A Data Path Verifier for Register Transfer Level Using Temporal Logic Language Tokio

Hiroshi NAKAMURA* Yuji KUKIMOTO† Masahiro FUJITA‡
Hidehiko TANAKA†

Institute of Information Sciences and Electronics, The University of Tsukuba, Japan *
Department of Electrical Engineering, The University of Tokyo, Japan †
Fujitsu Laboratories Ltd., Japan ‡

Abstract

A data path verifier for register transfer level is presented in this paper. The verifier checks if all the operations and the data transfers in a behavioral description can be realized on a given data path without any scheduling conflicts. Temporal logic based language *Tokio* is adopted as a behavioral description language in this verifier. In *Tokio*, designers can directly describe concurrent behaviors controlled by more than one finite state machine without unfolding parallelism. The verifier checks for the consistency between a behavior and a structure automatically and lightens the load of designers. The actual LSI chip which consists of 18,000 gates on CMOS gate array has been successfully verified. This verifier is concluded to have the ability to verify practical hardware design.

1 Introduction

Recently, extensive studies have been carried out on the derivation of efficient and error-free data paths in register transfer level assistance. High-level synthesis [5] is one of the solutions to this problem. The approach adopted in high-level synthesis is to synthesize a data path automatically from a given behavioral description. The derived data paths with this approach, however, are not as satisfactory as those which are designed manually yet .

On the other hand, designers initially have the image of a data path to be designed in an actual design process, because they have designed many similar circuits. In addition, they seldom develop hardware which is completely different from the one ever designed. In such situations, it is better to utilize the designers' experience positively instead of synthesizing a data path automatically. It is very important to construct an effective

*Address: 1-1-1 Ten'nou-dai, Tsukuba, Ibaraki 305, Japan. E-mail: nakamura@arch3.is.tsukuba.ac.jp
†Address: 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan. E-mail: kuki@mtl.t.u-tokyo.ac.jp
‡Address: 1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan. E-mail: fujita@flab.fujitsu.co.jp

design assistance system based on this approach. However this has not been discussed well so far. Thus we propose a practical assistance system of register transfer level design based on this approach and present a data path verifier, which is the core part of the system.

Figure 1 shows the design flow of the proposed assistance system. The basic idea in this flow is to construct final data paths by modifying initial data paths given by designers.

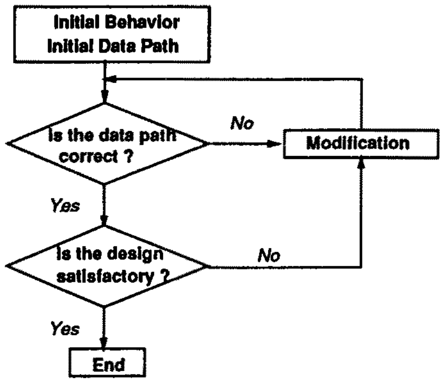


Figure 1: Flow Chart of Practical Design Assistance

At first, a designer gives an initial behavioral description and an initial structure of a data path to be designed. The initial structure is formed through the designer's experience. Now we must verify whether the given behavior can be realized on the given structure. The proposed data path verifier points out when and where scheduling conflicts happen, i.e., which paths and functional units are doubly allocated at which time slot. The structure and the behavior are modified to compensate for that error, and again they are verified. If there is no design error, the performance of the behavior is simulated and the cost of the structure is estimated. If the behavior and the structure satisfy required performance and permitted cost, the design at register transfer level finishes and it proceeds to the lower level design such as logic synthesis. If the cost/performance constraints are not satisfied, the design flow is followed by the stage of modification and the process of verification. This design flow of Figure 1 is justified because the initial data path given by a designer is fairly good in many cases. If the initial data path is fairly good, the revised data path will be very efficient. The data path verifier proposed in this paper plays an important role in this assistance system.

2 An Example

In this section, we explain the semantics of Tokio by showing some simple Tokio programs. After that, taking the example of the circuit which computes square roots, we show how this verifier is effectively used in the process of deriving the proper data path for a pipelined behavior from the data path for a sequential behavior with modification.

