PAPER Special Section on Formal Approach

Word-Level Equivalence Checking in Bit-Level Accuracy by Synthesizing Designs onto Identical Datapath

Tasuku NISHIHARA[†a)], ***Member***, **Takeshi MATSUMOTO**[††], ***and*** **Masahiro FUJITA**[††], ***Nonmembers***

SUMMARY Equivalence checking is one of the most important issues in VLSI design to guarantee that bugs do not enter designs during optimization steps or synthesis steps. In this paper, we propose a new word-level equivalence checking method between two models before and after high-level synthesis or behavioral optimization. Our method converts two given designs into RTL models which have same datapaths so that behaviors by identical control signals become the same in the two designs. Also, functional units become common to the two designs. Then word-level equivalence checking techniques can be applied in bit-level accuracy. In addition, we propose a rule-based equivalence checking method which can verify designs which have complicated control structures faster than existing symbolic simulation based methods. Experimental results with realistic examples show that our method can verify such designs in practical periods.

key words: *high-level synthesis, behavioral synthesis, formal verification, equivalence checking*

1. Introduction

System-level design methodology plays an important role to improve VLSI design productivity and shorten design periods. C or C++ based design languages such as SpecC [1] or SystemC [2] are uniformly used to describe both hardware and software parts.

A typical VLSI design flow starting from system-level is shown in Fig. 1. First, the specification of a design is described without distinguishing the hardware and software parts. Then, they are transformed into a software program and an RTL design, respectively, through hardware/software partitioning and high-level synthesis processes. In this design flow, designs are gradually refined in step-by-step. Therefore, design bugs can be newly inserted during the refinements. It is important to detect such bugs in early steps in order to prevent design modifications in later steps, since they take much higher costs. Equivalence checking gives a solution for that goal. When two designs are proved to be equivalent and one of them has no bugs, the other is also guaranteed to have no bugs. We can apply equivalence checking to designs before and after each refinement step, which enables us to avoid bug insertions when the original design has been verified sufficiently.

Manuscript received July 22, 2008.
Manuscript revised November 20, 2008.

[†]The author is with the Department of Electronics Engineering, The University of Tokyo, Tokyo, 113–8656 Japan.
[††]The authors are with the VLSI Design and Education Center (VDEC), The University of Tokyo, Tokyo, 113–8656 Japan.
a) E-mail: tasuku@cad.t.u-tokyo.ac.jp

DOI: 10.1587/transinf.E92.D.972

One of the simplest equivalence checking methods is proposed in [3]. It translates designs into Boolean formulas, and checks the equivalence of those formulas with BDD or SAT. However, large designs cannot be verified, since the complexity of such a bit-level analysis increases exponentially with the size of designs.

To avoid bit-level analysis as much as possible, word-level symbolic simulation [4]–[6] which treats each variable or operator as a symbol is applied. However, since the complexity of symbolic simulation is doubled for each conditional branch, it is still not applicable to entire designs. Also, loops are not acceptable, and they must be unrolled in advance.

Therefore, only textually different parts of two designs are compared in [5]. This method can handle large designs when compared designs are similar. Also in [6], equivalences of paths between conditional branches are checked, and the results are gathered to prove the entire equivalence. To apply this divide-and-conquer approach, correspondences of intermediate variables or registers between two designs must be known or given by users (e.g. Names of variables or registers in two designs are same).

However, in practical refinement steps, it is usual that the entire structure of a design is changed or correspondences of intermediate variables or registers are unknown (e.g. between two designs before and after automated high-level synthesis). Also, since bit-width or sign are not taken into account in symbolic simulation, bit-level accuracy is not considered in [4]–[6].

Based on the arguments above, in this paper, we propose a new equivalence checking method between two models before and after a refinement step, such as high-level synthesis or behavioral optimization. In this method, we focus on a feature that designs after automated high-level synthesis are usually composed of controllers and datapaths. In such a design, computations of the design are executed at the datapath, and the controller determines computations executed at each clock cycle. In our method, two designs are converted into RTL models which have same datapaths. Then, we can get advantages of abstracting computations of the datapaths, and concentrate on the verification of the controllers. Concretely, since the datapaths are identical, the functional units in those RTL models become the same. Also, same control signals from the controllers represent same behaviors, and then the behaviors are equivalent in bit-level accuracy. Therefore, existing word-level methods,

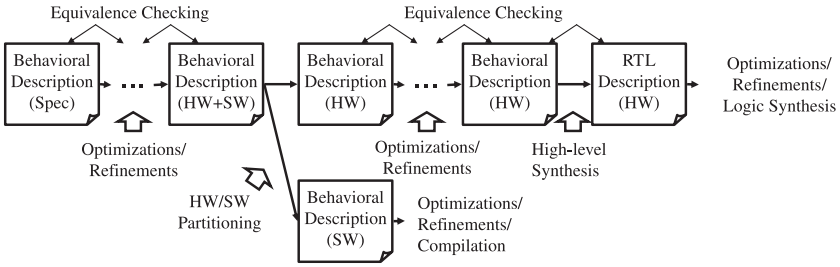

Fig. 1 VLSI design flow from system-level.

such as symbolic simulation, can be easily applied in bit-level accuracy.

However, since correspondences of intermediate variables or registers are not given in most cases, we have to compare entire designs in such cases. As discussed above, symbolic simulation cannot handle designs which include large numbers of conditional branches or loops whose numbers of iterations are dependent to input values or infinite. Therefore, we propose a new word-level method which propagates equivalences of inputs to those of outputs with pre-defined rules. Since the rules are proposed to handle conditional branches and loops, the proposed rule-based method can be used as a complement of symbolic simulation based methods.

The remainder of this paper is organized as follows: Section 2 explains existing techniques used in our method. Section 3 describes our verification flows. In Sect. 4, we explain the proposed verification algorithms used in the flows. We report the experimental results with realistic examples in Sect. 5. In Sect. 6, we give a conclusion of this work and show our future directions.

2. Basic Notions

2.1 FSMD (Finite State Machine with Datapath)

FSMD [7] is a specification description of sequential RTL design. In an FSMD, control and computation of a design are specified separately. The control is specified as a controller in FSM style, and the computation is specified as a datapath. Though there are many different definitions of FSMD [6]–[9], we define FSMD in our own style as follows.

Definition 1 (**Finite state machine with datapath).** An FSMD is defined as a tuple $M = <D, C>$, where D is a datapath with data registers, and C is a controller FSM.

A datapath is defined as a septuple:

$$D = <I, O, V, K, F, F_{call}, A>$$

where I is a set of inputs, O is a set of outputs, V is a set of data registers, K is a set of constants, F is a set of functions each of which represents a functional unit, F_{call} is a set of function calls, and A is an assignment relation. F_{call} is defined as a set of vector:

$$F_{call} = \{<f, e_1, \ldots, e_n> \mid f \in F, e_1, \ldots, e_n \in E_R\}$$

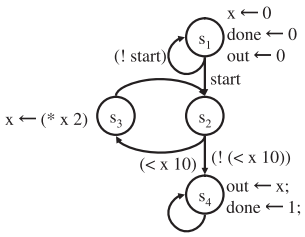

Fig. 2 An example of FSMD.

where f is the symbol of a called function, $e_1, \ldots, e_n$ are arguments, n is a number of the arguments, and $E_R = I \cup V \cup K \cup F_{call}$ represents a set of expressions which can be right-hand side expressions of assignments or arguments of function calls. A is defined as:

$$A \subseteq (O \cup V) \times E_R$$

and represents an assignment relation between data registers or outputs to be updated and expressions which represent new values.

A controller is defined as a sextuple:

$$C = <D, S, \alpha, R, P, Q>$$

where D is a datapath defined above, S is a set of control states, $\alpha \in S$ is an initial state, $R \subseteq S \times S$ is a transition relation, $P \subseteq S \times A$ represents a relation between states and assignments executed at the states, and $Q{:}R \rightarrow E_R$ is a function which returns the condition that a transition is performed. Note that an executed transition from a state is determined to only one with a given condition, which means transition conditions of all transitions from a state are exclusive.

