Formal Verification of Designs with Complex Control by Symbolic Simulation

Gerd Ritter, Hans Eveking, and Holger Hinrichsen

Dept. of Electrical and Computer Engineering
Darmstadt University of Technology, D-64283 Darmstadt, Germany
{ritter,eveking,hinrichsen}@rs.tu-darmstadt.de

Abstract. A new approach for the automatic equivalence checking of behavioral or structural descriptions of designs with complex control is presented. The verification tool combines symbolic simulation with a hierarchy of equivalence checking methods, including decision-diagram based techniques, with increasing accuracy in order to optimize overall verification time without giving false negatives. The equivalence checker is able to cope with different numbers of control steps and different implementational details in the two descriptions to be compared.

1 Introduction

Verifying the correctness of designs with complex control is crucial in most areas of hardware design in order to avoid substantial financial losses. Detecting a bug late in the design cycle can block important design resources and deteriorate the time-to-market. Validating a design with high-confidence and finding bugs as early as possible is therefore important for chip design.
"Classical" simulation with test-vectors is incomplete since only a non-exhaustive set of cases can be tested and costly as well in the simulation itself as in generating and checking the tests. Formal hardware verification covers all cases completely, and gives therefore a reliable positive confirmation if the design is correct.
The formal verification technique presented in this paper uses *symbolic* simulation. Employing symbolic values makes the complete verification of *all* cases possible. One symbolically simulated path corresponds in general to a large number of "classical" simulation runs. During symbolic simulation, relationships between symbolic terms, e.g., the equivalence of two terms are detected and recorded. A given verification goal like the equivalence of the contents of relevant registers, is checked at the end of every symbolic path. If it is not demonstrated, more time-consuming but also more accurate procedures including decision diagram based techniques are used to derive undetected relationships.
Currently, the approach is used to check the *computational equivalence* of two descriptions but it is also applicable to the verification of properties which is planned for future research. Two descriptions are computationally equivalent if both produce the same final values on the same initial values relative to a set

L. Pierre and T. Kropf (Eds.): CHARME'99, LNCS 1703, pp. 234–250, 1999.

of relevant variables. For instance, the two descriptions in Fig. 1 are computationally equivalent with respect to the final value of the relevant variable r. The equivalence checker simulates symbolically all possible paths. False paths

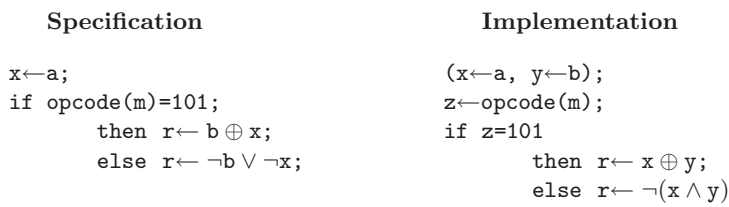

Fig. 1. Example of two computationally equivalent descriptions

are avoided by making only consistent decisions at branches in the description. A case split is performed if a condition is reached which cannot be decided but depends on the initial register and memory values, e.g., `opcode(m)=101` in Fig. 1. The symbolic simulation of the specification and of the implementation is executed in parallel. The example in Fig. 1 requires, therefore, the symbolic simulation of two paths. Note that both symbolic paths represent an important number of "classical" symbolic runs.

Each symbolically executed assignment establishes an equivalence between the destination variable on the left and the term on the right side of an assignment. Additional equivalences between terms are detected during simulation. Equivalent terms are collected in equivalence classes. During the path search, only relationships between terms that are fast to detect or are often crucial for checking the verification goal are considered on the fly. Some functions remain uninterpreted while others are more or less interpreted to detect equivalences of terms, which is considered by unifying the corresponding equivalence classes.

Having reached the end of both descriptions with consistent decisions, a complete path is found and the verification goal is checked for this path, e.g., if both produce the same final values of r. This check is trivial for the *then*-branches in Fig. 1 since the equivalence of b⊕x and x⊕y is detected on the fly.

Using only a selection of function properties for equivalence detection which are fast to compute during the path search, we may fail to prove the equivalence of two terms at the end of a path, e.g., the equivalence of ¬b ∨ ¬x and ¬(x ∧ y) in the *else*-branches of Fig. 1 (application of De Morgan's Law on *bit-vectors*). In these cases the equivalence of the final values of r is checked using *decision diagrams* as described in section 5. If this fails, it is verified whether a false path is reached, since conditions may be decided inconsistently during the path search due to the limited equivalence detection. If the decisions are sound, the counterexample for debugging is reported. Relevant details about the symbolic simulation run can be provided since all information is available on every path in contrast to formula based verification. Our automatic verification process does not require insight of the designer into the verification process.

Some related work is reviewed in section 2. Section 3 shows the preparation of the initial data structure. Section 4 describes the path search itself, the symbolic simulation in the proper sense, and gives an overview of the algorithm. Section 5 presents the more powerful, but less time-efficient algorithms that are used if the verification goal can not be demonstrated at the end of a path. The use of *vectors* of OBDD's as canonizer is discussed and compared to other approaches. Experimental results are presented in section 6. Finally, section 7 gives a conclusion and directions for future work.

