EQUIVALENCE CHECKING IN C-BASED SYSTEM-LEVEL DESIGN BY SEQUENTIALIZING CONCURRENT BEHAVIORS

Thanyapat Sakunkonchak[1], Takeshi Matsumoto[2], Hiroshi Saito[3], Satoshi Komatsu[1], Masahiro Fujita[1]
[1] VLSI Design and Education Center, University of Tokyo, Tokyo 113–0032 Japan
[2] Dept. of Electronics Engineering, University of Tokyo, Tokyo 113–8656 Japan
[3] Dept. of Computer Hardware, University of Aizu, Aizu–Wakamatsu, 965–8580 Japan
email: {thong,matsumoto}@cad.t.u-tokyo.ac.jp, hiroshis@u-aizu.ac.jp, komatsu@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp

ABSTRACT
In system-level designs, since many incremental refinements are applied to the designs, equivalence checking between each refinement should be applied. However, proving whether two concurrent designs are equivalent is a difficult task, not to mention that the concurrent design itself can be error-prone. In this paper, we propose an equivalence checking method for C-based descriptions of system-level designs by sequentializing the concurrent behaviors. Before sequentializing concurrent behaviors, we need to check that the design must not contain neither deadlock nor race condition. After the sequentialization, equivalence checking is performed by symbolic simulation. To show that our methodology can be applied to practical designs, we experiment with some SpecC designs developed by University of California Irvine (UCI). The results show that the proposed method is promising. Although the size of some designs are large, with heuristic search for concurrency and synchronization, the size of designs are reduced accordingly and hence we can perform equivalence checking with the sequentialized ones.

KEY WORDS
VLSI design and CAD, system-level design, equivalence checking, sequentialization, concurrent behaviors, synchronization verification

1 Introduction

Building reliable hardware and software systems is a major challenge, and the system design process is made even more difficult by continual increases in design complexity. As semiconductor technology advances, entire systems can be realized within single LSIs as Systems-on-a-Chip (SoCs). At the same time, competitive pressures have been pushing system designers to shorten the design cycle and reduce time-to-market. To cope with these competing demands, new design paradigms that offer more levels of abstraction have been proposed. Designing an SoC is a process of both hardware and software development, and requires a uniform design flow from specification to implementation. Recently, there has been a lot of interest in approaches built around the C/C++ programming languages. Since C/C++ are commonly used in software development, C-based SoC design (using languages like SystemC or SpecC) is a promising approach to cover both hardware and software design with a single design/specification language.

In C-based design as shown in Figure 1 starting from specifications, designers usually describe the design at the very beginning stage with the sequential descriptions (e.g. C). At this step, there is no clear distinction between hardware and software parts in the descriptions. As the refinement of the design continues, communications, HW/SW partitioning and concurrency are inserted. System-level design languages such as SystemC or SpecC can be utilized.

At each modification made with the design (e.g. each design refinement), we may need to ensure that the refined one is functionally equivalent to the current one. There are some related researches on equivalence checking of the two systems [1, 2, 3, 4, 5]. The traditional equivalence checking method constructs finite-state models and proves whether they are equivalent. Although there are many commercial equivalence checking tools available, most of them still suffers from the state explosion problem. In [5], equivalence checking of the two different C descriptions was proposed. The method avoids the state-space explosion problem by suppressing the similar descriptions appear in both designs. However, the method is applicable to the sequential C descriptions. It cannot directly apply to the design descriptions that contain concurrency. This paper has the following contributions:

- Our goal is to check the equivalence of two concurrent descriptions, in this case they are written in SpecC language. We propose an algorithm to sequentialize the concurrent SpecC descriptions and then prove whether two sequentialized descriptions are equivalent. Verification process can be done in four steps: 1) check for any deadlock, 2) check for any race condition, 3) if no error from previous steps, sequentialized descriptions, and 4) equivalence checking the two descriptions.