Assignments are executed when the FSMD reaches the states to where the assignments belong. Also, in each state, for every register and for every output $e_l \in V \cup O$, there must be one and only one assignment to e_l such as $<e_l, e_r> \in A$, where $e_r \in E_R$.

Figure 2 shows an example of FSMD. It repeats doubling an input $in \in I$ while it is smaller than 10. If it becomes equivalent to or greater than 10, then the number is assigned to an output $out \in O$. The FSMD has four states ($S = \{s_1, \ldots, s_4\}$), and $\alpha = s_1$ is the initial state. The other symbols in the FSMD are as follows: $I = \{in, start\}$, $V = \{x\}$, $O = \{out, done\}$, $K = \{0, 1, 2, 10\}$, $F = \{*, !, <\}$. A left

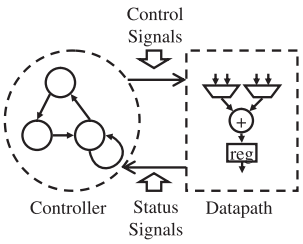

Fig. 3 A typical RTL design after high-level synthesis.

arrow (←) represents an assignment which is included in A. Assignments to keep previous values of registers, such as $x \leftarrow x$ in s_2, are omitted in the figure. Function calls are written in Lisp-like style. When an FSMD transits to a state, all assignments of the state are executed simultaneously. The expression described on each state transition represents its transition condition.

2.2 Separation of Designs' Equivalence to That of Controllers and Datapaths

The basic idea of our method is proposed in [10]. An RTL design generated by high-level synthesis is usually composed of a controller and a datapath as shown in Fig. 3. At each clock cycle, first, the controller sends control signals to the datapath, depending on the current state. Next, the datapath executes operations based on the control signals. Finally, the datapath returns status signals to the controller, and the controller determines the next state. Then, in [10], a behavioral design is mapped to a virtual controller and a virtual datapath, so that the equivalence can be separated into the equivalences of the (virtual) datapaths and the (virtual) controllers.

If the two datapaths are not the same, we have to compare both of the controllers and the datapaths. We must check the equivalence of the datapath operations under each pair of the control signals which is a candidate to be equivalent. This step might be time consuming since usually we do not know the correspondences of the control signals nor the status signals between the two datapaths. On the other hand, if the two datapaths are the same, we do not have to compare the datapaths. In addition, two controllers generated from equivalent designs can be similar since they are for the identical datapath. Then, we can apply an equivalence checking method based on the difference of controllers which is similar to [5] so that large designs can be verified.

Our method extends this approach, by forcibly making the datapaths of two designs the same by generating controller(s) for an identical datapath which are equivalent to the original design(s). This method is described in Sect. 3. Also, since only a brief approach to compare the controllers is shown in [10], we give a concrete method in Sect. 4.

2.3 NISC (No Instruction Set Computer) Compiler

NISC [11] is a computer architecture which is composed of an arbitrary datapath and its controller. Different from the other computer architecture, NISC has no instruction set, and a set of control signals is directly stored in a control memory instead of a set of instructions. Those control signals are called as "control words", and include not only signals which control the operations of the datapath, but also next values of the program counter. This structure enables us to use an arbitrary datapath, since we do not have to newly define an instruction set for it. Users can give suitable datapaths for their requirements by specifying their structures (i.e. numbers of various computation units, registers, data memories, bus-widths, and their connections). NISC compiler can generate a set of control words for any given datapath from an ANSI-C code, if the datapath has sufficient resources to execute the code. Thus, NISC architecture can achieve both high-performance of custom hardware and flexibility of software.

The aim of NISC compiler is the same as that of our method in the point that it generates a control for a given datapath. In NISC compiler, this process is performed by the following steps. First, a Data Flow Graph (DFG) is created from an input ANSI-C code. Next, the DFG is traversed backwardly from the outputs, and each operation is assigned to a functional unit at a cycle in the datapath with ALAP like scheduling. Multiple operations can be mapped to a single cycle while resources (functional units and lines of buses) are enough. At this step, delay of the functional units is considered. This avoids creating long paths of the functional units for a single cycle. Finally, control words to be stored in the control memory are generated. The control words include:

- Signals to the multiplexers in the datapath which correspond to the values of the program counter
- Next values of the program counter which can be considered as next states
- Constants used in the operations at the datapath

The above method is quite simple and reasonable. Since we are focusing on verification and do not have to consider the performance, we can use a similar (or simpler) solution. The method is discussed in Sect. 3.2.

2.4 Equivalence Checking with Symbolic Simulation

As briefly explained in Sect. 1, symbolic simulation is a simulation where values of variables and meanings of operations are not interpreted. Exhaustive analyses can be performed, since one symbol can express all values of a variable.

In [4], [5], equivalence class based methods are proposed. Equivalence class is a set where expressions in a same class are equivalent. Equivalence class based symbolic simulation is performed by allocating expressions to equivalence classes.

[4], [5] are targeting on equivalence checking of ANSI-C designs. Before the verification, all loops must be unrolled. First, each pair of corresponding inputs is assigned

to a same equivalence class. Verification is performed for each execution path based on the equivalences of inputs. For each assignment on the path, since its left hand side and right hand side are equivalent, those expressions are assigned to a same equivalence class. Finally, if each pair of corresponding outputs is in a same equivalence class, the two designs are proved to be equivalent.

We apply this method to check the equivalence of two FSMDs. Its detail is explained in Sect. 4.5

3. Generation of RTL Designs with Identical Datapath

3.1 Verification Flow

Based on the argument in Sect. 2.2, in the proposed method, we make datapaths of two designs identical. If they are identical, we can get the following advantages.

- Same control signals represent behaviors which are equivalent in bit-level accuracy
- Controllers generated from equivalent designs tend to be similar since they are generated for an identical datapath.

Figure 4 shows the verification flow to check the equivalence between designs before and after high-level synthesis. One design is a behavioral design and the other is an RTL design. We assume that the RTL design is composed of a controller and a datapath, and easily separated into them. As we have mentioned in Sect. 2.2, results of high-level synthesis usually satisfy the assumption. If the assumption is not satisfied, we have to separate them by determining its state variables. Next, from the behavioral design, we generate a controller for the datapath in the RTL design. The generated controller is written in RTL, and it must represent the same behavior as the behavioral design. Details of this step are described in Sect. 3.2. Then, we can get the two controllers for the identical datapath. Comparison methods for those designs are described in Sect. 4.

A similar method can also be applied to check the equivalence between two designs before and after behavioral optimization, and its flow is shown in Fig. 5. Input designs are both in behavioral level. The difference between the previous flow is that we must give a new datapath to generate controllers for the datapath, since neither of the input designs are RTL. The datapath should be as simple as possible, since same arithmetic operations in the designs should be executed by a same set of functional units in the datapath. If same operations are executed by different sets of functional units, the equivalence between those sets must be checked in bit-level.

Here we discuss about the possibility of false-positive case (A case that two designs are proved to be equivalent, although the two designs are actually not equivalent).

In the equivalence checking between designs before and after high-level synthesis, such a case happens only when differences between two designs disappear after the conversion which generates a controller for the datapath of the RTL design from the behavioral design. The possibility that this case happens can be dramatically decreased by using different synthesis tools on the two conversions (high-level synthesis and controller generation).

In the equivalence checking between designs before and after behavioral optimization, such a case also happens when the differences between two designs disappear after the conversions. However, the possibility that such the case happens is much smaller than the case that bugs are inserted during an optimization by hand.

3.2 Generation of a Controller for a Given Datapath

As we have mentioned in Sect. 2.3, we can use a similar method to NISC compiler to generate a controller for a given datapath. With the following limitations, we can directly apply the scheduling method of NISC compiler.