2 Related Work

Several approaches have been proposed for formal verification of designs with complex control. Theorem provers were used to verify the control logic of processors requiring extensive user guidance from experts which distinguishes the approach from our automated technique. Prominent examples are the verification of the FM9001 microprocessor [4] using Nqthm, of the Motorola CAP processor [5] using ACL2 and the verification of the AAMP5 processor using PVS [23]. [18] proposed an interesting approach to decompose the verification of pipelined processors in sub-proofs. However, the need for user guidance especially for less regular designs remains.
The idea of symbolic state-space representation has already been applied by [3,12] for equivalence checking or by [10] for traversing automata for model checking. Their methods use decision diagrams for state-space representation, and are therefore sensitive to graph explosion. [24] developed an encoding technique for uninterpreted symbols, i.e., the logic of uninterpreted functions with equality is supported, and they abstracted functional units by memory models. The complexity of their simulation increases exponentially if the memory models are addressed by data of other memory models, therefore the processor they verified contains no data-memory or branching.
[11] proposed an approach to generate a logic formula that is sufficient to verify a pipelined system against its sequential specification. This approach has also been extended to dual-issue, super-scalar architectures [19,9,25] and with some limitations to out-of-order execution by using incremental flushing [22,20]. SVC (the Stanford Validity Checker) [1,2,19] was used to automatically verify the formulas. SVC is a proof tool requiring for each theory to add that functions are canonizable and algebraically solvable, because every expression must have a unique representation. If a design is transformed by using theories, that are not fast to canonize/solve or that are not supported, SVC can fail to prove equivalence. Verifying bit-vector arithmetic [2], which is often required to prove equivalence in control logic design, is fast in SVC if the expressions can be canonized without slicing them into single bits, otherwise computation time can increase exponentially. Our approach does not canonize expressions in general. Only if the verification goal can not be demonstrated at the end of a path, formulas are constructed *using previously collected information* and are checked for equivalence using *vectors* of OBDD's. The efficiency of vectors of OBDD's in

our application area is compared with SVC and *BMD's in section 5. Another problem of building formulas first and verifying them afterwards is the possible term-size explosion which may occur if the implementation is given at the structural rt-level or even gate-level (see section 4 and 6). In addition, the debugging information given by a counter-example is restricted to an expression in the initial register values.
Symbolic simulation of executable formal specifications as described in [21] uses ACL2 without requiring expert interaction. Related is the work in [13], where pre-specified microcode sequences of the JEM1 microprocessor are simulated symbolically using PVS. Expressions generated during simulation are simplified on the fly. Multiple "classical" simulation runs are also collapsed but the intention of [21] is completely different since concrete instruction sequences at the machine instruction level are simulated symbolically. Therefore a *fast* simulation on *some* indeterminate data is possible for debugging a specification. Our approach checks equivalence for *every* possible program, e.g., not only some data is indeterminate but also the control flow. Indeterminate branches would lead in [21] to an exponentially grow of the output to the user. Furthermore, insufficient simplifications on the fly can result in unnecessary case splits or/and term-size explosion.

3 The Internal Data Structure

Our equivalence checker compares two *acyclic* descriptions at the rt-level. For many cyclic designs, e.g., pipelined machines the verification problem can also be reduced to the equivalence check of acyclic sequences, which is shown for some examples in section 6.
The inherent timing structure of the initial descriptions is expressed explicitly by indexing the register names. An indexed register name is called a *RegVal*. A new *RegVal* with an incremented index is introduced after each assignment to a register. An additional upper index s or i distinguishes the *RegVals* of specification and implementation. Only the initial *RegVals* as anchors are identical in specification and implementation, since the equivalence of the two descriptions is tested with regard to arbitrary but identical initial register values. Fig. 2 gives a simple example written in our experimental rt-level language LLS [15]. Parenthesis enclose synchronous parallel transfers. The sequential composition operator ";" separates consecutive transfers. "Fictive" assignments (italic in Fig. 2) have to be generated, if a register is assigned in only one branch of an *if-then-else* clause in order to guarantee that on each possible path the sequence of indexing is complete. Checking computational equivalence consists of verifying that the final *RegVals*, e.g., adr_2 or pc_1 are equivalent to the according final *RegVals* in the other description. The introduction of *RegVals* makes all information about the sequential or parallel execution of assignments redundant which is, therefore, removed afterwards. Finally every distinct term and subterm is replaced for technical reasons by an arbitrary chosen distinct variable. A new variable is introduced for each term where the function type or at least one argument is

```
adr←pc;                                  adr1 ←pc;
ir←mem(adr);                             ir1 ←mem(adr1);
if ir[0:5]=000111                        if ir1[0:5]=000111
then   (pc←pc+1, adr←ir[6:15]);          then   (pc1 ←pc+1, adr2 ←ir1[6:15]);
       mi←mem(adr);                              mi1 ←mem(adr2);
       ac←ac+mi;                                 ac1 ←ac+mi1;
else   pc←pc+2;                          else   pc1 ←pc+2;
                                                adr2 ←adr1;
                                                mi1 ←mi;
                                                ac1 ←ac;
```

Fig. 2. Numbering registers after each new assignment

distinct, e.g., $\texttt{pc}_1\texttt{+2}$ and $\texttt{pc}_2\texttt{+2}$ are distinguished.
Formula based techniques like SVC do not use distinct *RegVals*, because they represent the changes of registers in the term-hierarchy implicitly. Expressing the timing structure explicitly has several advantages. Term size explosion is avoided, because terms can be expressed by intermediate *RegVals*, see also section 4. We do not loose information about intermediate relationships by rewriting or canonizing so that arbitrary additional techniques can be used at the end of a path to establish the verification goal. In addition, support of debugging is improved.