2.1 Behavioral Description Language Tokio

Tokio [2][1][4] is a logic programming language based on first-order interval temporal logic [6]. Intuitively, Tokio is regarded as an extension of Prolog with temporal operators. Since Tokio has the notion of time in its own semantics, the various algorithms of hardware can be described flexibly. The essential notion of hardware, such as concurrency and sequentiality, can be specified accurately and simply. Using Tokio, designers can directly describe concurrent behaviors controlled by more than one finite state machine without unfolding parallelism. Parallel behaviors such as pipelined execution can be easily described.

Now we explain the semantics of temporal operators. The expression

```
head :- p,q.
```

denotes that the predicates p and q are executed in the same time interval where the predicate head is defined. Concurrency is represented with a *comma* operator.

Sequentiality is expressed as follows.

```
head :- p && q.
```

The *chop* operator (&&) specifies the sequential execution of the two predicates p and q. This operator divides the interval where the predicate head is defined into two subintervals as shown in Figure 2. The predicate p is executed in the former interval and q is executed in the latter interval.

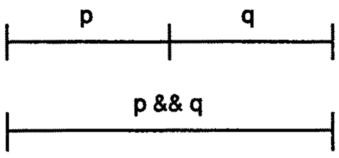


Figure 2: Chop Operator

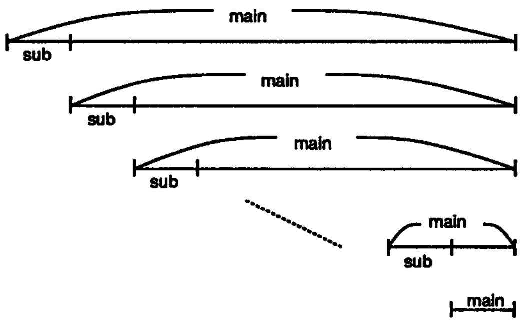
Conditional branches are described by using *cut* operators(!). The semantics of the following predicates is explained on the right side.

head :- condition1 ,!, p.	*if condition1 = true then execute p*
head :- condition2 ,!, q.	*else if condition2 = true then execute q*
head :- !,r.	*else execute r*

Let us take a simple example shown in Figure 3-(a). The statement "*tmp <= *c + *tmp" represents that the data of register *c is added to the data of register *tmp and that the result of the computation is stored in register *tmp at the next clock. The predicate sub denotes the parallel execution of the computation described above and the decrement of the data of register *c. The predicate main denotes that the computation of the predicate sub is repeated until the data of register *c is equal to zero.

```
main :- *c=0 ,!,*output <= *tmp.
main :- !,sub && main.
sub :- *tmp <= *c + *tmp,
       *c <= *c - 1.
```

(a) Sample Tokio Program



(b) Timing Chart of Sample Program

Figure 3: Sample Tokio Program

As seen from the example, the description in Tokio is based on a top-down approach. This approach is very useful for describing hardware because designers can give behavioral descriptions with high modularity. Using this programming style, we need not unfold concurrency of hardware for every clock cycle. On the other hand, it becomes rather difficult for designers to check the consistency between a behavior and a data path. Our strategy is to adopt a flexible behavioral description language and to verify the consistency automatically.

2.2 Computing Square Roots by Sequential Execution

The example to be taken is the circuit which calculates square roots by using Newton's method. The algorithm of Newton's method is shown in Figure 4-(a). The behavioral description in Tokio and the structure of a data path are shown in Figure 4-(b) and Figure 4-(d) respectively. In this behavioral description, the computation of Newton's method is realized by sequential execution for each input data. In this case, the verification of the data path is not so difficult because there is little concurrency in the description. We have only to verify the data path for each local interval and need not check whether some of these intervals occur concurrently.

2.3 Deriving a Data Path for Pipelined Execution

Now we modify the sequential behavioral description of the circuit computing square roots and derive a pipelined behavioral description from it. It is quite easy to get a pipelined description from the original one by using temporal operators. The pipelined description is shown in Figure 4-(c). We have only to change the top-level predicate `main`. The second `main` predicate denotes that `stage1` is followed by both `main` and `(stage2 && true)`. Since `main` starts the computation for the next input data, this behavior represents a pipelined execution. Figure 5 shows a timing chart for the pipelined execution.