- Buses can only be used to transmit inputs, outputs, and status signals to the controller
- Use of datapath memories is prohibited (since it cannot be represented in FSMD)
- Delay of functional units can be neglected (since we do not have to consider the performance)

As the next step, each of generated control words is divided into signals to multiplexers and next values of the program counter. The signals to the multiplexers correspond to the control signals in Fig. 3, and the next values of the program counter corresponds to next states of the controller. Therefore, we can easily generate an RTL controller without control memory from them.

However, the above method fails in the following cases.

- Optimizations with limitations of input values, such as bit-width reduction and table-lookup division, are applied

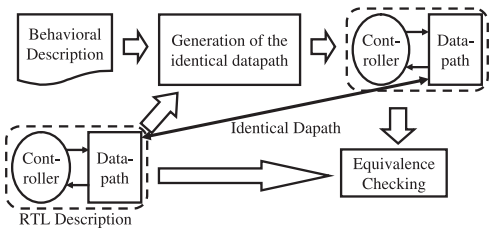

Fig. 4 Proposed equivalence checking flow between a behavioral level design and an RTL design.

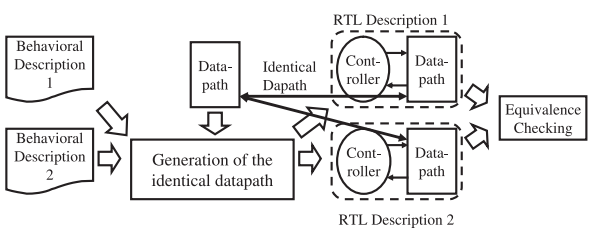

Fig. 5 Proposed equivalence checking flow between two behavioral level designs.

- Precisions of variables or operations are different in two designs, such as floating value and fixed-point value.
- Operations of corresponding computations are different in two designs, such as constant multiplication, and bit-shift with addition

Since two designs are not logically equivalent in the first two cases, our method cannot handle them.

For the last case, we can extend the controller generation method by giving information about equivalences of operations. When we append such information, we must guarantee the correctness in bit-level, since it affects the accuracy of equivalence. The correctness can be checked with decision procedure (SMT solver), such as CVC3 [12]. However, this solution is difficult to be applied when we use an external tool such as NISC compiler, since we have to give information about equivalences of operations internally. For such a case, we can add circuits which perform the lacking computations to the datapath, and re-generate a controller for the modified datapath.

4. Equivalence Checking of RTL Models Which Have Same Datapaths

4.1 Equivalence Checking in Bit-Level Accuracy

As shown in Figs. 4 and 5, inputs of the final step of our verification flows are RTL models which have identical datapaths. Since the datapaths are identical, same control signals represent same behaviors with same sets of functional units. Since, operations executed by those control signals are equivalent in bit-level, we can apply word-level equivalence checking methods such as symbolic simulation in bit-level accuracy.

However, this bit-level accuracy may be too limited to verify various designs. In such a case, we can check equivalences among operations executed by different sets of functional units. Some candidates to be checked are listed below. All operations are written in Lisp-like style.

- Commutative law
  e.g. $(+\ a\ b) \equiv (+\ b\ a)$
- Scalar multiplication executed by addition $\times n$
  e.g. $(*\ a\ 2) \equiv (+\ a\ a)$
- Scalar multiplication executed by shift + addition
  e.g. $(*\ a\ 5) \equiv (+\ (<<\ a\ 2)\ a)$, where << represents a shifter-left operation.

Such equations can be checked by property checkers or equivalence checkers for combinational circuits. Operations of a datapath are fixed with a given control signals, and we can check the equivalence between circuit portions which are related to the operations corresponding to an equation. These portions must be combinational circuits.

As we described in Sect. 3.2, equivalence of operations is considered in both the controller generation stage and this equivalence checking stage. If much equivalence is considered in one stage, the effort of the other stage is reduced.

However this stage is required when there are multiple ways to perform an operation in a datapath (e.g. both a multiplier and a shifter exist in the datapath for constant multiplication), since the operation can be mapped differently in two designs with the method in Sect. 3.2. In such a case, we can perform a verification with the method described in this section.

4.2 Input of Equivalence Checking

We describe two RTL models which are inputs of the equivalence checking stage as FSMDs. Since the RTL models have already been separated to controllers and datapaths, we can easily describe them in FSMD. In those FSMDs, functional units in the RTL datapaths are represented as functions in F. We can consider some functions as identical functions if they are proved to be equivalent in the process described in the previous section. Also, we assume that correspondences of inputs and outputs between two FSMDs are known. These correspondences are required to define the equivalence of two designs. In our method, these correspondences must be given by users.

4.3 Definition of Equivalence

In this section, we define some notations and equivalences. Also, equivalences of inputs and outputs which must be given by users are explained.

Before defining the notations or the equivalences, we define two sets. $E = E_R \cup O$ is a set of expressions including outputs. $T = \{t \mid t \in R^n, 1 \leq n\}$ is a set of vector which represents executable sequences of state transitions, where n is the length of the sequence.

From this section, we denote a symbol X_1 as a symbol X of the first design. Also, a symbol X_2 denotes a symbol X of the second design. We compare two FSMDs M_1 and M_2. Since the datapaths of M_1 and M_2 are the same, we can assume $F_1 = F_2$ and describe them as F.

First, we define symbolic values of expressions.

Definition 2 (Symbolic value at state). Let

$$Z_S \subseteq (E_1 \times S_1) \cup (E_2 \times S_2)$$

denote a set of symbolic values at states, and $<e, s> \in Z_S$ denote the symbolic value of an expression e at a state s. Since $<e, s>$ is symbolic, it represents all values of e at s. Concrete values and a number of arrival times are abstracted.

$<e_1, s_1> \in Z_S$ and $<e_2, s_2> \in Z_S$ are equivalent, when the following conditions are satisfied.

- Conditions to reach s_1 and s_2 from the initial states for the same number of times are equivalent.
- Values of e_1 and e_2 are always equivalent when arriving at s_1 and s_2 for the same number of times, respectively

We denote the equivalence by an operator "≡" as

$$<e_1, s_1> \equiv <e_2, s_2>$$

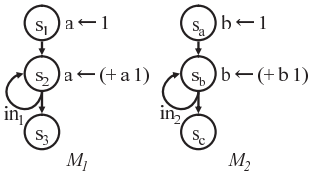

Fig. 6 Example for equivalence definitions.

For example, in Fig. 6, $<a, s_2> \equiv <b, s_b>$ is true when inputs in_1 and in_2 are corresponding. Here, we represent the values of in_1 and in_2 at k-th cycle as in_1^k and in_2^k, respectively. Conditions to reach s_2 and s_b for n times are $\bigwedge_{i=1}^{n-1}(in_1^i)$ and $\bigwedge_{i=1}^{n-1}(in_2^i)$, respectively. Then the first condition of the definition is satisfied. The values of a and b on nth arrivals at s_2 and s_b, respectively, are both n. Therefore, the second condition is satisfied, and the equivalence is valid.

Definition 3 (**Symbolic value on transition).** Let

$$Z_T \subseteq (E_1 \times T_1) \cup (E_2 \times T_2)$$

denote a set of symbolic values on sequences of state transitions, and a pair $<e, t> \in Z_T$ denote the symbolic value of an expression e on a sequence of state transition $t = <r_1, r_2, \ldots, r_n>$, where $r_k = <s_k, s_{k+1}> \in R$. Since $<e, t>$ is symbolic, it represents all values of e at $s_{n+1} \in S$ when M_1 or M_2 transits through t. Concrete values and a number of transition times are abstracted.

$<e_1, t_1> \in Z_T$ and $<e_2, t_2> \in Z_T$ are equivalent, when the following conditions are satisfied.