- We use a method for synchronization verification in [6] for synchronization and deadlock checking. Once synchronization is checked, synchronization points can be defined. Between each synchronization point,

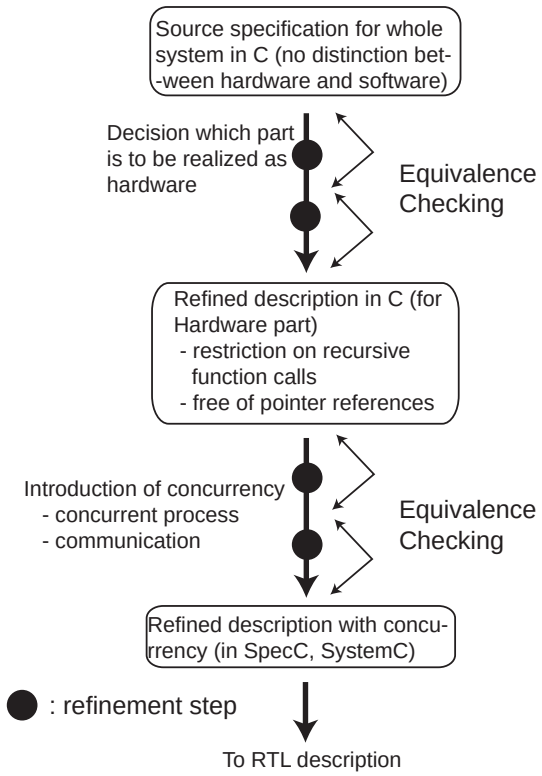

Figure 1. C-based design flow

we check if there is any race condition of any variable across different concurrent processes.

- Generate the sequentialized descriptions. Then, we can compare the two generated sequential descriptions with the method presented in [5].

This paper is organized as follows. Some related researches are briefly reviewed in Section 2. Section 3 describes our proposed method for sequentialized concurrent design and prove the generated sequential ones whether they are equivalent. Some preliminary results are presented in Section 4. And finally we conclude and discuss about the further direction of this research in Section 5.

2 Related Work

2.1 Synchronization Verification

Software model checking poses its own challenges, as software tends to be less structured than hardware. In the software verification domain, predicate abstraction [7, 8, 9] is widely applied to reduce the state-space by mapping an infinite state-space program to an abstract program of Boolean type while preserving the behaviors and control constructs of the original. Counterexample-Guided Abstraction Refinement (CEGAR) [10] is a method to automate the abstraction refinement process. More specifically, starting with a coarse level of abstraction, the given property is verified. A counterexample is given when the property does not hold. If this counterexample turns out to be

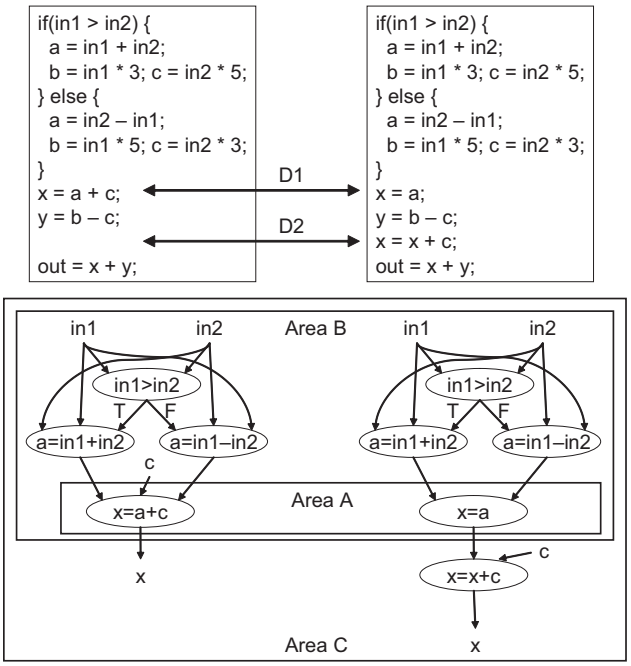

Figure 2. Equivalence checking example

spurious, the previous abstract programs are then refined to a finer level of abstraction. The verification process is continued until there is no error found or there is no solution for the given property.

