Abstracting from Register-Transfer to Algorithmic Level for Verification

Jens Schönherr

Signalion GmbH, Am Waldschlösschen 2, 01099 Dresden, Germany

jens.schoenherr@signalion.com

Abstract

To exploit the advantages of the verification at algorithmic level also for RTL designs the designs must be abstracted to this level first. This abstraction is based on the input-output scheme of the RTL design. In this article it is shown how a generalized input-output scheme is mapped to a simple scheme for which the abstraction to algorithmic level is known. Additional generalizations for the input-output scheme are presented.

1 Introduction

1.1 Motivation

A significant part of digital circuits is designed in the area of digital signal processing (DSP). These circuits implement arithmetic algorithms that can be implemented in hardware more efficiently than on a processor. The verification of those circuit designs must show that the specified algorithm has been implemented correctly i.e. that the design produces for each set of parameters the same result as the specified algorithm that is typically described in C/C++ or Matlab.

During simulation-based verification [1][13] parameters are applied in a testbench to the design at register-transfer level (RTL) according to the input scheme. The results are read out from the design according to the output scheme. These values are compared with the results that are produced by the specified algorithm for the same input parameters.

Basically, model checking [12] can be used for the formal verification of digital circuits. However, the internal representation is based on BDDs and the size of BDDs for arithmetic circuits cannot be minimized well [16, p. 97ff.]. This prevents from the application for DSP. In the rest of this article model checking is, hence, not regarded. Bounded Model Checking (BMC), instead, does not rely on BDDs. In BMC, the behavior of the algorithm and the input-output scheme of the RTL design are described in a property language. Prove algorithms such as SAT show that for all parameter values the output of the RTL design and the algorithm are equal. [2][3][5][17]

From the users point of view both approaches have different pros and cons: Simulation must be restricted to a relatively small portion of the possible parameter values. Coverage metrics provide only a hint for the functional coverage [11]. Nevertheless, it is a clear advantage of the simulation that the behavior of the designs can be verified for large time frames (RTL) or number of loop iterations (algorithm). Methods of formal verification do not allow the verification of large time frames because of resource limitations (computation time). In those parts of the
parameter space that could be verified formally the correctness of the implementation at RTL is proven i.e. 100% functional coverage is achieved. In both approaches, the input-output scheme of the RTL design is not described explicitly: In the simulative approach the input-output scheme is implicitly contained in the testbench, in formal verification in the description of the formal properties together with the specified algorithm.

From the user perspective it would be beneficial to combine the advantages of both approaches. For instance, formal verification could be applied for all those parameters that let the algorithm finish within a few loop iterations. For other parameters simulation is used for verification. Assuming the tools that are currently available such a combination would require a setup of two different verifications (testbench and properties) which is not acceptable from the economic view. Therefore, it is proposed in [15] to abstract the RTL design in a first step to the algorithmic level using an explicitly given input-output scheme. This algorithmic description can then be verified against the specified algorithm either using simulative or formal methods. (Figure 1)

It is described in [15] how the abstraction from RTL to algorithmic level can be carried out automatically based on the next-state and output function of the design. During this abstraction step it is checked if the design is compliant to the specified input-output scheme. The formal verification of the abstracted algorithm against the specified one can also be automated because no manually written properties are needed. Even the simulation can be further automated: The description of the implementation at algorithmic level enables the application of methods for functional test generation like [14].

The proposal of [15] assumes a simple input-output scheme (see Section 2.1). In industrial designs often more complex input-output schemes are used. Therefore, methods are needed that can abstract designs with such input-output schemes to the algorithmic level. This article investigates how a complex input-output scheme can be mapped using an input-output mapping to the simple input-output scheme that is assumed in [15] (Figure 2). This input-output mapping together with the design at RTL is then abstracted to algorithmic level using the method described in [15].

It is described in [15] how the abstraction from RTL to algorithmic level can be carried out automatically based on the next-state and output function of the design. During this abstraction step it is checked if the design is compliant to the specified input-output scheme. The formal verification of the abstracted algorithm against the specified one can also be automated because no manually written properties are needed. Even the simulation can be further automated: The description of the implementation at algorithmic level enables the application of methods for functional test generation like [14].