- Conditions to transit through t_1 and t_2 from the initial states for the same number of times are equivalent.
- When $t_1 = <r_{11}, r_{12}, \ldots, r_{1n}>$, where $r_{1k} = <s_{1k}, s_{1k+1}>$, and $t_2 = <r_{21}, r_{22}, \ldots, r_{2m}>$, where $r_{2j} = <s_{2j}, s_{2j+1}>$, values of e_1 and e_2 are always equal when arriving at s_{1n+1}, s_{2m+1} for the same number of times, respectively.

We denote the equivalence by an operator "≡" as

$$<e_1, t_1> \equiv <e_2, t_2>$$

For example, in Fig. 6, $<a, <s_2, s_2>> \equiv <b, <s_b, s_b>>$ is true when in_1 and in_2 are corresponding. Conditions to transit through the transitions $<s_2, s_2>$ and $<s_b, s_b>$ for n times are $\bigwedge_{i=1}^{n}(in_1^i)$ and $\bigwedge_{i=1}^{n}(in_2^i)$, respectively. Then the first condition of the definition is satisfied. The values of a and b on nth arrivals at s_2 and s_b, respectively, are both n. Therefore, the second condition is satisfied, and the equivalence is valid.

Any function can be applied to those symbolic expressions, $<e_1, s_1> \in Z_S$ and $<e, t> \in Z_T$. If all argument expressions of a function are at a same state or on a same transition, such a function can be considered as a function at the state or on the transition. This can be represented with the next equations.

$$(f\ <e_1, s>\ <e_2, s>\ \ldots) \equiv <(f\ e_1\ e_2\ \ldots), s>$$

$$(f\ <e_1, t>\ <e_2, t>\ \ldots) \equiv <(f\ e_1\ e_2\ \ldots), t>$$

where $f \in F$, $e_1, e_2, \ldots \in E$, $s \in S$, $t \in T$, and functions are represented in Lisp-like style.

Also, when two symbolic values $z_1, z_2 \in Z_S \cup Z_T$ are equivalent and a part of a symbolic value $z_3 \in Z_S \cup Z_T$ corresponds to z_1, we can substitute the part as the substitution of z_1 with z_2. z_3s before and after the substitution are equivalent. For example, when $<e_1, s_1> \equiv <e_2, s_2>$ is true, $<(f\ e_1), s_1> \equiv <(f\ e_2), s_2>$, where $f \in F$, becomes true by substituting e_1 at s_1 with e_2 at s_2.

Next, we define equivalence classes each of which represents a set of equivalent symbolic values.

Definition 4 (**Equivalence class).** Equivalence class of states is a set $Z_{S1} \subseteq Z_S$ that all contained elements are equivalent. Similarly, equivalence class of sequences of state transitions is a set $Z_{T1} \subseteq Z_T$ where all contained elements are equivalent. If same elements are contained in more than one equivalence classes, we can merge them into a single equivalence class.

Then, correspondences of inputs and outputs which are given by users are described by sets of equivalence classes as follows.

$$\{\{<in_{1i}, s_{1in_i}>, <in_{2i}, s_{2in_i}>\} \mid 1 \le i, in_{1i} \in I_1, in_{2i} \in I_2, s_{1in_i} \in S_1, s_{2in_i} \in S_2\}$$

$$\{\{<out_{1i}, s_{1out_i}>, <out_{2i}, s_{2out_i}>\} \mid 1 \le i, out_{1i} \in O_1, out_{2i} \in O_2, s_{1out_i} \in S_1, s_{2out_i} \in S_2\}$$

s_{1in_i} and s_{2in_i} are states where the ith inputs are valid, respectively. s_{1out_i} and s_{2out_i} are states where the ith outputs are valid, respectively.

Here, the equivalence of two designs (FSMDs) is defined as follows.

Definition 5 (**Equivalence of FSMDs).** Two FSMDs M_1 and M_2 are equivalent, when

$$\bigwedge_i (<in_{1i}, s_{1in_i}> \equiv <in_{2i}, s_{2in_i}>) \Longrightarrow \bigwedge_i (<out_{1i}, s_{1out_i}> \equiv <out_{2i}, s_{2out_i}>)$$

is true, where $in_{1i} \in I_1$ and $out_{1i} \in O_1$ are the i th input and output of M_1, respectively, and $in_{2i} \in I_2$ and $out_{2i} \in O_2$ are the i th input and output of M_2, respectively.

Therefore, in our method, equivalences between all corresponding outputs of two designs are checked with an assumption that all corresponding inputs of the two designs are equivalent.

4.4 Equivalence Checking of Symbolic Expressions

To apply the equivalence checking method explained in the

latter two sections, equivalences of the symbolic expressions are checked. This section explains how to check the equivalence of the symbolic expressions for states or sequences of state transitions defined in the previous section.

A symbolic expression consists of an expression ($e \in E$), and a state ($s \in S$) or a sequence of state transition ($t \in T$), and an expression consists of a combination of variables, inputs, outputs ($V_1 \subseteq I \cup O \cup V$), and functions ($F_1 \subseteq F$).

With the relation described in the previous section, a function at a state or on a sequence of state transitions, such as $<(f\ e_1\ e_2\ \ldots), s>$ or $<(f\ e_1\ e_2\ \ldots), t>$, where $f \in F$, $e_1, e_2, \ldots \in E$, $s \in S$, $t \in T$, can be converted into $(f\ <e_1, s>\ <e_2, s>\ \ldots)$ or $(f\ <e_1, t>\ <e_2, t>\ \ldots)$, respectively. We repeat applying this conversion to symbolic expressions while it can be applied.

Then, the expressions are represented only with variables at states, variables on sequences of state transitions (we denote them as symbolic variables), and functions which are applied to those symbolic variables. Here, we treat a symbolic variable as an unit, and same symbolic variables (same variables at same states or on same sequences of state transitions) are equivalent.

With a conversion that each unit into a variable and each function into an Uninterpreted Function (UF), we can apply methods of Logic of Equality with Uninterpreted Function (EUF) [13] to check the equivalence. If an EUF formula that says two symbolic expressions are equivalent is valid, the two symbolic expressions are proved to be equivalent.

Here, to improve the possibility to prove such equivalences, it is important to make as much functions as same UFs. As explained in Sect. 3, by the conversion which makes two designs have identical datapaths, computations with same control signals from controllers can be converted into same UFs. Also, expressions which are proved to be equivalent in Sect. 4.1 can be converted into the same UFs.

Symbolic expressions in an equivalence class are also assumed to be equivalent when we check the validity of the EUF formulas. Practically, this step is performed with decision procedures (SMT solvers) which can handle EUF, such as CVC3 [12].

4.5 Equivalence Checking of FSMDs by Symbolic Simulation

In this section, we explain how to apply equivalence class based symbolic simulation introduced in Sect. 2.4 to check the equivalence of FSMDs defined in Sect. 4.3.

Before the verification, we must unroll all loops. The verification is performed as follows.

1. From the initial states, transitions are traversed forwardly with getting equivalences of left-hand sides and right-hand sides of assignments. The left-hand side value at the next state is equivalent to the right-hand side value at the current state.
2. When there are more than one next states, the current checking process forks.
3. FSMDs are equivalent when in all checking paths the equivalence of FSMDs (Definition 5) is satisfied.

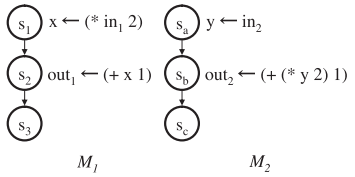

Fig. 7 Example 1.