In system-level design languages such as SpecC, extra constructs are added to C in order to describe the characteristics of hardware. These extra constructs support description of parallel behaviors, pipelined behaviors, finite state machines, and operations on arbitrary-length bit-vectors. System-level models are organized as a collection of cooperating processes running in parallel. In order to keep all processes executing as the designer intended, proper scheduling of statement execution in all processes (known as *synchronization*) is necessary. *Deadlock* is an error that is caused by synchronization failure.

In [6], an approach to synchronization verification of systems described in SpecC was proposed. SpecC contains the *waitfor* and *notify/wait* constructs to schedule and synchronize concurrent processes. The *waitfor* statement delays a process by a specific number of time units and therefore introduces a timing constraint. In this approach timing constraints are expressed with equalities/inequalities which can be solved by integer linear programming (ILP) tools.

2.2 Equivalence Checking based on Symbolic Simulation

In [5], the verification method for equivalence checking of hardware-oriented C descriptions is proposed. The method is based on symbolic simulation, which symbolically executes descriptions with generating Equivalence Classes (EqvClasses).

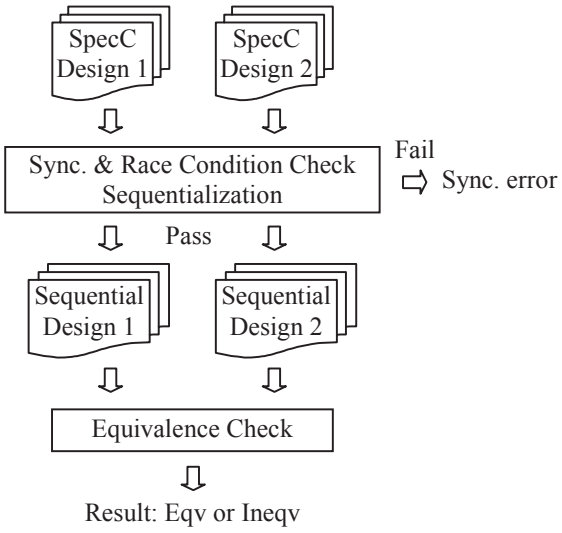

Figure 3. Verification flow

Algorithm 1 EquivalenceChecking(*SC*1, *SC*2)

declare
1: *SC*1, *SC*2: SpecC descriptions
2: *Seq*1, *Seq*2: Sequential descriptions generated from Sequentialization()
3: *eqv*: result of equivalence checking
begin
4: /* Sequentialize *SC*1 and *SC*2 */
5: *Seq*1 := Sequentialization(*SC*1)
6: *Seq*2 := Sequentialization(*SC*2)
7: /* Equivalence checking by symbolic simulation */
8: *eqv* := EC_Symbolic(*Seq*1, *Seq*2)
9: **if** *eqv* **then**
10: **return** ("*SC*1 and *SC*2 are equivalent")
11: **else**
12: **exit** ("*SC*1 and *SC*2 are NOT equivalent")
13: **end if**
end

One of the most important features of the method is utilizing differences between descriptions under verification. Symbolic simulation is applied only to the differences and the portions that are related to the differences through dependences. Therefore, the verification can process large descriptions when the differences are small. Figure 2 shows how the method in [5] works. Assume that variables *in*1 and *in*2 are the primary inputs of the program, and variable *out* is the primary output. Both descriptions are similar except the two differences denoted by *D*1 and *D*2. Below descriptions, the dependence graphs are shown.