4 Symbolic Simulation

4.1 Identifying Valid Paths

One subgoal of our symbolic simulation method is the detection of equivalent terms.

Definition 1 (Equivalence of terms). *Two terms or RegVals are equivalent* $\equiv_{term}$*, if under the decisions* $C_0, ..., C_n$ *taken preliminary on the path, their values are identical for all initial RegVals. The operator* $\downarrow$ *denotes that each case-split, leading to one of the decisions* $C_0, ..., C_n$*, constrains the set of possible initial RegVals.*

$$term_1 \equiv_{term} term_2 \Leftrightarrow \forall RegVal_{initial} : (term_1 \equiv term_2) \downarrow (C_0 \wedge C_1 ... \wedge C_n)$$

Equivalent terms are detected along valid paths, and collected in *equivalence classes*. We write $term_1 \equiv_{sim} term_2$ if two terms are in the same equivalence class established during simulation. If $term_1 \equiv_{sim} term_2$ then $term_1 \equiv_{term} term_2$. Initially, each *RegVal* and each term gets its own equivalence class. Equivalence classes are unified in the following cases:

- two terms are identified to be equivalent;
- a condition is decided; if this condition is

  - a test for equality $a = b$, the equivalence class of both sides are unified *only if the condition is asserted*,
  - otherwise (e.g., $a < b$ or a status-flag) the equivalence class of the condition is unified with the equivalence class of the constant 1 or 0 if the condition is asserted or denied;
- after every assignment. Practically, this union-operation is significantly simpler because the equivalence class of the *RegVal* on the left-hand side of the assignment was not modified previously.

Equivalence classes permit to keep also track about unequivalences of terms:

Definition 2 (Unequivalence of terms). *Two terms or RegVals are unequivalent* $\not\equiv_{term}$*, if under the decisions* $C_0, ..., C_n$ *taken preliminary on the path their values are never identical for arbitrary initial RegVals:*

$$term_1 \not\equiv_{term} term_2 \Leftrightarrow \neg\exists RegVal_{initial} : (term_1 \equiv term_2) \downarrow (C_0 \wedge C_1 ... \wedge C_n)$$

We write $term_1 \not\equiv_{sim} term_2$ if two terms are identified to be $\not\equiv_{term}$ during simulation. Equivalence classes containing $\not\equiv_{sim}$ terms are unequivalent. This is the case

- if they contain different constants;
- if a condition with a test for equality (e.g., $a = b$) is decided to be false.

Implementing a path enumeration requires a decision algorithm each time an *if-then-else* is reached. Identifying *CondBits* in the conditions accelerates this decision procedure. *CondBits* replace

(a) tests for equality of bit-vectors, i.e., terms or *RegVals* (e.g., $\mathtt{r}_3^s = \mathtt{x}_2^s$ + $\mathtt{y}_1^s$);
(b) all terms with Boolean result (e.g., $\mathtt{r}_3^s$ < $\mathtt{x}_2^s$) except the connectives below;
(c) single-bit registers (e.g., status-flags).

After the replacement, the conditions of the *if-then-else*-structures contain only propositional connectives (NOT, AND, IOR, XOR) and *CondBits*. Because identical comparisons might be done multiple times in one path, this approach avoids multiple evaluation of a condition by assigning one of three values (UNDEFINED, TRUE, FALSE) to the *CondBits*. If a *CondBit* appears the first time in a path, its value is UNDEFINED. Therefore, its condition is checked by comparing the equivalence classes of two terms or *RegVals*: In case (a), we have to check the terms on the left-hand and right-hand side, whereas in cases (b) and (c) the equivalence class of the term is compared to the equivalence class of the constant 1. There are three possible results:

i. The two terms to be compared are in the same equivalence class. Then the *CondBit* is asserted or TRUE *in this path* for arbitrary initial register values;
ii. The equivalence classes of the terms have been decided preliminary to be unequivalent or contain different constants. The *CondBit* is always denied or FALSE;

iii. Otherwise the *CondBit* may be true or false, depending on the initial register and memory values. Both cases have to be examined in a case split. Denying/Asserting a *CondBit* leads to a decided unequivalence/union-operation.

Fig. 3 (a) gives an example of the symbolic simulation of one path during the equivalence check of the example in Fig. 1. The members of the equivalence classes after every simulation step are given in Fig. 3 (b). Initially all terms and *RegVals* are in distinct equivalence classes. S1 is simulated first. When symbolic