The proposal of [15] assumes a simple input-output scheme (see Section 2.1). In industrial designs often more complex input-output schemes are used. Therefore, methods are needed that can abstract designs with such input-output schemes to the algorithmic level. This article investigates how a complex input-output scheme can be mapped using an input-output mapping to the simple input-output scheme that is assumed in [15] (Figure 2). This input-output mapping together with the design at RTL is then abstracted to algorithmic level using the method described in [15].
The article is structured as follow: Relations between this and other known approaches are presented in Section 1.2. In Section 2 a method for the construction of an input-output mapping is given for a specific input-output scheme. This approach places new requirements on the verification. It is described in Section 3 how these requirements must be taken into consideration during the verification of the algorithms. In Section 4 it will be investigated how far this method of input-output mappings can be generalized to other input-output schemes. Section 5 summarizes this article and gives an outlook.

1.2 Similar Approaches

In [4] is presented how an algorithm described at RTL can be transformed to transaction level (TL) [8]. After the abstraction the sequence of the parameter input and result output as it is implemented in the RTL design is still contained at TL. The approach described in this article abstracts even from the parameter and result sequences. At TL dedicated input-output primitives are needed to enable connectivity to the FIFO-channels. Both facts prevent a direct comparison with the specified algorithm as it is achieved with the approach of this article.

In [7] an approach is described that allows the comparison of computation algorithms that are described in a C/C++-Subset with an implementation at RTL by sequential equivalence checking (SEC). The method synthesizes the algorithm to a description at RTL with a simple input-output scheme. In this article an opposite approach is proposed because a mapping between both input-output schemes in SEC is assumed to be rather difficult. It is mentioned in [7] that the precise input-output scheme of the RTL design must be provided to SEC. However, it is not described which schemes are allowed or how the schemes are described.

In the approach described in [10] the input-output scheme can be chosen very freely. The input-output scheme is described together with the specified algorithm in C. It is described how a formal verification can be carried out based on this description. There is no explicit abstraction from RTL to algorithmic level. The verification is carried out immediately at RTL design and algorithm. This holds also for the approach described in [9] even if the input-output scheme is described in another way in C and the formal verification uses another approach. The internal representation of algorithm and RTL design as FSM prevents an automation of the simulation by generation of stimuli that relies on a representation at algorithmic level [14]. Therefore, a combination of simulation and formal verification is not possible using this approach.

2 Approach

2.1 Simple Input-Output Scheme

The approach proposed in this article uses a complex input-output scheme which generalizes the simple input-output scheme used in [15]. This simple input-output scheme is shown in Figure 3 and is explained here in short.

An active output $rfd_o$ (ready for data) indicates that the module is ready for the next computation (1). The activation of $en_i$ (enable input) for one clock cycle starts the computation assuming that the parameter values are available at the input $params_i$ at the same time (2). The module indicates by activating $en_o$ (enable output) for one clock cycle that the computation is finished (4). In the same clock cycle the results are valid at the output $results_o$. It is assumed that $en_i$ stays inactive as long as $rfd_o$ is inactive (3).
In [15] it is described how designs with this input-output scheme are abstracted to algorithms. The input parameters of the algorithm are the values of params_i at (2) and the result values are results_o at (4). The control signals en_i, en_o, rfd_o, and clk_i are not part of the algorithm. The abstracted algorithm contains one loop. The computation of one clock cycle of the module is mapped to the computation of one loop iteration. This implies that the number of clock cycles between (2) and (4) is equal to the number of the loop iterations of the abstracted algorithm. The function of the computation step, the end condition of the loop, and the initialization of the variables at the start of the algorithm are derived from the next-state and output function of the design. Following this method, a design that implements for instance the computation of the greatest common divider (GCD) is abstracted to an algorithm that computes the GCD in a loop.

2.2 Generalized Input-Output Scheme

In many circuit designs it is not possible to apply all parameters at the inputs in the same clock cycle. For a large amount of data this would require a unacceptable large number of wires. Therefore, in such cases the parameters are handed over to the design sequentially. Between two consecutive parameters there may be some clock cycles without parameters.