At this point, we must construct a data path for the pipelined behavior, because the data path for sequential computation may not be sufficient. To construct a completely new data path, however, is not efficient. It is better to modify the original data path

```
Y := 0.222222 + 0.888889 * X;
C := 0;
DO UNTIL C = 2 LOOP
  Y := (Y + X / Y) / 2;
  C := C + 1;
ENDDO;
```

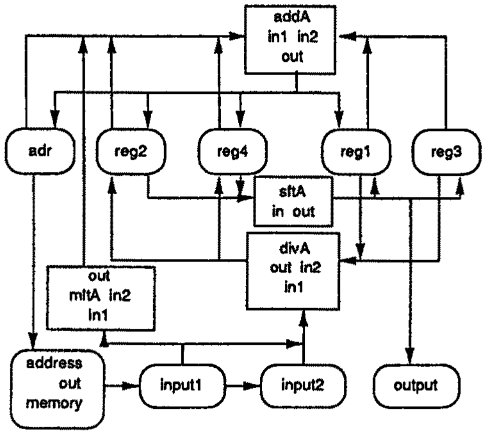
(a) Algorithm of Newton's Method

```
main :- *adr = 8,!,true.
main :- !,input && stage1 && stage2 && main.
input :- !, *input1 <= *memory(*adr), *adr <= *adr + 1
     && *reg1 <= 0.222222 + 0.888889 * *input1.
stage1 :-!, *reg2 <= *input1 / *reg1
     &&  *reg2 <= *reg2 + *reg1
     &&  *reg3 <= *reg2 / 2,  *input2 <= *input1.
stage2 :- !, *reg4 <= *input2 / *reg3
     &&  *reg4 <= *reg4 + *reg3
     &&  *output <= *reg4 / 2.
```

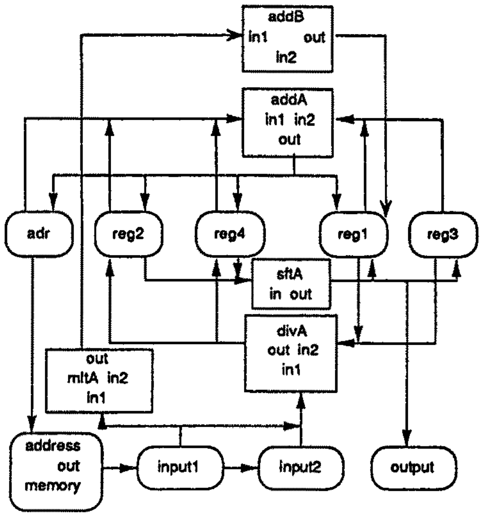
(b) Sequential Behavioral Description in Tokio

```
main :- *adr = 8,!,true.
main :- !,input && stage1 && main,(stage2 && true).
```

(c) Pipelined Behavioral Description in Tokio



(d) Data Path for Sequential Execution



(e) Data Path for Pipelined Execution

Figure 4: Computation of Square Roots

and to get the proper one. In such a case, we can use this verifier effectively. At first, the verifier checks whether the pipelined behavior can be realized on the original data path. If there are some scheduling conflicts, the behavioral description or the data path is modified considering the output of the verifier in order to avoid the conflicts. This process is repeated until there are no conflicts and the proper data path is constructed.

From the result of the verification, we can say that adder addA is doubly used in the second interval of input and the second interval of stage2 and that these intervals occur concurrently. Thus we add one more adder to the original structure and the final data path is obtained. The data path for the pipelined behavior is shown in Figure 4-(e).