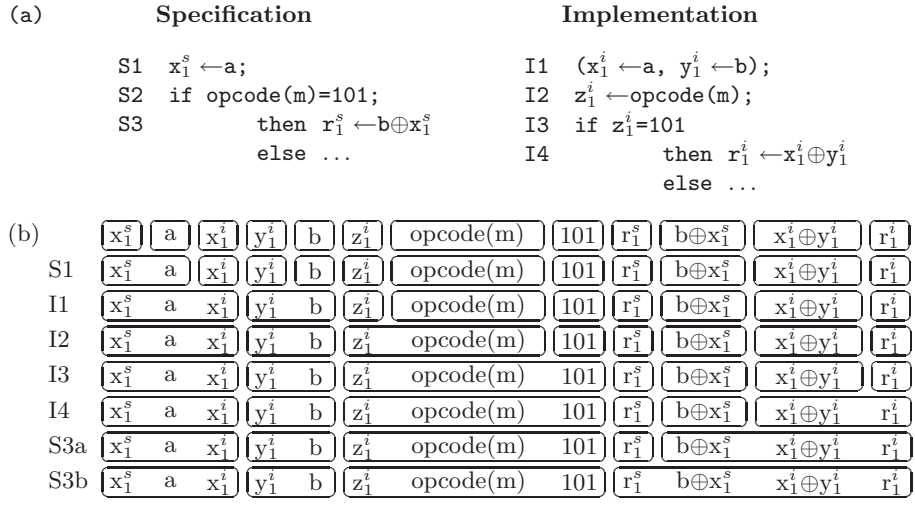

Fig. 3. Example of a simulation run

simulation reaches S2, the condition of S2 depends on the initial *RegVals* (case iii) and the simulation is blocked. Paths are searched simultaneously in specification and implementation. After the simulation of I1 and I2, I3 requires also a case split. Decisions in the normally more complex implementation have priority in order to facilitate a parallel progress. Therefore, a case split on the condition in I3 is performed. Only the case with the condition asserted is sketched in Fig. 3, where the equivalence classes of z_1^i and the constant 101 are then unified and I4 is simulated. The condition of S2 is now decidable in the given context since both sides of the condition are in the same *EqvClass* (case i), i.e., no additional case split is required. First the equivalence of $b \oplus x_1^s$ and $x_1^i \oplus y_1^i$ is detected (S3a) and then the assignment to r_1^s is considered (S3b). Finally r_1^s and r_1^i are in the same equivalence class, therefore, computational equivalence is satisfied at the end of this path. If they were in different equivalence classes, equivalence would be denied. Note that simultaneous progress in implementation and specification avoids simulating S1 again for the denied case.

4.2 Identifying Equivalent Terms

Ideally, all $\equiv_{term}$ equivalent terms and *RegVals* are in the same equivalence class, but it is too time consuming to search for all possible equivalences on the fly. Therefore, no congruence closure is computed *during the path search*, i.e., building eventually incomplete equivalence classes is accepted in favor of a fast path search. If congruence closure or undetected equivalences are required to check the verification goal, the algorithms described in section 5 are used.
In order to speed up the path search, the following simplifications are made with respect to completeness of equivalence detection:

- Some functions, e.g., user-defined functions are always treated as uninterpreted.
- Only fast to check or "crucial" properties of interpreted functions are considered. Some examples are:
  - If a bit or a bit-vector of a term or a *RegVal* is selected which is in an equivalence class with a constant, the (constant) result is computed (e.g., from IR$\equiv_{sim}$ 011 follows IR[1]$\equiv_{sim}$ 1). If at least one argument of a Boolean function is $\equiv_{sim}$ to 1 or 0 then it is checked whether the function is also $\equiv_{sim}$ to one of these constants.
  - Functions representing multiplexers, i.e., structures where N control signals select one of $M = 2^N$ data-words, have to be interpreted. A transformation into an adequate *if-then-else*-structure is feasible, but blows up the descriptions. Note that, therefore, multiplexers can lead to term-size explosion, if the overall formula is build in advance and verified afterwards (e.g., if a big ROM is used). This can be avoided in symbolic simulation by using intermediate carriers and evaluating expressions on the fly.
  - Symmetric functions are equivalent, if every argument has an equivalent counter-part (e.g., $(a \equiv d) \wedge (b \equiv c) \Rightarrow (a + b) \equiv (c + d)$). Note that preliminary sorting of the arguments can not always tackle this problem because different terms can be assigned to *RegVals*.
- The transformation steps done during preprocessing preserve the timing structure. In general, equivalence of the arguments of two terms is already obvious, when the second term is found on the path. Therefore, it is sufficient to check only at the first occurrence of a term whether it is equivalent to terms previously found.
- In most cases the equivalence of terms can be decided by simply testing if the arguments are $\equiv_{sim}$ or $\not\equiv_{sim}$ which avoids the expansion of the arguments.
- Equivalence checking for a term is stopped after the first union operation, since all equivalent terms are (ideally) already in the same equivalence class.

This procedure fails in two cases:

- Equivalence cannot be detected by the incomplete function interpretation.
- A decision about the relationship of the initial *RegVals* is done *after* two terms are found on the path and equivalence of the terms is given only considering this decision.