We start by verifying the first difference *D*1. The verification area according to *D*1 is *A*, and its local input variables are *a* and *c*, and its local output variable is the variable *x*. Equivalence of *x* cannot be proved in both descriptions because, with this scope area *A*, the local input variables are unknown. Thus, in this case, we decide to backwardly extend the area from *A*. Then, the extended verification area becomes area *B*, and the verification is carried out again. In this case, the local input variables are *in*1, *in*2, and *c*, and the local output variable is *x*. Since the equivalence of *x* cannot be proved after the verification within the area *B*, we decide to forwardly extend the area from *x* and obtain the area *C*. After the verification with area *C*, we can prove the equivalence of *x* in both descriptions. Verification for the difference *D*2 is not carried out since it is already included when verified *D*1. Then, as all the differences are verified, the two descriptions are functionally equivalent.

3 The Proposed Verification Method

The entire verification flow of the proposed method is shown in Figure 3 and the corresponding algorithms are shown in Algorithm 1 and 2. Prior to sequentializing both of the given designs, they are checked whether they do not have any deadlock or race condition. Note that at this point, we assume that the designs contained deadlock or race condition bugs are the mistakes of the designs. If these errors were found, the verification is aborted with the result "Error". We left the task to correct these errors to the designers. Otherwise, when there is no such a bug, the two sequential designs are generated, and equivalence checking is applied.

3.1 Synchronization & Race Condition Checking and Sequentialization

To sequentialize the design descriptions, the following conditions must be satisfied.

- **No deadlock.** Every *wait* statement is always executed with at least one corresponding *notify* statement being eventually executed.
- **No race condition.** There is no possibility for any shared variable to be accessed at the same time.

These are the necessary conditions required to generate the sequential design that is equivalent to the original one. If a given design has a deadlock, its execution can halt somewhere in the design, while the execution of the sequential design should never halt. In this case, behaviors of the two designs are not equivalent for any sequentialization. On second condition, if there exists any global variable which is local to parallel behaviors, access to this variable (known as read/write or write/write access) at the same time can cause the design to perform different functionalities.

Synchronization Check

As described in previous section and according to the verification approach in [6], we can provide some constraints or properties to prove some synchronization errors or to validate any execution order. This can be referred as solving

Algorithm 2 Sequentialization(*SC*)

```
declare
  1:  error: a Boolean variable
  2:  Sync: a set of synchronization point
  3:  SET_par: a set contains heuristic depth of par
begin
  4:  /* Heuristically search for par */
  5:  SET_par := HeuristicSearch(SC)
  6:  foreach par in SET_par (start from the deepest one)
      do
  7:    /* Check if there is any deadlock */
  8:    (error, Sync) := SynchronizationVerification(SC)
  9:    if error then
 10:      exit ("There is a deadlock")
 11:    end if
 12:    foreach synchronization point in Sync do
 13:      /* Check if there is any race condition */
 14:      error := RaceConditionDetect(SC)
 15:      if error then
 16:        exit ("There is a race condition")
 17:      end if
 18:    end for
 19:  end for
 20:  return (SequentialGen(SC))
end
```

constraints of Integer Linear Programming (ILP) problems. Consider an example of two parallel behaviors written in SpecC *par*{*A.main();B.main();*} where *A* has total number of *m* statements and contains a *notify* statement and *B* has total number of *n* statements and contains a *wait* statement. The following constraints are generated. Note that $T(A_x)$ represents a time where statement *x* in behavior *A* is executed.

- $T(A_{start}) = T(B_{start}),\ T(A_{end}) = T(B_{end})$
- $T(A_{start}) < T(A_{st\_1}) < \ldots < T(A_{st\_m}) < T(A_{end})$
- $T(B_{start}) < T(B_{st\_1}) < \ldots < T(B_{st\_n}) < T(B_{end})$
- $T(A_{notify}) < T(B_{wait})$

According to the SpecC Language Reference Manual [11], the first constraints represent that the concurrent behaviors under *par* start/end at the same time. The second and third constraints show that execution order in each behavior is in sequential fashion. With the fact that *wait* statement will hold any process until *notify* statement is executed, the last constraint can be generated.

Given a formulae of constraints of a SpecC design as mentioned above, deadlock occurs when the formulae is infeasible.