Here, we show an example of the verification with two FSMDs in Fig. 7. $s_1, s_2, s_3 \in S_1$ and $s_a, s_b, s_c \in S_2$ are states, $x \in V_1$ and $y \in V_2$ are variables, $in_1 \in I_1$ and $in_2 \in I_2$ are corresponding inputs, $out_1 \in O_1$ and $out_2 \in O_2$ are corresponding outputs, $1, 2 \in K_1 \cap K_2$ are constants, and $+, * \in F$ denotes addition and multiplication, respectively. A given initial equivalence class is

$$\{<in_1, s_1>, <in_2, s_a>\}$$

The output equivalence to be proved is

$$<out_1, s_3> \equiv <out_2, s_c>$$

First, from the assignments in s_1 and s_a, the following equations becomes true:

$$<x, s_2> \equiv <(* \ in_1\ 2), s_1>$$
$$<y, s_b> \equiv <in_2, s_a>$$

Then, with substitutions, the equivalence classes become:

$$\{\{<in_1, s_1>, <in_2, s_a>, <y, s_b>\},$$
$$\{<x, s_2>, <(* \ in_1\ 2), s_1>,$$
$$<(* \ y\ 2), s_b>\}\}$$

Next, from the assignments in s_2 and s_b, we get the following equations:

$$<out_1, s_3> \equiv <(+ \ x\ 1), s_2>$$
$$<out_2, s_c> \equiv <(+ \ (* \ y\ 2)\ 1), s_b>$$

Then, with substitutions, the equivalence classes become:

$$\{\{<in_1, s_1>, <in_2, s_a>, <y, s_b>\},$$
$$\{<x, s_2>, <(* \ in_1\ 2), s_1>,$$
$$<(* \ y\ 2), s_b>\},$$
$$\{<out_1, s_3>, <(+ \ x\ 1), s_2>,$$
$$<(+ \ (* \ y\ 2)\ 1), s_b>\},$$
$$\{<out_2, s_c>, <(+ \ (* \ y\ 2)\ 1), s_b>\}\}$$

Since the third and forth equivalence classes include the same entry, we can merge them. The new equivalence class includes both $<out_1, s_3>$ and $<out_2, s_c>$, then M_1 and M_2 are proved to be equivalent.

As we mentioned in Sect. 1, this method is fast and reasonable only when there are a small number of control

branches and loops whose numbers of iterations are small. Also, if there are infinite loops, we can get only limited results. For such cases, we propose a rule-based method explained in the next section.

4.6 Rule-Based Equivalence Propagation

In this section, we explain our rule-based equivalence checking method. This method can be applied to FSMDs directly, and four rules explained below propagate equivalences of inputs to those of outputs.

4.6.1 Rules of Equivalence

Rule 1. *If* $r = <s_1, s_2> \in R$ *satisfies:*

$$(\forall(s_a \neq s_2)(<s_1, s_a> \notin R)) \wedge$$
$$(\forall(s_b \neq s_1)(<s_b, s_2> \notin R)),$$

then the next equation becomes true:

$$\forall(<s_1, a> \in P \,|\, a = <e_1, e_2>)$$
$$(<e_1, s_2> \equiv <e_2, s_1>)$$

Proof. r is the only transition from s_1, and it is also the only transition to s_2. Therefore, when M_1 or M_2 reaches to s_1, it always reaches to s_2 by the next transition. Then, the transition condition to reach s_1 and s_2 for the same number of times from the initial state must be equivalent. Also, the values of the left-hand sides of the assignments are always updated to the values of the right-hand sides after the transition. Therefore, the value of e_1 at s_2 is always equivalent to that of e_2 at s_1, when arriving at s_1 and s_2 for the same number of times, respectively. Since the two conditions in Definition 2 are satisfied, $<e_1, s_2> \equiv <e_2, s_1>$ becomes true. □

This rule corresponds to symbolic simulation explained in the previous section, and returns the same results when there are no conditional branches in FSMDs. Figure 7 shows an example where this rule can be applied, and the results are the same as described in the previous section.

Rule 2. *Let* t_1 *and* t_2 *be sequences of state transitions such that:*

$$t_1 = <<s_{11}, s_{12}>, \ldots, <s_{1m}, s_{1m+1}>> \in T_1$$
$$t_2 = <<s_{21}, s_{22}>, \ldots, <s_{2n}, s_{2n+1}>> \in T_2$$

Also, let c_{1i} *and* c_{2i} *be transition conditions for each of the transitions in* t_1 *and* t_2*, respectively, such that:*

$$c_{1i} = Q(<s_{1i}, s_{1i+1}>) \,|\, 1 \leq i \leq m$$
$$c_{2i} = Q(<s_{2i}, s_{2i+1}>) \,|\, 1 \leq i \leq n$$

When we assume that t_1 *and* t_2 *are executed, Rule 1 can be applied for all transitions in them, since there are no joins and branches. Under the assumption, for each* $<e_1, e_2> \in E \times E$*, if*

$$(\bigwedge_{1 \leq i \leq m} (<c_{1i}, s_{1i}>) \equiv \bigwedge_{1 \leq i \leq n} (<c_{2i}, s_{2i}>)) \wedge$$

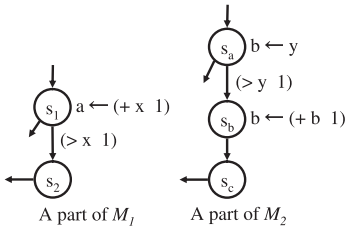

Fig. 8 Example 2.

$$(<e_1, s_{1m+1}> \equiv <e_2, s_{2n+1}>)$$

is true, $<e_1, t_1> \equiv <e_2, t_2>$ *becomes true, where* $\bigwedge$ *represents AND.*

Proof. If the second half part of the equation is true, from Definition 2, transition conditions to reach the last states in t_1 and t_2 for the same number of times from the initial states must be equivalent. Also, from this part, the second condition in Definition 3 is satisfied. In addition, from the first half part of the equation, transition conditions between t_1 and t_2 are equivalent. Then, with taking conjunction of those transition conditions, respectively, the conditions to transit t_1 and t_2 for the same number of times from the initial states become equivalent. Then, the first condition in Definition 3 is satisfied. Since the two conditions in Definition 3 are satisfied, $<e_1, t_1> \equiv <e_2, t_2>$ becomes true. □

Figure 8 shows an example where Rule 2 can be applied. $s_1, s_2 \in S_1$ and $s_a, s_b, s_c \in S_2$ are states. $a, x \in V_1$ and $y, b \in V_2$ are variables. $+, > \in F$ denote addition and less than operation, respectively. $1 \in K_1 \cap K_2$ is a constant. We give an initial equivalence class as follows:

$$\{<x, s_1>, <y, s_a>\}$$

First, we assume that M_1 transits through $t_1 = <<s_1, s_2>>$, and M_2 transits through $t_2 = <<s_a, s_b>, <s_b, s_c>>$. Then, with applying Rule 1, we get the following equivalence classes.

$$\{\{<x, s_1>, <y, s_a>, <b, s_b>\},$$
$$\{<a, s_2>, <(+\ x\ 1), s_1>\}$$
$$\{<b, s_c>, <(+\ b\ 1), s_b>\}\}$$

Since $<(>\ x\ 1), s_1> \equiv <(>\ y\ 1), s_a>$ becomes true from the first equivalence class, the transition conditions are equivalent. Also, from a substitution and a merger, we get the following equivalence class.

$$\{<a, s_2>, <(+\ x\ 1), s_1>, <b, s_c>,$$
$$<(+\ b\ 1), s_b>\}$$

Then, $<a, s_2> \equiv <b, s_c>$ becomes true. Finally, from Rule 2, we can prove $<a, t_1> \equiv <b, t_2>$.

Rule 3. *Let* $T_a \subseteq T$ *be a set of sequences of state transitions,* $S_{T_a} \subseteq S$ *be a set of states included in the transitions of* T_a*,* $s \in S$ *be a state. If there is no sequence of state transitions* $t \in T$*, where its first state is* s *and the states in* t *are not*

included in S_{T_a}, T_a covers all paths to s.

Let T_{a1} and T_{a2} denote sets of sequence of states transitions such that

$$T_{a1} = \{t_{1i} \mid 1 \le i \le m, t_{1i} \in T_1\}$$
$$T_{a2} = \{t_{2i} \mid 1 \le i \le n, t_{1i} \in T_2\}$$

which reach states $s_1 \in S_1$ and $s_2 \in S_2$ with covering all paths to s_1 and s_2, respectively.