Figure 4: Generalized input-output scheme with n_params = 3, gaps(0) = 1, gaps(1) = 2 and n_results = 2. p0...p2 are abbreviations for params(0)...params(2), r0 and r1 are abbreviations for results(0) and results(1)

Figure 4 shows such a complex input-output scheme. The n_params parameters params(i) are applied to the design. The number of clock cycles between the parameters i and (i + 1) are named as gaps(i). The n_results results results(i) are provided at the output sequentially.
where the number of clock cycles between the output of the single result values is arbitrary. The design is ready for new inputs \((\text{rfd}_o = 1)\) if the last result is available at the output.

### 2.3 Construction of the Input-Output Mapping

The input-output mapping maps the inputs and outputs of the simple input-output scheme to the inputs and outputs of the design under verification (DUV). The composition of design and this input-output mapping will be called simplified design in the following where the attribute simplified refers to the input-output scheme of this design. Figure 5 illustrates this naming. To ensure the abstraction to an algorithm from the simplified design following [15] the input-output mapping must be described at RTL.

- **Initialization**
  - \(\text{eni}_s = 0\),
  - \(\text{rfd}_o = 1\),
  - \(\text{en}_o = 0\),
  - \(\text{res}_\text{cnt} = n_{\text{results}}\),
  - \(\text{error} = 0\)

- If \(\text{rfd}_o = 1\) and \(\text{en}_i = 1\)
  - \(\text{params}_i(0)\) is assigned to \(\text{param}_s\) and for one clock cycle \(\text{eni}_s\) is set to 1,
- params_i and gaps_i are stored in the registers params_r and gaps_r, resp.,
- the counter param_cnt is set to 1,
- the counter gap_cnt is initialized by gaps_i(0),
- the counter res_cnt is set to 0,
- rfd_o is set to 0.

• If rfd_o = 0, param_cnt < n_params, en_i = 0, and gap_cnt > 0
  - gap_cnt is decremented,
  - if rfd_s = 1 then error is set to 1.

• If rfd_o = 0, param_cnt < n_params, en_i = 0, and gap_cnt = 0
  - params_r(param_cnt) is assigned to param_s and for one clock cycle eni_s is set to 1,
  - if param_cnt < (n_params - 1) then gap_cnt is initialized with gaps_r(param_cnt),
  - param_cnt is incremented,
  - if rfd_s = 1 then error is set to 1.

• If rfd_o = 0, param_cnt = n_params, en_i = 0, eno_s = 1, and rfd_s = 0
  - result_s is assigned to results_r(res_cnt),
  - res_cnt is incremented,

• If rfd_o = 0, param_cnt = n_params, en_i = 0, eno_s = 0
  - if rfd_s = 1 then error is set to 1.

• If rfd_o = 0, param_cnt = n_params, en_i = 0, eno_s = 1, and rfd_s = 1
  - result_s is stored in results_r(res_cnt),
  - res_cnt is incremented,
  - if res_cnt < n_results then error = 1,
  - results_o gets the value of results_r and for one clock cycle en_o is set to 1 if error = 0,
  - rfd_o is set to 1,

The simplified design must allow checking the compliance of the DUV to the generalized input-output scheme. In this article, it is proposed to split this check into two parts: some input-output errors are propagated to the result output such that the comparison of the abstracted algorithm with the specified one will uncover them. There is an internal signal error that forces a deviation at the result output during verification. The signal error remains inactive as long as rfd_s is activated only after the output of n_results result values.

The remaining input-output errors are checked by the additional condition that rfd_o stays active, as long as en_i = 0, i.e.

\[
en_{i_0} = 0 \land \lambda_{rfd_o}(in_0, \text{sig}) = 1 \rightarrow \lambda_{rfd_o}(in_1, \delta(in_0, \text{sig})) = 1
\]

(1) is fulfilled, where \(\delta\) and \(\lambda\) stand for the state transition and output function of the DUV and \(in_0\) and \(in_1\) stand for two consecutive inputs. This condition can be checked automatically using a SAT solver.
3 Verification