Race Condition Check

Before introducing race condition check, definitions of *basic block* and *synchronization point* are introduced.

- **Basic block (BB):** A series of statements that do not include conditional branches nor synchronization point.
- **Synchronization point (SP):** A pair of *notify* and *wait* statements of the same *event* is considered as a synchronization point.

Now, assume that there are two basic blocks, *BB1* and *BB2*, where $T(BB1_{starttime}) < T(BB1_{endtime})$ and $T(BB2_{starttime}) < T(BB2_{endtime})$ are the timing constraints for each block. Together with these constraints, we can check the race condition between *BB1* and *BB2* by checking the following pair of properties.

$$\textbf{Prop1}:\ T(BB1_{starttime}) < T(BB2_{endtime})$$
$$\textbf{Prop2}:\ T(BB1_{endtime}) > T(BB2_{starttime})$$

We will not consider the condition when both **Prop1** and **Prop2** are unsatisfied at the same time since it is obvious that this condition cannot be occurred. If **Prop1** is satisfied and **Prop2** is not, *BB*1 is proved to be executed prior to *BB*2 (*BB*1 $\rightarrow$ *BB*2). In contrast, if **Prop1** is not satisfied and **Prop2** is, we can say that *BB*1 is proved to be executed after *BB*2 (*BB*2 $\rightarrow$ *BB*1). If both of the properties are satisfied, *BB1* and *BB2* can be interleaved and it is possible for race condition to occur. All variables in *BB1* and *BB2* must be checked for data dependency. If, for any variable, there is data dependence (read/write or write/write) between *BB1* and *BB2*, then there is a race condition. Otherwise, *BB1* and *BB2* are race condition free and we can sequentialize in either (*BB*1 $\rightarrow$ *BB*2) or (*BB*2 $\rightarrow$ *BB*1) order.

According to Algorithm 2, the sequentialization process is described as follows.

- Since SpecC supports design hierarchy and concurrency, we store all *par* statements in a set SET_{par} and, for each *par*, we mark its depth as we explore the design to lower hierarchy. The depth of a *par* statement is the depth from the top *par* statement of the hierarchy. If a *par* statement is not nested, the depth is 0.
- For each *par* statement, the deepest depth of SET_{par} is taken from the set and sequentialized by the following procedure. Let *Bhvr*1 and *Bhvr*2, for example, denote the two behaviors under the *par* statement to be sequentialized.
  - With the method for race condition check as described above, find all pairs of basic blocks *BB*1 in *Bhvr*1 and *BB*2 in *Bhvr*2 and put them into a set BB_{pair}.
  - For each pair in BB_{pair}, check **Prop1** and **Prob2**. If **Prob1** is true and **Prob2** is not, *BB1* and *BB2* can be sequentialized directly as *BB1* before *BB2*, and vice versa. However, if both properties are satisfied and there is no data dependence, *BB1* and *BB2* can be sequentialized in any order. Either *BB1* before *BB2* or *BB2* before *BB1*.