Then the next formula becomes true.

$$(m = n) \wedge \bigwedge_{i=1}^{m} (<e_1, t_{1i}> \equiv <e_2, t_{2i}>)$$
$$\Longrightarrow <e_1, s_1> \equiv <e_2, s_2> \mid e_1 \in E_1, e_2 \in E_2$$

This rule shows that if all paths to s_1 and s_2 have corresponding paths where e_1 and e_2 are equivalent, then the values of e_1 at s_1 and e_2 at s_2 are always equivalent. In this rule, the number of corresponding paths in the two FSMDs must be same. It means FSMDs which have the same structures of conditional branches can be verified with this rule. This condition is also valid in Rule 4, since Rule 3 is performed to apply Rule 4.

Proof. Each equivalence of corresponding sequences of state transitions shows that, transition conditions to transit through those transitions from the initial states are equivalent, respectively. Therefore, the orders to reach s_1 and s_2 among those corresponding transitions are fixed, and completely equivalent in each pair of corresponding transitions. Then, the first condition in Definition 2 is satisfied. Also, the second condition in Definition 2 is clearly satisfied by the equivalences of e_1 and e_2 on corresponding sequences of state transitions. Therefore, both the conditions in Definition 2 are satisfied, and $<e_1, s_1> \equiv <e_2, s_2>$ is proved to be true. □

Figure 9 shows an example where Rule 3 can be applied. $s_1, s_2, s_3, \in S_1$ and $s_a, s_b, s_c \in S_2$ are states, and $a \in V_1$ and $b \in V_2$ are variables.

$$t_1 = <<s_1, s_3>>, \quad t_2 = <<s_2, s_3>>$$
$$t_a = <<s_a, s_c>>, \quad t_b = <<s_b, s_c>>$$

are sequences of state transitions. Here, we can see that $T_{a1} = \{t_1, t_2\}$ and $T_{a2} = \{t_a, t_b\}$ covers all paths to s_3 and s_c, respectively. Assume that the following equivalence classes have already been proved.

$$\{\{<a, t_1>, <b, t_a>\}, \{<a, t_2>, <b, t_b>\},$$
$$\{<a, t_3>, <b, t_c>\}\}$$

Then, all the paths to s_3 and s_c have corresponding paths where a and b are equivalent. From Rule 3, $<a, s_3> \equiv <b, s_c>$ becomes true.

The last rule is for FSMDs which have loops such as M_1 and M_2 in Fig. 10. The equivalence of such FSMDs cannot be proved only with Rule $1 \sim 3$, since previous results of the computation are used in each iteration. In such the case, the next rule can be applied.

Rule 4. *Let $s_1 \in S_1$ and $s_2 \in S_2$ denote one of the states in different loops, respectively. Let $T_{11} = \{t_i^1 \mid 1 \le i \le l,\ t_i^1 \in T_1\}$ and $T_{12} = \{t_i^2 \mid 1 \le i \le m,\ t_i^2 \in T_2\}$ denote sets of sequences of state transitions reaching s_1 and s_2 which cover all paths from the inside of the loops to s_1 and s_2, respectively. Also, Let $T_{13} = \{t_i^3 \mid 1 \le i \le n,\ t_i^3 \in T_1\}$ and $T_{14} = \{t_i^4 \mid 1 \le i \le k,\ t_i^4 \in T_2\}$ denote sets of sequences of state transitions reaching s_1 and s_2 which cover all paths from the outside of the loops to s_1 and s_2, respectively.*

Then, $<e_1, s_1> \equiv <e_2, s_2>$, where $e_1 \in E_1$, $e_2 \in E_2$, becomes true when the following two conditions are satisfied.

- *The next equation is true*

$$(n = k) \wedge \bigwedge_{i=1}^{n} (<e_1, t_i^3> \equiv <e_2, t_i^4>)$$

- *Under an assumption that $<e_1, s_1> \equiv <e_2, s_2>$ is true, the next equation becomes true with Rule $1 \sim 3$.*

$$(l = m) \wedge \bigwedge_{i=1}^{l} (<e_1, t_i^1> \equiv <e_2, t_i^2>)$$

Proof. We prove this rule with unrolling the loops as shown in Fig. 11, and the following induction. Let ${}^i s_1, {}^i s_2$ denote ith s_1 and s_2 after the loops are unrolled, respectively. Also, let ${}^i t_j^1, {}^i t_j^2$ denote ith t_j^1 and t_j^2. The first condition in Rule 4 is the basic case which proves $<e_1, {}^1 s_1> \equiv <e_2, {}^1 s_2>$ with Rule 3. The second condition is the inductive step which

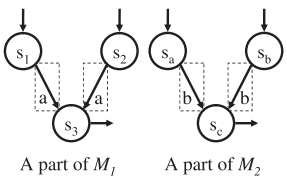

Fig. 9 Example 3.

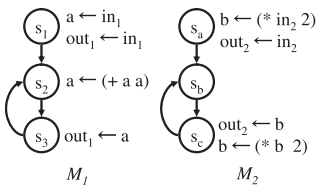

Fig. 10 Example 4.

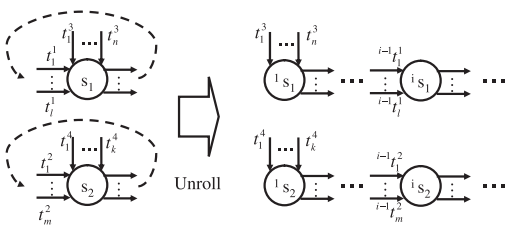

Fig. 11 Loop unrolling for the proof of Rule 4.

proves $<e_1, {}^{i+1}s_1> \equiv <e_2, {}^{i+1}s_2>$ with Rule 3 under the assumption $<e_1, {}^{i}s_1> \equiv <e_2, {}^{i}s_2>$. Therefore, the next equation is inductively proved.

$$\bigwedge_{i=1}^{\infty} <e_1, {}^{i}s_1> \equiv <e_2, {}^{i}s_2>$$

This is equivalent to $<e_1, s_1> \equiv <e_2, s_2>$. □

As written in the proof, in the inductive step, equivalences are propagated from the assumption to apply Rule 3 for the state s_1 and s_2. This propagation is performed by applying Rule 1 ∼ 3 multiple times. Here, we only have to finally prove the assumption, and do not have to consider how to apply Rule 1 ∼ 3. Then, only the final step where Rule 3 is applied is defined in the rule. Therefore, the first states of t_i^1 and t_i^2 in the rule can be arbitrary states in the insides of the loops.

With this rule, the equivalence of M_1 and M_2 in Fig. 10 can be proved. $s_1, s_2, s_3 \in S_1$ and $s_a, s_b, s_c \in S_2$ are states. $in_1 \in I_1$ and $in_2 \in I_2$ are corresponding inputs. $out_1 \in O_1$ and $out_2 \in O_2$ are corresponding outputs. $a \in V_1$ and $b \in V_2$ are variables. $2 \in K_1 \cap K_2$ is a constant. $+, * \in F$ are addition and multiplication, where $(+\ x\ x) = (*\ x\ 2)$ for $x \in V$ has already been proved to be equivalent.

Initial equivalence classes are

$$\{\{<in_1, s_1> \equiv <in_2, s_a>\}\}$$

The goal is to prove $<out_1, s_2> \equiv <out_2, s_b>$.

First, from Rule 2, the following equations are proved.

(1) $<out_1, <<s_1, s_2>> \equiv <out_2, <<s_a, s_b>>$
(2) $<a, <<s_1, s_2>, <s_2, s_3>> \equiv <b, <<s_a, s_b>, <s_b, s_c>>$

Next, we apply Rule 4 to prove (3) $<a, s_3> \equiv <b, s_c>$.

Basic Case

(2) satisfies the first condition of Rule 4.

Inductive Step

(3) is assumed. Then, from Rule 2, the next equation is proved

(4) $<a, <<s_3, s_2>, <s_2, s_3>> \equiv <b, <<s_c, s_b>, <s_b, s_c>>$

With (4), the second condition of Rule 4 is satisfied.