The last situation occurs especially in the case of operations to memories. Similar to [11,1], two array operations are used to model memory access: `read(mem,adr)` reads a value at the address `adr` of memory `mem` while `store(mem,adr,val)` stores `val` corresponding without changing the rest of the memory. In the example of Fig. 4, the order of the `read` and the `store` operations is reversed in the implementation. Thus, `val` is forwarded if the addresses are identical. The prob-

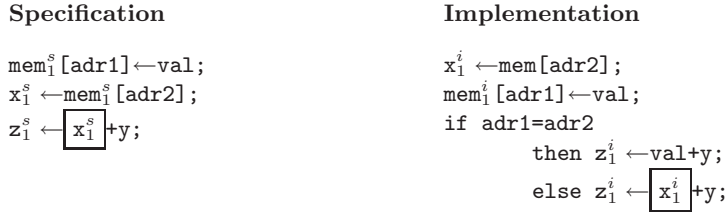

Fig. 4. Forwarding example

lem is to detect, that in the opposite case the final values of `x` are identical, which is only obvious *after* the case split (setting adr1 $\not\equiv_{sim}$ adr2) and not already after the assignments to `x`. The example indicates, that it is important to check `read` and `store`-terms whenever the equivalence classes of the related addresses are modified. Note that our address comparison uses only the information of the *EqvClasses* and does not evaluate boolean expressions as in [24].

4.3 Overview of the Algorithm

Lines 3 to 10 in Fig. 5 summarize the path search. For every case split due to a condtition `to_decide`, first the denied case is examined (line 9) while the asserted case is stored in `rem_cases` (line 8). Initially `rem_cases` contains the whole specification and implementation with a dummy-condition (line 1). Note that only those parts of the descriptions, that are not simulated yet in this path, are examined after case splits, i.e., `remain(act_case`$_{spec/impl}$`)` (line 8). Lines 12 to 22 describe the case where computational equivalence is not reported at the end of a path (line 11), and are explained in the next section in full detail.

5 Examining Differences of the Descriptions

5.1 Overview

In the first hierarchy-level of the checking algorithm, arbitrary function properties can be considered in order to detect term-equivalence. Adding the check of function properties during the path search is a trade-off: the accuracy increases, therefore less false negatives to be checked afterwards occur. But the additional checks can be time-consuming since every time a term is found a check is done

which may actually be necessary only in few cases or not at all.
According to *Algorithm Equivalence Check*, if the verification goal is not given in a path (line 11), then the first step is to consider additional function properties which are less often necessary or more time consuming to check.
If then the verification goal is not yet reported for all pairs of final *RegVals* an attempt is made to decide the equivalence. Formulas are built considering knowledge about path-dependent equivalence/unequivalence of intervenient terms which are sufficient for the equivalence of the final *RegVals* (line 14). A

```
INPUT spec, impl;
 1. rem_cases := {(dummy_cond,spec,impl)};
 2. WHILE rem_cases ≠ ∅ DO
 3.    act_case := pop(rem_cases);
 4.    assert(act_case_to_decide);
 5.    REPEAT
 6.        to_decide := simulate_parallel(act_case);
 7.        IF to_decide THEN
 8.            push(to_decide,remain(act_case_spec),remain(act_case_impl))
                    rem_cases;
 9.            deny(to_decide);
10.    UNTIL to_decide not found;
11.    IF ∃i : R^spec_ifinal ≠_sim R^impl_ifinal THEN
12.         check_additional_properties;
13.        IF ∃i : R^spec_ifinal ≠_sim R^impl_ifinal THEN
14.            ∀i : (R^spec_ifinal ≠_sim R^impl_ifinal) : LET F_j ⇒ R^spec_ifinal ≡_sim R^impl_ifinal ;
15.            IF ∃j : F_j ≡ TRUE THEN
16.                  mark_new_relations;
17.                  return_in_path;
18.            ELSIF ∃k : inconsistent(decision_k)
19.                  mark_new_relations;
20.                  return_to_wrong_decision;
21.            ELSE report_debug_information;
22.                  RETURN(FALSE);
23. ENDWHILE;
24. RETURN(TRUE);
```

Fig. 5. Algorithm Equivalence Checking

pre-check follows, which applies some logic minimization techniques and which checks whether a formula was built previously and stored in a hash-table. Before hashing, all *RegVals* and cut-points (see below) in the formulas are replaced in order of their appearance in the formula by auxiliary variables T1, T2,...,Tn, because the same formula may appear with only different *RegVals* or cut-points with regard to a previously computed formula. New formulas are checked using binary decision diagrams. This is the first time a canonical form is built.
If none of the formulas is satisfiable, all decided *CondBits*, i.e., conditions for which a case-split was done, are checked in order of their appearance to search

for a contradictory decision due to the incomplete equivalence detection on the fly. Using the information about the equivalence classes again facilitates considerably building the required formulas.
If at least one formula is valid (line 15) or if a contradictory decision has been detected (line 18), the path is backtracked and the relationship is marked so that it is checked during further path search on the fly. This is done since the probability is high, that also in other paths the more time consuming algorithms are invoked unnecessarily again due to this relationship.
Otherwise the descriptions are not equivalent and the counterexample is reported for debugging (line 21). A complete error trace for debugging can be generated since all information about the symbolic simulation run on this path is available. For example, it turned out that a report is helpful which summarizes different microprogram-steps or the sequence of instructions carried through the pipeline registers. Note that if formulas were canonized only a counterexample in the initial *RegVals* would be available. Simulation-information can also be useful if the descriptions are equivalent. For instance, a report of never taken branches *in one if-clause* indicates redundancy which may not be detected by logic minimizers.