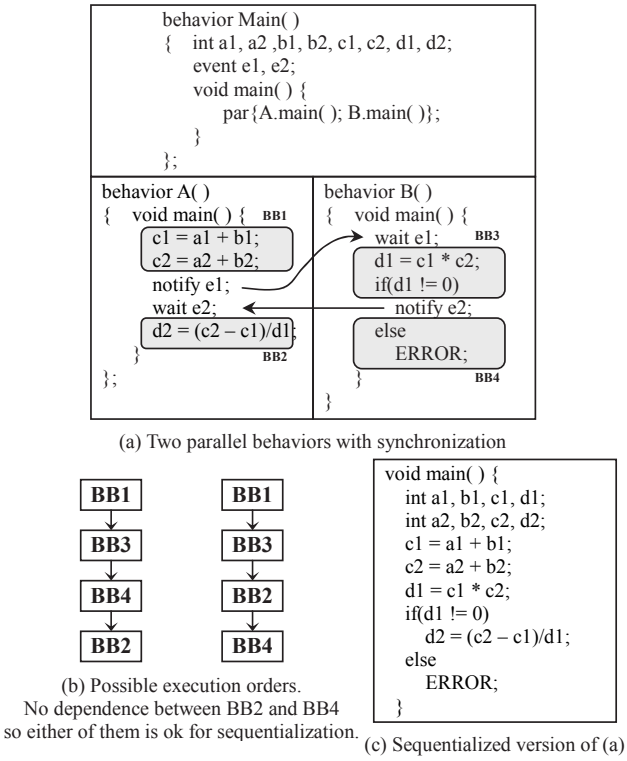

Figure 4. Sequentialization example 1

– Otherwise, race condition is found and verification terminates with error report.

Examples

Figure 4 and 5 show the examples of sequentialization of concurrent behaviors. Let us begin with an example in Figure 4. Behaviors *A* and *B* are calling under *par*. There are two pairs of events used for synchronization. It is clearly seen that both synchronization points are always exist since all synchronization statements are not guarded. According to these synchronization points, we can divide portions of code into four basic blocks ($BB1, \ldots, BB4$). Possible execution orders of these basic blocks are shown in Figure 4(b) where $BB1 \rightarrow BB3 \rightarrow (BB2\ interleaved\ with\ BB4)$. There is no race condition since data dependency (no read/write or write/write condition) between *BB*2 and *BB*4 does not exist. Both of them are functionally equivalent, hence, generation of either of them is the sequentialization that of Figure 4(a) and is shown in Figure 4(c).

On the other hand, let us consider the similar example as shown in Figure 5. Both behavior *A* and *B* of Figure 5(a) are slightly different from that of Figure 4(a). By using the same approach, possible execution orders are $(BB1\ interleaved\ with\ BB3) \rightarrow (BB2\ interleaved\ with\ BB4)$ as shown in Figure 5(b). For *BB*2 and *BB*4, they are exactly the same as in Figure 4. However, interleaving of statements between *BB*1 and *BB*3 has data dependencies that caused different behaviors and the read/write access violation. Therefore, in this case, we

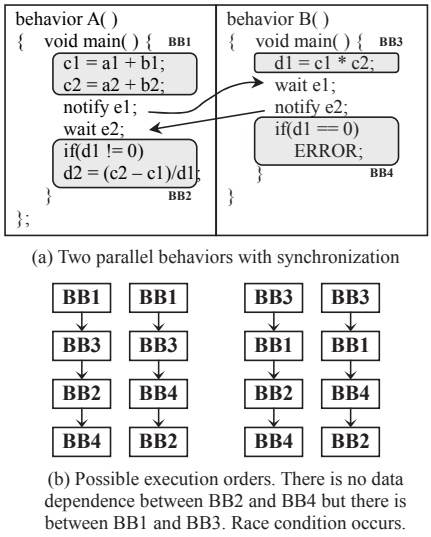

Figure 5. Sequentialization example 2

terminate the sequentialization process and report that this design contains a race condition error.

As shown in Figure 4, even there are two possible execution orders, generation of either of them does not alter the result during equivalence checking. However, sequentialization of parallel behaviors may yield more than one sequential output descriptions. As in our experience, in system-level design the designs usually contain relatively fair numbers of parallel behaviors, hence, the number of the generated sequential descriptions is not significantly large. At present stage of this research, we do not have a systematic or automatic method to classify which sequential description should be selected. We left this task to the designers and we consider this as a challenge in the future research direction.

3.2 Equivalence Checking between Two Sequential Descriptions

When the sequentialization is successfully performed, equivalence checking is applied to the two generated sequential descriptions. The equivalence checking method used here is based on [5]. The method in [5] tries to prove the equivalence by symbolically simulating the descriptions. Also, functions that are proved to be equivalent can be treated uninterpretedly, which results in the speed-up of the verification because most of the functions remain unchanged.