Then, (3) is proved to be true. Next, from Rule 2 and (4), the following equation is proved.

(5) $<out_1, <<s_3, s_2>> \equiv <out_2, <<s_c, s_b>>$

Finally, from Rule 3, (1), and (5), the following equation is proved.

(6) $<out_1, s_2> \equiv <out_2, s_b>$

Now, the equivalence of outputs is proved and M_1 and M_2 are proved to be equivalent.

This rule can handle nested loop by recursively applying the rule to inner loops under the assumption of the inductive step.

4.6.2 Algorithm to Apply the Rules

In this section, we explain an algorithm to apply the proposed four rules to designs. Also, we discuss about designs which can be verified by this rule-based method.

Figure 12 shows a simple algorithm to apply the four rules proposed in Sect. 4.6.1. This algorithm consists of the following four steps:

1. Equivalences of inputs given by users are added to the equivalence classes.
2. Rule 1 is applied to all transitions.
3. Rule 2 and 3 are applied to each state
4. Rule 4 is applied to prove the equivalence of loops.

Rule 4 is recursively applied to handle nested loops. Assumptions of the second step of Rule 4 are incrementally made from outer loops to inner loops, and those assumptions are guaranteed from that of the inner loops to that of the outer loops. L is a parameter which defines the maximum length of sequences of state transitions when applying Rule 2. If L becomes larger, the number of target sequences of state transitions becomes large, and the complexity becomes higher. Then, L should be started from 1 and incremented until the equivalence is proved.

The termination of this algorithm is proved as follows. There are two infinite loops by *while*(*true*) in the algorithm shown in Fig. 12. Both of them break when no more equivalence classes are generated in the loops. The number of equivalence classes is finite, since the number of equivalence candidates is finite. Also, the recursive call of sub2 eventually stops, since the levels of multiple loops are finite. Therefore, this algorithm must terminate.

Here, we have to mention that our rule-based verification method (including the four rules and the algorithm) is not complete. It just says "equivalent" in particular cases when the rules can prove equivalences. In other cases, our method just says "indeterminable". However, our method is fast, and when a result is equivalent, the result is guaranteed to be true.

By the proposed four rules to propagate equivalences, as mentioned in the explanation of Rule 3, FSMDs which have same structures of conditional branches can be verified. Note that lengths of transitions can be different between two FSMDs under verification unless there are no branches in the transitions. Also, outsides and insides of corresponding loops in two FSMDs must be equivalent, respectively. Therefore, the method can verify the designs before and after scheduling, retiming, or some optimizations like common sub-expression elimination, unless such optimizations are applied beyond loops.

5. Experimental Results

We applied the verification flows shown in Sect. 3.1 to realistic examples.

```
// Main procedure
bool main(){
  ZS zs =empty; //A set of equivalence classes for states
  zs += Init(); // Add initial equivalence class
  //For each assignment in each state, apply Rule 1
  foreach((s, a) in (S1*A1)+(S2*A2))
    zs += Rule1(s, a);
  /* Check that the output equivalence is included
     in the equivalence class which have already been proved */
  if(CheckResult(zs))
    return true;
  // Apply Rule2 and Rule 3 with the sub-procedure sub1
  zs += sub1(zs);
  if(CheckResult(zs)) return true;
  zs += sub2(zs); // Apply Rule4 with the sub-procedure sub2
  if(CheckResult(zs, null)) return true;
  return false;
}
/* Check equivalences with Rule 2 and 3 and
   return newly proved equivalence classes */
ZS sub1(ZS zs){
  while(true){
    //A local set of equivalence classes for states
    ZS zs_local = empty;
    foreach(<s1, s2> in S1*S2){
      /* Collect sequences of state transitions reach to s1 and
         whose length is equal to or less than L */
      t1 = {t| t in T1, size(t)<=L, last_state(t)==s1};
      /* Collect sequences of state transitions reach to s2 and
         whose length is equal to or less than L */
      t2 = {t| t in T2, size(t)<=L, last_state(t)==s2};
      // For each pair of expressions
      foreach(<e1, e2> in E1*E2){
        /* A set of equivalence class for
           sequences of state transitions */
        ZT zt = empty;
        /* For each pair of sequence of state transitions
           apply Rule 2 */
        foreach(<ta, tb> in t1*t2)
          if(Rule2(zs+zs_local, e1, ta, e2, tb))
            zt += {{<e1, ta>, <e2, tb>}};
        // If some elements are added to zt, apply Rule3
        if(zt != empty)
          if(Rule3(zt, e1, s1, e2, s2))
            zs_local += {{<e1, s1>, <e2, s2>}}
      }
    }
    // If no equivalences of states are newly proved
    if(zs_local == empty)
      return zs_local;
  }
}
/* Check equivalences with Rule 4 and
   return newly proved equivalence classes */
ZS sub2(ZS zs, Loop current_loop){
  /* Get the set of loops which are one level inner from
     current_loop when it is not null.
     Get the most outside loops when current_loop is null */
  Loops inner_loops = getInnerLoops(current_loop);
  while(true){
    //A local set of equivalence classes for states
    ZS zs_local = empty;
    foreach(inner_loop in inner_loops){
      foreach(<s1, s2> in inner_loop){
        //Apply Rule 2 to prove the basic case
        t1 = {t| t in T1, size(t)<=L, last_state(t)==s1,
                 first_state(t) is not in inner_loop};
        t2 = {t| t in T2, size(t)<=L, last_state(t)==s2,
                 first_state(t) is not in inner_loop};
        foreach(<e1, e2> in E1*E2){
          ZT zt = empty;
          foreach(<ta, tb> in t1*t2)
            if(Rule2(zs+zs_local, e1, ta, e2, tb))
              zt += {{<e1, t1>, <e2, t2>}};
          if(zt == empty) continue;
          //Check the equivalence of states when entering the loop
          if(!Rule3(zt, e1, s1, e2, s2)) continue;
          // Check the inductive step
          ZS zs_assumed = {{(e1, s1), (e2, s2)}};
          // Apply Rule2 and Rule 3 with the assumption
          ZS zs_guaranteed = sub1(zs+zs_assume);
          /* If the assumption is not proved, apply Rule 4
             incrementally to prove equivalences about the inner
             loops */
          if(zs_assumed is not a subset of zs_guaranteed)
            zs_guaranteed += sub2(zs_guaranteed+zs_assume,
            inner_loop);
          // If the assumption is proved, Rule4 is satisfied
          if(zs_assumed is a subset of zs_guaranteed)
            zs_local += zs_guaranteed;
        }
      }
    }
    // If no equivalences of states are newly proved
    if(zs_local == empty) return zs_local;
  }
}
```

Fig. 12 An algorithm to apply rules.

5.1 Tool Implementations

To generate a controller for a given datapath (explained in Sect. 3.2), we used an on-line NISC complier demo [14]. Separation of controllers and datapaths of the designs, and translation from RTL description into FSMD were done by hand.

Also, we implemented two tools to check the equivalence of FSMDs with C and C++. One is a symbolic simulator in which the method described in Sect. 4.5 is implemented. The other is a rule-based verifier in which the method explained in Sect. 4.6 is implemented. Both tools run on a PC with a 3 GHz processor (dual core) and 1 GB memory.

5.2 Examples

We used three examples, DCT (Discrete Cosine Transform), IDCT (Inverse Discrete Cosine Transform), and Ellip (Elliptical Filter). All examples are originally written in C, and the details are in Table 1.

We applied optimizations and high-level synthesis to those examples by hand. Therefore, there are three versions for each example such as, (1) original design, (2) design after behavioral optimization, (3) design after high-level synthesis. The optimizations are removal of temporal variables, refinement of operations, and others. All synthesized designs use the same datapath which is about 1000 lines in Verilog, and those designs are pipelined. In all examples, variable names are not corresponding. The numbers of states, inputs, outputs, variables in translated FSMDs are also shown in Table 1.