5.2 Building Logic Formulas without Sequential Content

For each unsatisfied goal (equivalence of two *RegVals*), a formula is built. The knowledge about equivalence/unequivalence of terms, which is stored in the equivalence classes, is used in order to obtain formulas which are easy to check. It is possible to obtain formulas in terms of the initial *RegVals* without term-size explosion by backward-substitution because a specific path is chosen. In many cases, however, less complex formulas can be derived by using intermediate cut points already identified to be equivalent in specification and implementation during symbolic simulation. A greedy algorithm guides the insertion of cut-points in our prototype version. Validating the formulas may be infeasible if these cut-points are misplaced or hide necessary function properties. Therefore, a failed check is repeated without cut-points.

5.3 Checking Formulas by Means of Decision Diagrams

A *Multiple-Domain Decision Diagram Package* (TUDD-package) [16,17] developed at TU Darmstadt with an extension for vectors of OBDD's [6] is used to prove the formulas. Another possibility is to use word-level decision diagrams like *BMD's [7,8]. In practical examples of control logic, bit-selection functions are used frequently, either explicitly, e.g., R[13:16], or implicitly, e.g., storing the result of an addition in a register without carry. Using *BMD's, terms are represented by one single *BMD. Bit-selection, therefore, requires one or two *modulo*-operations which are worst-case exponentially with *BMD.
Bit-selection is quasi for free, if terms are expressed as *vectors* of OBDD's, where each graph represents one bit. Bit-selection can then be done by simply skipping the irrelevant bits, i.e., the corresponding OBDD's and by continuing the computation with the remaining OBDD's. Checking equivalence just consists of

comparing each bit-pair of the vectors.
The initial *RegVals* and the cut-points are represented by a vector of decision diagrams, where each of the diagrams represents exactly one bit of the *RegVal* or cut-point. There is no fixed assignment of vectors of decision diagrams to initial *RegVals*/cut-points, but association is done dynamically after a formula is built. Decision diagrams with a fixed variable ordering (interleaved variables) are built during pre-processing since reordering would be too time consuming.
All formerly applied algorithms are (fairly) independent of the bit-vector length. Results obtained during symbolic simulation are used to simplify formulas before OBDD-vector construction. But even without simplification even large bit-vectors can be handled by OBDD-vectors in acceptable computation time. In [2] the results of SVC on five bit-vector arithmetic verification examples are compared to the results of the *BMD package from Bryant and Chen and also to Laurent Arditi's *BMD [2] implementation which has special support for bit-vector and Boolean expressions. We verified these examples also with OBDD-vectors. Tab. 1 summarizes the results. All our measurements are on a Sun Ultra II with 300 MHz. Various orderings of the variables for our *BMD-measurements are used. The line DM contains the verification results for a bit-wise application of De Morgan's law to two bit-vectors a and b, i.e., $\overline{a_0 \wedge b_0}\#...\#\overline{a_n \wedge b_n} \equiv (\overline{a_0} \vee \overline{b_0})\#...\#(\overline{a_n} \vee \overline{b_n})$, and the ADD-example is the verification of a ripple-carry-adder. Note that the input is also one *word* for the two last examples and not a vector of inputs (otherwise *BMD-verification is of course fast since no slicing or modulo operation is required). The inputs may represent some intermediate cut-points for which, e.g., the *BMD is already computed.
Obviously, *BMD-verification suffers from the modulo-operations in the examples. According to [2], the results of example 1 to 4 are independent of the bit-vector length for SVC, but the verification times with OBDD-vectors are also acceptable even for large bit-vectors. These times can be reduced especially for small bit-vectors by optimizing our formula parsing. In example 5, SVC ends up slicing the vector and thus the execution time depends on the number of bits and shows, therefore, a significant increase, whereas the computation time for OBDD-vectors increases only slightly. The increase in *this* example may be eliminated in a future version of SVC [2], but the general problem is that slicing a vector has to be avoided in SVC. This can be seen for the examples DM and ADD, where verification is only practical with OBDD-vectors.
Note that functions that are worst-case exponentially with OBDD's or have no representation, e.g., multiplication, are only problematic in the rare cases where special properties of the functions are necessary to show equivalence. Normally, the terms are replaced by cut-points during the formula-construction since we use information from the simulation-run.

	SVC[1]		*BMD Bryant/Chen[1]		*BMD Arditi[1]		OBDD-vector TUDD				
	200MHz Pentium		200MHz Pentium		300MHz UltraSparc 30		300MHz Sun Ultra II				
Bits	16	32	16	32	16	32	16	32	64	128	256
1	N/A	0.002	N/A	N/A	N/A	0.04	0.14	0.27	0.38	0.68	1.38
2	N/A	0.002	N/A	N/A	N/A	1.10	0.13	0.20	0.25	0.44	0.93
3	0.002	0.002	265.0	>500	0.07	0.18	0.21	0.32	0.51	0.95	1.95
4	0.002	0.002	26.4	>500	0.72	8.79	0.24	0.40	0.71	1.53	4.38
5	0.111	0.520	22.7	>500	0.39	3.78	0.14	0.21	0.31	0.57	1.15
	Measured at TUDD		Measured with TUDD *BMD-package								
Bits	16	32	16	32	64		16	32	64	128	256
DM	>5min		>5min				0.12	0.22	0.28	0.48	1.03
ADD	-[2]		5.19	37.2	282.7		0.21	0.31	0.48	0.98	1.90

[1] Measurements reported in [2].
[2] 2 Bit: 1.01s; 4 Bit: 9.47s; 5 Bit 44.69s; Verification with more than 5 Bit was not feasible with the current version of SVC.