4 Experimental Results

We experiment our tool on several designs written in SpecC. Since we focus on proving mainly for correctness of hardware, our tool does not support the use of pointers, dynamic memory allocation, and recursive functions. If the design under verification contains any of these, we need to

Table 1. Sequentialization results

Benchmark	LOC	Behaviors	Channels	Deadlock	Race condition	Time(s)
IDCT1_1	300	4	1	0	0	0.71
IDCT1_2	314	6	1	0	0	0.75
IDCT2_1	300	4	1	0	0	0.69
IDCT2_2	256	18	1	0	0	0.75
VocSpec	9165	102	4	1	0	38.97
VocArch	10178	144	14	1	0	48.54
VocSched	10139	144	14	1	0	41.97

Table 2. Equivalence checking results

Sequentialized Benchmarks (LOC)		*diff* (LOC)	EqvChecking	Time(s)
seq_IDCT1_1 (225)	seq_IDCT1_2 (235)	156	18304	0.056
seq_IDCT2_1 (225)	seq_IDCT2_2 (161)	148	18304	0.048
seq_VocSpec (8336)	seq_VocArch (8346)	25	22	0.020
seq_VocSpec (8336)	seq_VocSched (8346)	25	22	0.024
seq_VocArch (8346)	seq_VocSched (8346)	0	-	-

remove them manually. Fortunately, in our experiment, all designs contain none of them.

Different levels of implementation of the Inverse Discrete Cosine Transform (IDCT) and the Vocoder, provided by University of California Irvine (UCI), are tested. We performed the experiments on Pentium4 2.8 GHz PC running Linux with 2 GB RAM. Table 1 shows the results of sequentializing all the designs according to the algorithm described in Algorithm 2. For each design benchmark, we give the code size in LOC (line of codes), total number of behaviors and channels utilized, reported errors for deadlock and race condition, and the total runtime in seconds.

As the results from sequentialization, there is no report on race condition in all designs. However, we found one deadlock bug in all Vocoder designs. This is caused by a use of core communication channel that has a corner condition that caused a deadlock. We fix this error and continue to sequentialize the design. Note that even the number of behaviors in the design is large, not all of them are concurrently running. And because in synchronization verification we heuristically explore to find the behaviors running in parallel and contain only synchronization semantics, then, the design size can be reduced and so as the verification time.

Before generating the sequential descriptions, since there is no race condition in all designs, when combining behaviors, any execution order yields the same result. However, because our equivalence checking method is obviously based on proving the equivalence of the difference of the two descriptions, randomly generation of the sequential descriptions can cause the elevated performance of equivalence checking. For example, if we have one description that is *par*{*A1.main(); B1.main(); C1.main();*} and another is *A2.main(); par*{*B2.main(); C2.main();*}. Assume that they are no deadlock and race condition, and behavior $A1$ is equivalent to $A2$, $B1$ is equivalent to $B2$, and $C1$ is equivalent to $C2$, the generated sequential descriptions can be $A1 \rightarrow B1 \rightarrow C1$ and $A2 \rightarrow B2 \rightarrow C2$, or they can be $A1 \rightarrow B1 \rightarrow C1$ and $A2 \rightarrow C2 \rightarrow B2$. Both cases yield the same results, however, in the first one our symbolic simulator can tell right away that the two descriptions are equivalent, whereas it needs to simulate both descriptions in the second case and tells later that they are equivalent. In order to avoid this performance variety, we manually assist in generating the sequential descriptions to make the difference among them as small as possible.

After generating the sequential descriptions, we translate all behaviors into uninterpreted functions. This is for the symbolic simulation propose. The results of equivalence checking of the designs in Table 1 are presented in Table 2. The first two columns presents a pair of designs for equivalence checking. The number inside parentheses represents the number of LOC of the sequentialized design. The next three columns represent the number of LOC that the two designs are different (justify by the size of the output of *diff* command in Unix system, see [5]), the number of times our symbolic simulator's proof engine was called, and the total runtime. For IDCT examples, due to both designs are quite different, the number of time to prove equivalence is increased accordingly. In contrast, the Vocoder designs (except the last row) are quite similar to each other, hence, the proof engine was called only 22 times. For the

last row of the Vocoder designs, the generated descriptions are exactly the same. This is because VocSched is a refined version of VocArch with some schedulings and the control and data dependencies are preserved. There is no need to do symbolic simulation for these designs.