5.3 Verification Results

For each example, it took about 10 seconds to synthesize the controller by NISC compiler.

The verification time of equivalence checking between the FSMDs are shown in Table 2. All results were equivalent, and they were correct. Since all examples include loops, symbolic simulation could be applied only after un-

Table 1 Information of examples.

Example	Version	LOC in C	Control structure	Num. of states	Num. of inputs	Num. of outputs	Num. of variables
DCT	(1)	54	Trilaminar structure of 8-iterations	14	64	64	68
	(2)	52		13	64	64	66
	(3)	-		11	64	64	66
IDCT	(1)	134	Two large 8-iterations	98	64	64	83
	(2)	120		95	64	64	83
	(3)	-		90	64	64	75
Ellip	(1)	74	One large infinite loop	36	1	1	37
	(2)	67		33	1	1	35
	(3)	-		20	1	1	31

Table 2 Verification time of rule-based equivalence propagation.

Example	Target	Symbolic simulation	Symbolic simulation with loop unrolling	Rule-based verification
DCT	(1) vs (2)	-	< 1 s	2.4 s
	(2) vs (3)	-	< 1 s	3.1 s
IDCT	(1) vs (2)	-	> 24 h	24.3 s
	(2) vs (3)	-	> 24 h	30.8 s
Ellip	(1) vs (2)	-	< 1 s	7.5 s
	(2) vs (3)	-	< 1 s	7.9 s

rolling the loops. Also, since Ellip examples include infinite loops, they are unrolled for only 1-iteration. Then the results are not complete. Rule-based verification could be successfully applied to all examples directly. In these experiments, we set the parameter L of the algorithm in Fig. 12 to 1. Also, since the methods explained in Sects. 4.1 and 4.4 were not implemented, equivalences of symbolic expressions were checked only with simple replacements of equivalent expressions in equivalent classes.

The results show that symbolic simulation could verify the DCT and Ellip examples faster, since there are no conditional branches. Rule-based verification checks all candidates of equivalence exhaustively. Moreover, when an equivalence of a candidate is proved, all other candidates are checked again, since their equivalences can be proved with the information of the newly proved equivalence. Then, each candidate of equivalence may be checked multiple times. However, since symbolic simulation checks the equivalences of expressions at states from the initial state only once for each execution path, it is basically faster to verify designs without many conditional branches than rule-based verification. Also, if there are loops in target designs, we have to apply Rule 4 for "*number of states in loops* × *number of expressions*" times in the worst case. Then rule-based verification becomes much slower. However, even there are a lot of conditional branches in the IDCT examples, rule-based verification could verify them within relatively short times which are not so much different from the other examples. Symbolic simulation could not verify them within 24 hours. This is because the complexity is square to the number of conditional branches in rule-based verification, and exponential in symbolic simulation.

From these experimental results, we could confirm the following facts.

- The overall proposed method can successfully applied to real designs.
- Verification time of rule-based verification is not strongly affected from numbers of conditional branches
- Rule-based verification can directly verify designs which include loops without unrolling the loops.

6. Conclusion

In this paper, we proposed a word-level equivalence checking method in bit-level accuracy with synthesizing two designs with the same datapath. We also proposed a new word-level rule-based comparison method, and the experimental results show that our method is fast and it can verify some designs which cannot be verified by symbolic simulation. Since our method is a rule-based method, we can extend the range of verifiable designs by introducing additional rules.

We are planning the following future works.

- **Utilizing potentially equivalent states or sequences of state transitions**
  In the current implementation, the explorations of the four rules are done exhaustively. If we can determine potentially equivalent states or sequences of state transitions by some sort of simulations, then the search domains can be reduced drastically.
- **Internal equivalent point**
  Utilizing internal equivalent points can improve the verification speed. In [15], a cut-point insertion method for equivalence checking between designs before and after high-level synthesis is proposed. We are also planning to utilize such kind of techniques to optimize the search of equivalent candidates.

References

[1] D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao, SpecC: Specification Language and Methodology, Kluwer Academic Publisher, 2000.
[2] "SystemC," http://www.systemc.org/
[3] E. Clarke, D. Kroening, and K. Yorav, "Behavioral consistency of c and verilog programs using bounded model checking," Proc. Design Automation Conference, pp.368–371, June 2003.
[4] H. Saito, T. Ogawa, T. Sakunkonchak, M. Fujita, and T. Nanya, "An equivalence checking methodology for hardware oriented c-based specification," Proc. International Workshop on High Level Design Varidation and Test, pp.139–144, Oct. 2002.
[5] T. Matsumoto, H. Saito, and M. Fujita, "Equivalence checking of c programs by locally performing symbolic simulation on dependence graphs," Proc. International Symposium on Quality Electronic De-

sign, pp.370–375, March 2006.
[6] C. Karfa, M. Mandal, D. Sarkar, S.R. Pentakota, and C. Reade, "A formal verification method of scheduling in high-level synthesis," Proc. International Symposium on Quality Electronic Design, pp.110–115, March 2006.
[7] D. Gajski, N. Dutt, A. Wu, and S. Lin, High-Level Synthesis: Introduction to Chip and System Design, Kluwer Academic Publisher, 1992.
[8] A.W. Group, "RTL semantics draft specification," http://www.eda.org/alc-cwg/cwg-open.pdf
[9] P. Schaumont, S. Shukla, and I. Verbauwhede, "Design with race-free hardware semantics," Proc. Conference on Design, Automation and Test in Europe, pp.571–576, April 2006.
[10] M. Fujita, "Equivalence checking between behavioral and RTL descriptions with virtual controllers and datapaths," ACM Trans. Des. Autom. Electron. Syst., vol.10, no.4, pp.610–626, Oct. 2005.
[11] M. Reshadi and D. Gajski, "A cycle-accurate compilation algorithm for custom pipelined datapaths," Proc. International Conference on Hardware/Software Codesign and System Synthesis, pp.21–26, Sept. 2005.
[12] "CVC3," http://www.cs.nyu.edu/acsys/cvc3/
[13] J. Burch and D. Dill, "Automated verification of pipelined microprocessor control," Proc. International Conference on Computer-Aided Verification, pp.68–80, June 1994.
[14] "University of California, Irvine. NISC technology," http://www.cecs.uci.edu/~nisc/
[15] X. Feng and A.J. Hu, "Early cutpoint insertion for high-level software vs. RTL formal combinational equivalence verification," Proc. Desing Automation Conference, pp.1063–1068, July 2006.

Tasuku Nishihara received the B.S. and M.S. degrees in electronic engineering from University of Tokyo, Tokyo, Japan, in 2005 and 2007, respectively. He is currently a Ph.D. student in the Department of Electronics Engineering, University of Tokyo. His research interests include formal verification and design analysis for high-level designs of digital systems.

Takeshi Matsumoto received the B.S., M.S., and Ph.D in electronic engineering from University of Tokyo, Tokyo, Japan, in 2003, 2005, and 2008, respectively. He has been a memeber of VLSI Design and Education Center in University of Tokyo since 2008. His research interests include computer-aided design and formal verification, especially for high-level designs of digital systems.

Masahiro Fujita received the B.S. degree in electrical engineering in 1980, and the M.S. and Ph.D. degrees in information engineering from the University of Tokyo, Tokyo, Japan, in 1982 and 1985, respectively. From 1985 to 1993, he was a Research Scientist with Fujitsu Laboratories, Kawasaki, Japan. From 1994 to 1999, he was the Director of the Advanced Computer-Aided Design Research Group, Fujitsu Laboratories of America, Sunnyvale, CA. He is currently a Professor in VLSI Design and Education Center, University of Tokyo, Tokyo, Japan. He has been on program committees for many conferences dealing with digital design and is an Associate Editor of Formal Methods on Systems Design. His primary research interest is in the computer-aided design of digital systems. Dr. Fujita received the Sakai Award from the Information Processing Society of Japan in 1984.