Table 1. Comparison of SVC, *BMD and OBDD-Vectors. Times are in seconds.

6 Experimental Results

DLX-Processor Descriptions

Two implementations of a subset of the DLX processor [14] with 5-pipeline-stages have been verified, the first from [18], initially verified in [11], and a second one designed at TU Darmstadt. The latter contains more structural elements, e.g., the multiplexers and corresponding control lines required for forwarding are given.

For both descriptions, acyclic sequences are generated by using the flushing approach of [11]; i.e., execution of the inner body of the pipeline loop followed by the flushing of the pipeline is compared to the flushing of the pipeline followed by one serial execution. Different to [11] (see also [9]), our flushing schema guarantees, that one instruction is fetched *and* executed in the first sequence, because otherwise it has to be communicated between specification and implementation if a instruction has to be executed in the sequential processor or not (e.g., due to a load interlock in the implementation). [9] describes this as keeping implementation and specification in sync; using their flushing approach with communication reduces our number of paths to check and verification time, too. Verification is done automatically, only the (simple) correct flushing schema, guaranteeing that one instruction is fetched and executed, has to be provided by the user. In addition, some paths are collapsed by a simple annotation that can be used also for other examples. Forwarding the arguments to the ALU is obviously redundant, if the EX-stage contains a NO_OP or a branch. The annotation expresses, that in these cases the next value of these arguments can be set to a distinct unknown value. The verification remains *complete*, because the equiva-

lence classes of the *RegVals* to check would always be different, if one of these final *RegVals* depends on such a *distinct* unknown value. Note that verification has been done for both cases also without this annotation, but with ≈ 90% more paths to check. Two errors introduced by the conversion of the data-format used

Version	paths	aver. time per path	total time
DLX from [18]	310,312	12.6 ms	1h 5min 13s
DLX with multiplexers	259,221	19.5 ms	1h 24min 14s

Table 2. Verification results for DLX-implementations

by [18] and several bugs in our hand crafted design have been detected automatically by the equivalence checker. Verification results of the correct designs are given in Tab. 2. Measurements are on a Sun Ultra II with 300 MHz. Note that the more detailed and structural description of the second design does not blow up verification time: the average time per path increases acceptable, but the number of paths remains nearly the same (even decreases slightly due to a minor different realization of the WB-stage).
Verifying the DLX-examples does not require the more complex algorithms, especially the decision diagrams, because with exception of the multiplexers in the second design, the pipelined implementation can be derived from a sequential specification using a small set of simple transformations (in [15] a formally correct automatic synthesis approach for pipelined architectures using such a set is presented). Verifying examples like the DLX is not the main intention of our approach since the capabilities of the equivalence checker are only partly used, but it demonstrates that also control logic with a complex branching can be verified by symbolic simulation.

Microprogram-Control with and without Cycle Equivalence

In this example, two behavioral descriptions of a simple architecture with microprogram control are compared to a structural implementation. In both behavioral descriptions, the microprogram control is performed by simple assignments and no information about the control of the datapath-operations, e.g., multiplexer-control, is given. The structural description of the machine compromises an ALU, 7 registers, a RAM, and a microprogram-ROM. All multiplexers and control lines required are included. The two behavioral descriptions differ in the number of cycles for execution of one instruction:

- The first is cycle-equivalent to the structural description; i.e., all register-values are equivalent in every step. Generating the finite sequences consists of simply comparing the loop-bodies describing one micro-program step.
- The second is less complex than the first and more intuitive for the designer. It contains an instruction fork in the decode phase. No cycle equivalence is

given, therefore, the sequences to be compared are the complete executions of one instruction. The only annotation of the user is the constant value of the microprogram counter, that indicates the completion of one instruction.

The ROM is expressed as one multiplexer with constant inputs. In this example, the read/write-schema used also in SVC would not work, since the ROM has constant values on all memory-places. The ROM-accesses and the other multiplexers would lead to term-size explosion if they are interpreted as functions (canonizing!) as well as if they are considered as *if-then-else*-structures, since symbolic simulation goes over several cycles in this example. Results are given in Tab. 3.

Example	paths*	ext. checks	false paths	time
with cycle equivalence	291	56	39	24.53s
different number of cycles	123	41	16	19.58s

* including false paths

Table 3. Verification of microprogram-controller

Measurements are on a Sun Ultra II with 300 MHz, verification times include the construction of decision diagrams. The third column indicates how often the extended checks of section 5 are used either to show equivalence or to detect an inconsistent decision, i.e., one of the false path reported in the fourth column is reached. The in principle more difficult verification without cycle equivalence requires less paths since the decisions in the behavioral description determines the path in the structural description. Note that again no insight into the automatic verification process is required.

7 Conclusion and Future Work

A new approach for equivalence checking of designs with complex control using symbolic simulation is presented. The number of possible paths to simulate can be handled even for complex examples since symbolic values are used. All indeterminate branches, that depend on initial register values, are considered by case splits to permit a complete verification for an arbitrary control flow.
Our equivalence detection on the fly is not complete to permit a fast simulation. If the verification goal is not given at the end of a path, additional and more powerful algorithms including decision-diagram based techniques are used to review the results of the simulation run.
The approach is flexible to integrate various equivalence detection algorithms which are applied either finally or during simulation on the fly. There are no special requirements like canonizability to integrate new theories since we keep track about equivalences of terms by assembling them in equivalence classes. Therefore, all information about the simulation run is available at the end of a path. This is also useful for debugging: simulation *"is a natural way engineers think"*.