3.1 General Method

Once the input-output mapping is constructed for a design the simplified design is abstracted to an algorithm as described in [15] (see Section 2.1). This abstracted algorithm gets as parameters \( params \) as well as \( gaps \). The specified algorithm, instead, gets only \( params \) as parameters because the gaps are not specified. The design is assumed to be correct only if the result value for each parameter is the same for all values of \( gaps \). The correctness condition is, therefore:

\[
\forall params. \forall gaps. results_{simpledesign}(params, gaps) = results_{algorithm}(params)
\] (2)

For a design at RTL a finite set of parameter combinations can be assumed. With (2) arbitrary large gaps (\( gaps(i) \) are natural numbers) between the input of the single parameters must be assumed. Therefore, SAT-based algorithms cannot be used for the verification of (2). Even simulation is limited to gaps of a finite size. The only workaround is the limitation of \( gaps \) to a fixed upper bound. This enables the automatic prove by SAT-based algorithms. From the practical point of view, such a limitation is acceptable in most cases, if the upper bound of the verified size of the gaps has been chosen large enough. The application of decision procedures such as CVC3 [6] that can reason about natural numbers is no solution here because the size of the gaps influences directly the number of loop iterations in the abstracted algorithm (see Section 2.1). However, the verification with CVC3 relies on a finite unrolling of the loops.

3.2 An Alternative Method

It seems to be not very handy to verify (2) either with simulation or with formal verification because the parameter space consists of \( gaps \) and \( params \). Assuming the generalized input-output scheme the design must produce the same result for arbitrary gaps between the parameters. The verification can, hence, be split into two sub problems. It can first be shown that the sizes of the gaps do not change the functionality of the design, i.e. that

\[
\forall params. \forall gaps. results_{simpledesign}(params, gaps) = results_{simpledesign}(params, (0, \ldots, 0))
\] (3)

is valid. If (3) has been shown the specified algorithm can then be compared directly with the algorithm abstracted from the simplified design for \( gaps = (0, \ldots, 0) \):

\[
\forall params. results_{simpledesign}(params, (0, \ldots, 0)) = results_{algorithm}(params)
\] (4)

The proof of (3) and (4) is as complex as the proof of (2). However, this alternative method allows an independent verification of the pure functionality (proof of (4)) and of the input-output scheme (proof of (3)). Assuming an incomplete verification, it can be chosen how deep the functionality or the input-output scheme is verified.

Moreover, this splitting allows the development of dedicated formal verification algorithms for sub problem (3). Basically, (3) can be proven by an induction proof over \( gaps \). This proof must show that incrementing one of the values of \( gaps \) does not change the values of \( results \). This seems to be possible for specific design architectures. A proof algorithm that can prove (3) for arbitrary architectures by induction appears to be unfeasible.

If the design has an internal signal \texttt{input\_phase} which is active between the input of the first and the last parameter a sufficient condition for (3) is

\[
\forall states.\forall inputs. (input\_phase = 1) \land (en_i = 0) \rightarrow (\delta(input, state) = state)
\] (5)
This condition could be applied only in rare cases because it assumes either a parallelization of the parameters that have been read in sequentially or an intermediate processing of each single parameter (e.g. accumulation).

4 Possible Generalizations

4.1 Variable Number of Parameters

The generalized input-output scheme described in Section 2.2 can be further generalized. In some applications, the number of parameters is not fixed for each run of the algorithm. Instead, it can vary in a defined range. For instance, a module for the computation of the mean value can be designed for 2...16 values. In this case, the specified algorithm and the input-output mapping have an additional input for the number \( n_{\text{params}} \) of the parameters of a computation.

The approach of the abstraction relies on the representation of the simplified design as a Mealy automaton with a fixed number of inputs. Hence, the upper bound for the number of parameters must be known and the array \( \text{params} \) must have this length. If less than the maximum number of parameters are applied it is assumed here as a convention that only the first entries of \( \text{params} \) are used. To support this generalization the input-output mapping must be adapted by the instantiation of an additional register \( n_{\text{params}}_r \) which is loaded with \( n_{\text{params}}_i \) when \( \text{en}_i = 1 \) and read instead of \( n_{\text{params}} \).

The design must also know the number of parameters for each computation. The design has either an additional input \( n_{\text{params}}_i \) which got the number of parameters together with the first parameter, or there is an additional input \( \text{last}_i \) which becomes active together with \( \text{en}_i \) at the last parameter. In both cases, the input-output mapping can easily be extended to produce these additional signals.