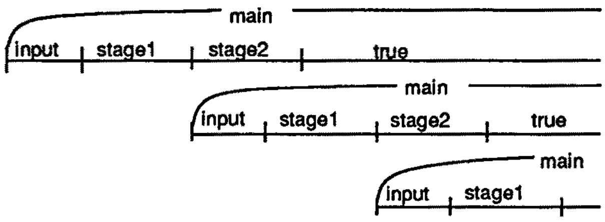


Figure 5: Timing Chart for Pipelined Execution

3 Verification Method

3.1 Overview

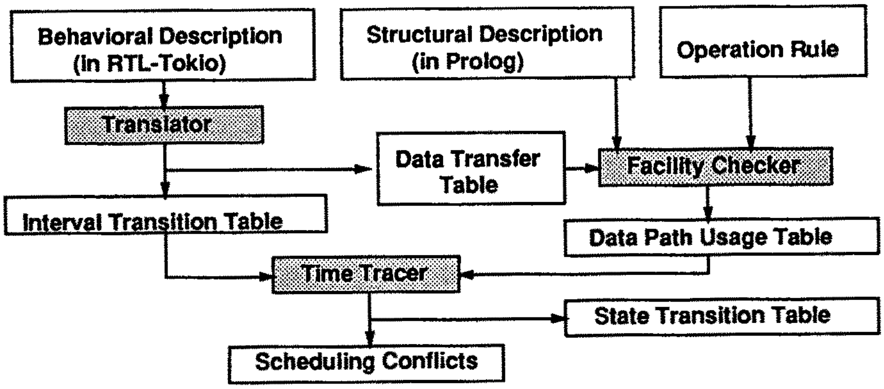


Figure 6: Structure of Data Path Verifier

The data path verifier proposed in this paper checks if all the operations and the data transfers in a behavioral description can be realized on a given data path without any scheduling conflicts. The structure of this verifier is shown in Figure 6. The inputs of this system are a behavioral description in RTL-Tokio (Register Transfer Level Tokio, which is a subset of Tokio), a structural description in Prolog, and operation rules in Prolog. The operation rule is a rule for linking an operation in the behavior with a functional unit in the data path.

The process of the verification is divided into two stages.

Linking a behavior with a data path: To find a set of paths and functional units for each data transfer and operation from the structural description. In this stage, all the data transfers and operations in the behavior are linked with appropriate data path elements in the structure using operation rules, and the link information between them is recorded in a data path usage table. This stage is executed by a *translator* and a *facility checker.*

Detection of scheduling conflicts: To verify whether any paths or functional units are doubly allocated in concurrent time intervals. The concurrency of the behavioral description in RTL-Tokio is unfolded in this process. The state transition table of a control part is also extracted. This stage is executed by a *time tracer.*

3.2 Linking a Behavior with a Data Path:

The link information between the behavior and the structure is derived in accordance with the following steps.

1. To find a set of functional units which realize an operation of each data transfer from the structure.
2. To search for a data path from a source register to the input of the functional unit and that from the output of the unit to a destination register.

The function of each unit is defined by operation rules. At the end of this stage, the link information is recorded in a data path usage table. From this table, we can know which data path element is used by each data transfer and operation. The names of the registers and those of the memories in the behavior are assumed to be the same as those in the structure.

3.3 Detection of Scheduling Conflicts

In this stage, scheduling conflicts in data paths are detected by unfolding the concurrency of a behavior to actual data transfers with a time tracer. The time tracer makes clear what operations and data transfers are done concurrently by traversing all the transitions. After that we check if any paths or functional units are doubly allocated in concurrent time intervals by using a data path usage table. If no path or functional unit is doubly allocated, the given data path turns out to be correct. Otherwise designers modify the behavior and the data path to avoid the detected scheduling conflicts. We have implemented this time tracer in two ways: a forward tracer and a backward tracer.

- *Forward Trace*: At first all the concurrent time intervals are searched for by tracing transitions forward. In the second stage, all these intervals are checked whether they use the same data path element. The scheduling conflicts of data paths are detected in this stage.
- *Backward Trace*: In a backward trace, all the pairs of time intervals which use the same data path element are searched for at first, and they are verified whether they occur concurrently by tracing transitions backward in accordance with an interval transition table.

In this paper, only the algorithm of the forward trace is explained. The algorithm of the backward trace is similar to that of the forward trace except for a direction of tracing.

3.3.1 Unfolding Concurrency of a Behavior

In the first stage, all the concurrent time intervals are searched for with a forward tracer. It traverses all the transitions from an initial state in depth-first. A state in RTL-Tokio description is defined by [*interval-name*,*clock-number*].

step1: An initial interval I_{init} and an initial clock I_{clock} (usually 0) are selected.
$S_0 = \{[I_{init}, I_{clock}]\}$.

step2: (Unfolding the concurrency of a behavior)
If S_i has predicate calls, all these concurrent intervals are added to S_i and S_i' is obtained. S_i' is a list of the intervals which occur concurrently at the i-th clock.

step3: (Proceeding to the next clock)
S_{i+1} is obtained from S_i' by proceeding to the next clock. The computation of this process is as follows.

1. Increment the clock number I_{clock} for each element of S_i' if the obtained clock number is not larger than the length of that interval.

2. Transfer to the next interval after *chop* operator &&.

Step3 is followed by step2.