First experimental results demonstrate the applicability to complex control logic verification problems. The equivalence checker supports different number of control steps in specification and implementation. Structural descriptions with implementational details can be compared with their behavioral specification. By using intermediate carriers, term-size explosion is avoided which can occur in formula-based techniques when implementational details are added.
The approach has so far only been used to check the computational equivalence of two descriptions. An application to designs, where relationships between intermediate values or temporal properties have to be verified, is planned in future work. Another topic is to parallelize the verification on multiple workstations in order to reduce the overall computational time.

Acknowledgement

The authors would like to thank the anonymous reviewers for helpful comments.

References

1. C. W. Barrett, D. L. Dill, and J. R. Levitt: Validity checking for combinations of theories with equality. In Proc. FMCAD'96, Springer LNCS 1166, 1996. 236, 242
2. C. W. Barrett, D. L. Dill, and J. R. Levitt: A decision procedure for bit-vector arithmetic. In Proc. DAC'98, 1998. 236, 245, 246
3. D. L. Beatty and R. E. Bryant: Formally verifying a microprocessor using a simulation methodology. In Proc. DAC'94, 1994. 236
4. B. Brock, W. A. Hunt, and M. Kaufmann: The FM9001 microprocessor proof. Technical Report 86, Computational Logic Inc., 1994. 236
5. B. Brock, M. Kaufmann, and J. S. Moore: ACL2 theorems about commercial microprocessors. In Proc. FMCAD'96, Springer LNCS 1166, 1996. 236
6. R. E. Bryant: Graph-based algorithms for Boolean function manipulation. In IEEE Trans. on Computers, Vol. C-35, No. 8, pages 677-691, 1986. 244
7. R. E. Bryant and Y.-A. Chen: Verification of arithmetic functions with binary moment diagrams. Technical Report CMU-CS-94-160, Carnegie Mellon University, 1994. 244
8. R. E. Bryant and Y.-A. Chen: Verification of arithmetic circuits with binary moment diagrams. In Proc. DAC'95, 1995. 244
9. J. R. Burch: Techniques for verifying superscalar microprocessors. In Proc. DAC'96, 1996. 236, 246
10. J. R. Burch, E. Clarke, K. McMillan, and D. Dill: Sequential circuit verification using symbolic model checking. In Proc. DAC'90, 1990. 236
11. J. R. Burch and D. L. Dill: Automatic verification of pipelined microprocessor control. In Proc. CAV'94. Springer LNCS 818, 1994. 236, 242, 246
12. O. Coudert, C. Berthet, and J.-C. Madre: Verification of synchronous sequential machines based on symbolic execution. In Proc. Automatic Verification Methods for Finite State Systems, Springer LNCS 407, 1989. 236
13. D. A. Greve: Symbolic simulation of the JEM1 microprocessor. In Proc. FMCAD'98, Springer LNCS 1522, 1998. 237

14. J. L. Hennessy, D. A. Patterson: Computer architecture: a quantitative approach. Morgan Kaufman, CA, second edition, 1996. 246
15. H. Hinrichsen, H. Eveking, and G. Ritter: Formal synthesis for pipeline design. In Proc. DMTCS+CATS'99, Auckland, 1999. 237, 247
16. S. Höreth: Implementation of a multiple-domain decision diagram package. In Proc. CHARME'97, pp. 185-202, 1997. 244
17. S. Höreth: Hybrid Graph Manipulation Package Demo. URL : http://www.rs.e-technik.tu-darmstadt.de/∼sth/demo.html, Darmstadt 1998. 244
18. R. Hosabettu, M. Srivas, and G. Gopalakrishnan: Decomposing the proof of correctness of pipelined microprocessors. In Proc. CAV'98, Springer LNCS 1427, 1998. 236, 246, 247
19. R. B. Jones, D. L. Dill, and J. R. Burch: Efficient validity checking for processor verification. In Proc. ICCAD'95, November 1995. 236
20. R. B. Jones, J. U. Skakkebæk, and D. L. Dill: Reducing manual abstraction in formal verification of out-of-order execution. In Proc. FMCAD'98, Springer LNCS 1522, 1998. 236
21. J. S. Moore: Symbolic simulation: an ACL2 approach. In Proc. FMCAD'98, Springer LNCS 1522, 1998. 237
22. J. U. Skakkebæk, R. B. Jones, and D. L. Dill: Formal verification of out-of-order execution using incremental flushing. In Proc. CAV'98, Springer LNCS 1427, 1998. 236
23. M. Srivas and S. P. Miller: Applying formal verification to a commercial microprocessor. In Computer Hardware Description Language, August 1995. 236
24. M. N. Velev and R. E. Bryant: Bit-level abstraction in the verification of pipelined microprocessors by correspondence checking. In FMCAD'98, Springer LNCS 1522, 1998. 236, 242
25. P. J. Windley and J. R. Burch: Mechanically checking a lemma used in an automatic verification tool. In Proc. FMCAD'96, November 1996. 236