable hardware for a 1D convolver, using reconfiguration to switch between constant multiplier coefficients. The following shows an implementation using our extended MaxJ:

```
1:  int coeff_a[W] = {...};
```

```
2:  int coeff_b[W] = {...};
3:  HWVar config =
       io.scalarInput("config",hwUInt(1));
4:  HWVar inp =
       io.input("inp", HWInt(32));
5:  HWVar sum = 0;
6:  for (int j = 0; j < W; j++) {
7:    reconfigure_if(config) {
8:      sum+=coeff_a[j] *
9:         stream.offset(inp, -j);
10:   } else {
11:     sum+=coeff_b[j] *
12:        stream.offset(inp, -j);
13:   }
14: }
15: io.output("outp", sum, HWInt(32));
```

where:

- lines 1 and 2 declare alternative sets of coefficients;
- line 3 declares a boolean scalar input used to switch between configurations;
- lines 4 and 5 are the stream input and local sum as before;
- lines 6-14 are a loop which uses the value of the scalar input to switch between the two configurations – the only difference is that each configuration uses a separate set of coefficients;
- finally, line 15 is the stream output as before.

Figure 6 shows the datapath of the run-time reconfigurable version, where the scalar input is used to switch between sets of constant multipliers. Note that we have factored out the common parts of the two configurations for clarity.

To drive the run-time reconfigurable version, we modify the software to swap between the configurations as follows:

```
1:  int inp[N], outp[N];
2:  for (int i1=0; i1<N/K; i1++) {
3:    for (int i2=0; i2<K;i2++) {
4:      i=i1*K+i2;
5:      in1[i]=inp[i];
6:    }
7:    set_scalar(conf, ...);
8:    run(in1, out1, K);
9:    for (int i2=0; i2<K;i2++) {
10:     i=i1*K+i2;
11:     out[i]=in1[i];
12:   }
13: }
```