Let us note that in the runtime column of IDCT and Vocoder designs, runtime of Vocoders are approximately half of IDCTs while the numbers of times calling the proof engine are a lot bigger. This is due to the overhead time that caused by our symbolic simulator. The time to call and use the proof engine is significantly small.

5 Conclusion

We proposed a method to prove equivalence of two system-level descriptions that contain parallel behaviors. Various methods such as deadlock checking, race condition detection, and symbolic simulation are utilized in our approach to come up with the generation of sequential descriptions when there is no deadlock or race condition errors, and equivalence checking them. Our prototype tool clearly has a good performance even in the large design like Vocoder, where the size is approximately about 10 KLOC. This is because, with descriptions at high-level, there are less details in communication parts and we target only the difference of concurrent behaviors in the designs.

As mentioned in Section 3, there might be more than one generated sequential descriptions. Systematic way to select which description should be compared with the other is, in our opinion, considered as a motivative direction in this research.

Acknowledgments

We would like to thank the Semiconductor Technology Academic Research Center (STARC) and Japan Society for the Promotion of Science (JSPS) for the support of this research. Also we thank Professor Daniel D. Gajski's research group for providing us the SpecC practical designs.

References

[1] S. Abdi and D. Gajski, "Functional validation of system level static scheduling," in *Proc. of Design, Automation and Test in Europe'05*, 2005.

[2] C. Karfa, C. A. Mandal, D. Sarkar, S. R. Pentakota, and C. Reade, "A formal verification method of scheduling in high-level synthesis," in *Proc. of International Symposium on Quality Electronic Design*, 2006.

[3] P. Ashar, A. Raghunathan, A. Gupta, and S. Bhattacharya, "Verification of scheduling in the presence of loops using uninterpreted symbolic simulation," in *Proc. of International Conference on Computer Design*, 1999.

[4] G. Ritter, *Formal Sequential Equivalence Checking of Digital Systems by Symbolic Simulation*. PhD Thesis, Darmastadt University of Technology, 2000.

[5] T. Matsumoto, H. Saito, and M. Fujita, "An equivalence checking method for c description based on symbolic simulation with textual differences," *IEICE Trans. on Special Section. on VLSI Design and CAD Algorithms*, vol. E88-A, no. 12, December 2005.

[6] T. Sakunkonchak, S. Komatsu, and M. Fujita, "Synchronization verification in system-level design with ILP solvers," in *Third ACM-IEEE International Confernece on Formal Methods and Models for Codesign (MEMOCODE2005)*, Verona, Italy, 2005.

[7] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," in *Proceeding of the International Conference on Computer Aided Verification (CAV'97)*, ser. Volume 1254 of LNCS, O. Grumberg, Ed. Springer-Verlag, 1997.

[8] T. Ball and S. Rajamani, "Boolean programs: A model and process for software analysis," Microsoft Research, Tech. Rep. MSR-TR-2000-14, Febuary 2000.

[9] T. A. Henzinger, R. Jhala, R. Mujumdar, and G. Sutre, "Lazy abstraction," in *Proc. of the 10th SPIN Workshop on Model Checking Software (SPIN)*, ser. Volume 2648 of LNCS. Springer-Verlag, 2003.

[10] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Proc. of the International Conference on Computer-Aided Verification (CAV'00)*, ser. Volume 1855 of LNCS, E. A. Emerson and A. P. Sistla, Eds. Springer-Verlag, 2000.

[11] M. Fujita and H. Nakamura, "The standard SpecC language," in *International Symposium on Systems Synthesis (ISSS2001)*. Montreal, Canada: ACM, 2001.