Detection of the iteration of a search: Let S_n' be the newly obtained state in step2. If $0 \leq \exists\, i < n,\ S_n' \subseteq S_i'$ holds or S_{n+1} next to S_n' cannot be obtained, the iteration of a search is detected.

3.3.2 Detection of Scheduling Conflicts

All the intervals in each S_i' occur concurrently unless they have exclusive transitive conditions. Using a data path usage table, all the concurrent intervals are checked whether they use the same data path resource or not. If they use the same path or the same functional unit, it results in scheduling conflicts of data paths. In this stage, we check if the data path conflicts can be avoided by using some alternative data paths.

4 Experimental Results and Evaluation

4.1 Network Interface Processor

The largest example to which we have applied this verifier is a network interface processor (NIP) in PIE64 [3]. PIE64 is a parallel inference machine which executes knowledge information processing in parallel, and is under development in our laboratory. This machine consists of 64 inference units and two high speed interconnection networks. The processor manages data transfers and process synchronization between inference units. It has already been designed and is going to be implemented on CMOS gate array. The total number of the gates is about 18,000 including both a data path and a control part. The structure of the processor is divided into four parts. The data path verifier has been applied to the main parts which consist of about 11,000 gates. Since the required performance of this processor is very severe, the behaviors of NIP are 6-stage pipelined.

The verification results of the two parts of the processor are shown in Table 1. The data path verifier is implemented on SUN4/260 using SICStus-Prolog (about 80KLIPS).

	CPU time (sec)				Number of Derived States	Number of State Transitions
	Trans-lator	Facility Checker	Forward Trace	Backward Trace		
Data Transfer Part (Upper Part)	0.80	1.07	2.36	40.1	14	53
Data Transfer Part (Lower Part)	3.23	4.66	183.3	>3600	81	379
Process Synchronization Part	5.97	7.85	1488	>3600	236	1168

Table 1: Verification Results of Network Interface Processor

4.2 Evaluation

Most of the CPU-time is spent in a *time trace* part. Since a forward tracer traverses all the state transitions, its cost is proportional to the number of transitions. The cost of detecting the iteration of the search is proportional to the number of states because traced states are presently recorded with enumeration, Thus the computational complexity for the forward trace is $O(n \times m)$, where n is the number of state transitions and m is the number of states.

In a backward trace, ${}_N C_2$ pairs of intervals are listed in the worst case, where N is the number of intervals. The worst case occurs when a certain data path element (such as a bus) is used in all the intervals. Though a backward tracer is not so efficient for complete verification from a computational aspect, it is quite useful and efficient for partial verification by selecting one interval and searching for concurrent time intervals with it. Therefore, it is recommended that the important part of design is tested partially using the backward trace at first, and then the whole design are verified using the forward trace.

5 Conclusion

We have presented a data path verifier at register transfer level and proposed a design assistance system based on this verifier. The verifier has been successfully applied to a real LSI chip. Practical hardware design can be assisted with this verifier if an interactive tool for improvement is provided. Our current research aims at the assistance of this improvement stage.

References

[1] T. Aoyagi, M. Fujita, and T. Moto-oka. Temporal Logic Programming Language Tokio. In *Logic Programming Conference '85*, pages 128–137, Springer-Verlag, 1985.

[2] M. Fujita, S. Kono, H. Tanaka, and T. Moto-oka. Aid to Hierarchical and Structured Logic Design Using Temporal Logic and Prolog. In *IEE Proceedings, Vol.133, Pt.E*, pages 283–294, IEE, 1986.

[3] H. Koike and H. Tanaka. Multi-Context Procesing and Data Balancing Mechanism of the Parallel Inference Machine PIE64. In *Fifth Generation Computer Systems*, pages 970–977, ICOT, 1988.

[4] S. Kono, T. Aoyagi, M. Fujita, and H. Tanaka. Implementation of Temporal Logic Programming Language Tokio. In *Logic Programming Conference '85*, pages 138–147, Springer-Verlag, 1985.

[5] M.C. McFarland, A.C. Parker, and R. Camposano. Tutorial on High-Level Synthesis. In *25th Design Automation Conference*, pages 330–336, ACM/IEEE, 1988.

[6] B. Moszkowski. A Temporal Logic for Multi-Level Reasoning about Hardware. In *CHDL '83*, IFIP, 1983.