4.2 Variable Number of Results

There are designs where the number \( n_{\text{results}} \) of the result values is not the same for all parameter values. In this case the specified algorithm produces as an additional result the number of the result values. The design can provide the number of result values in three different ways. The computation of \( n_{\text{results}}_o \) of the input-output mapping depends on the used way:

1. \( n_{\text{results}} \) depends only on \( n_{\text{params}} \). In this case, the mapping \( n_{\text{params}} \mapsto n_{\text{results}} \) can be used immediately in the input-output mapping to compute \( n_{\text{results}}_o \).

2. The design has an output \( n_{\text{results}}_o \) for \( n_{\text{results}} \), which has a valid value together with the output of the first result (\( \text{eno}_s = 1 \)). The input-output mapping must store this value in an additional register \( n_{\text{results}}_r \) and read this register instead of \( n_{\text{results}} \). The value of \( n_{\text{results}}_r \) must be visible at the output \( n_{\text{results}}_o \) of the input-output mapping.

3. The design marks the last result by the activation of \( \text{rfd}_s. \text{res}_\text{cnt} \) must become the output \( n_{\text{results}}_o \) of the input-output mapping. It cannot be checked in the input-output mapping if the correct number of results is produced by the designs. Instead, this is checked implicitly during the comparison of \( n_{\text{results}}_o \) of both algorithms.

The comparison of the specified algorithm with the algorithm abstracted from the design must be restricted to the first \( n_{\text{results}}_o \) entries of \( \text{results} \). The comparison of the algorithms fails for trivial reasons if \( n_{\text{results}}_o \) is different for both algorithms.
4.3 Different Types of Parameters and Results

In the previous sections it is assumed that all parameter values that are applied sequentially to the design have the same type. In general, this type can be complex (records, arrays). However, it is assumed that from all elements that are aggregated in the parameter type the same number of results is needed.

The input-output scheme of Section 2.2 can easily be modified in a way that it supports different parameter types. For that, it is necessary to duplicate for all parameter types all inputs (such as \texttt{params\_i}, \texttt{en\_i}) and registers (such as \texttt{n\_params\_r}, \texttt{param\_cnt}) that are needed for the parameters. The input of all parameters is finished for such a design if \texttt{param\_cnt} has reached for all types the appropriate value for \texttt{n\_params}.

The results can also be split into different types. It holds the same as said above for parameters of different types. The output of the results for a computation is finished if \texttt{rfd\_s} becomes active.

5 Conclusion and Outlook

In this article is described how the method described in [15] can be extended to designs with more complex input-output schemes. It has been investigated, which generalization can be supported using an input-output mapping.

The generalization described in this article allows verifying designs with input-output schemes that are used in industrial designs. However, the approach described in this article excludes all pipelined designs. The method described in [15] is based on a cyclic change of four different phases: input, computation, output, and wait for the next input. Pipelined designs do not show such a cyclic behavior because in general the output of the first computation is not available when the parameters of the second computation are read in. For principle reasons, such an input-output scheme cannot be mapped to the cycle described above because the input-output mapping cannot anticipate future events. Pipelined designs require, therefore, a complete different kind of abstraction method to represent their behavior by an algorithm. It remains a topic of future research to find such a method.

The comparison of two algorithms as such is complex because the parameter space is usually very large and the comparison must regard the different number of runs of the loop. If the input-output scheme becomes more flexible the parameter space to be verified becomes even larger. This is inherent to the problem and does also affect the approach described in this article. If the size of the gaps between the single parameters is kept unbound the parameter space that must be investigated during verification becomes infinite. It seems that there are only two ways to handle this problem: Either, an upper bound for the size of the gaps is assumed. Then the parameter space becomes finite. However, it is much larger than the space for the parameters that are used for computation. Alternatively, certain design architectures are defined for which the equivalence of the output values independent from the size of the gaps according to (3) is known. The only feasible way seems to be the second one even if this requires some restrictions regarding the design architecture. This could be acceptable if the verification could be simplified significantly. (5) shows a condition for such a design structure. However, this structure must be further generalized to enable an industrial application.

References