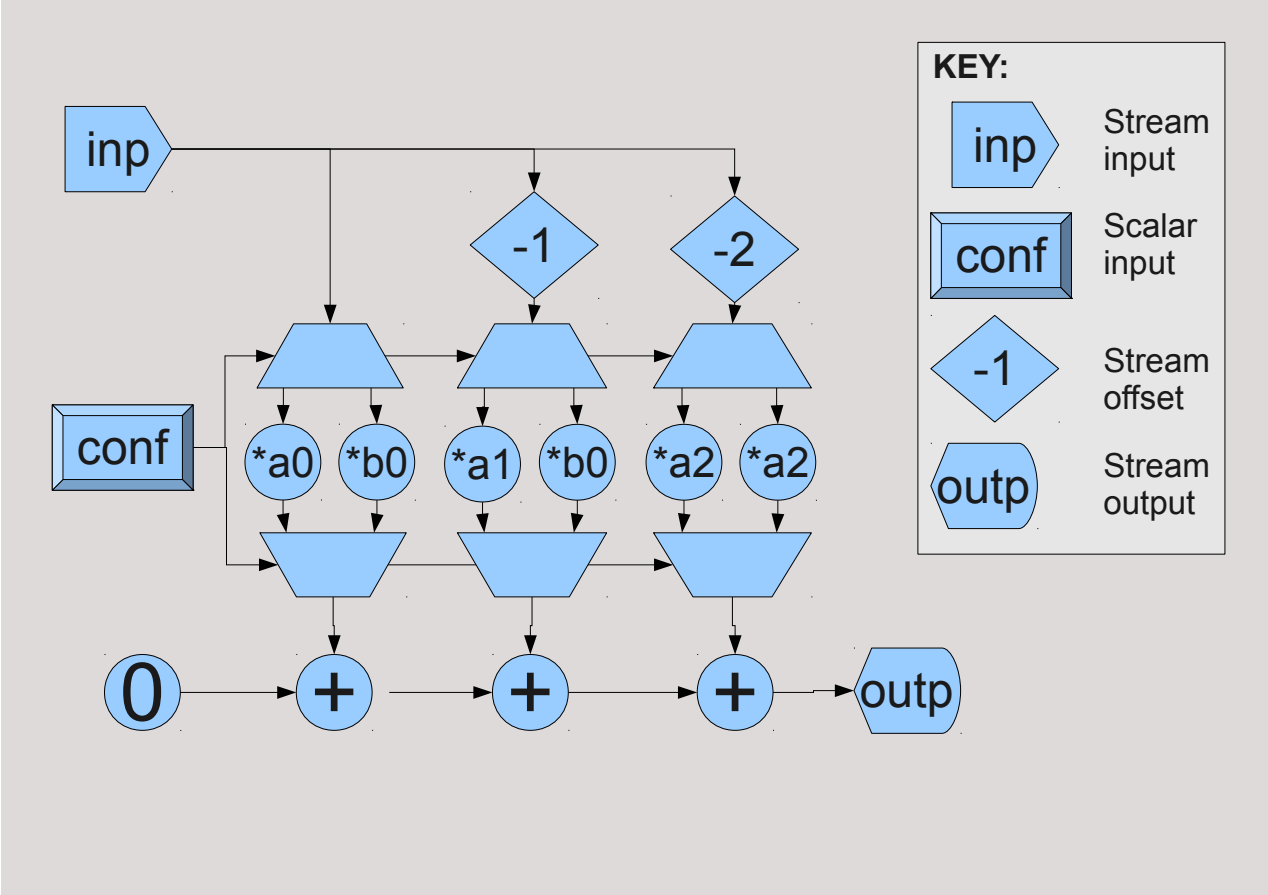

Figure 6: Run-time reconfigurable version of the 1D convolver design. The multiplexers and demultiplixers are virtual; their alternatives represent different partial configurations. For simplicity, we have drawn the multiplexers as a chain, meaning that they share the same control input. Writing to the scalar input `conf` causes the design to reconfigure from one coefficient set to the other. We use Maxeler's conventions for dataflow diagrams.

where:

- lines 2-6 load the data into memory to send to the hardware;
- line 7 sets the scalar which controls the reconfiguration – we leave this blank for this example, but this will be an application-dependent condition;
- line 8 runs the chosen hardware configuration on the input data, producing the output data;
- lines 9-13 copy the data back from the hardware.

Each time the user sets the scalar input `conf`, the configuration will change before the next call to `run`.

6. Conclusion

This document illustrates the procedures for verification of static and dynamic aspects of designs at compile time. This meets the first objective for task 3.4, by developing techniques for verifying static and dynamic aspects of a design at compile time. The approach uses symbolic simulation, which has been used in the past to verify static aspects of a design, to verify both static and dynamic aspects at compile time. This is combined with equivalence checking, to test whether differing symbolic outputs are semantically different or not. The static aspects verified are the equivalence of two designs, with a procedure given for this. The dynamic aspects verified are the equivalence of reconfiguring designs with one another, or with a non-reconfiguring design. A procedure is given to verify dynamic aspects at compile time, illustrated by an example.

Future work in this task will extend this approach to meet the remaining objectives of task 3.4. First, this work will be extended to verify static and dynamic aspects at run time, without significantly impacting on speed, area and power consumption. Second, the approach will be extended to help the run time system support low overhead and low power run time verification.

References

[1] P. Hoxey, C. McDonald, and D. Guinther, "An introduction to symbolic simulation," *EE Times*, December 2005.

[2] I. Page and W. Luk, "Compiling occam into FPGAs," in *FPGAs* (W. Moore and W. Luk, eds.), pp. 271–283, Abingdon EE&CS Books, 1991. Describes methodology to compile from high level language to FPGA, a precursor to Handel-C.

[3] "MaxCompiler White Paper," tech. rep., Maxeler Technologies, Feb. 2011.

[4] S. R. Guo and W. Luk, "An integrated system for developing regular array designs," *Journal of Systems Architecture*, vol. 47, pp. 315–337, Apr. 2001.

[5] B. Dutertre and L. de Moura, "The YICES SMT Solver," tech. rep., Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025 - USA, 2006.

[6] P. Lysaght and J. Stockwood, "A simulation tool for dynamically reconfigurable field programmable gate arrays," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 4, pp. 381–390, Sept. 1996.