ISSN 0002-306X. Изв. НАН РА и ГИУА. Сер. ТН. 2009. Т. LXII, № 2.

UDC 530.15: 004.032.26 COMPUTER SCIENCE AND INFORMATICS

PAVLUSH S. MARGARIAN

AN APPROACH FOR AUTOMATED ASSERTION GENERATION IN TEMPLATE-BASED RTL COMPILERS

An approach for assertion generation which can be used for functional verification of RTL instances generated by a template-based RTL compiler is presented. The suggested approach provides automatic generation of assertions during a RTL compiler development and maintenance. An application of the approach for specific RTL compilers for illustrating the effectiveness of the approach is presented.

Keywords: RTL compiler's output verification, formal verification, symbolic simulation, functional verification, model checking, hardware verification.

1. Introduction. An approach for automated generation of assertions to be used in functional verification of Register Transfer Level (RTL) instances [1] is presented for the case when these instances are generated by a RTL compiler [2-4]. A subset of RTL compilers - template based RTL compilers are considered. They usually consist of templates that describe a parameterized hierarchy of modules and interfaces (interconnections) between them and a generation engine. A typical example of a RTL generating template in Tcl is shown in *Fig 1*. The template input is a vector of input parameter values which defines features and a structure of design instances to be generated, and the output of the template is a RTL description with functionality corresponding to the given input.

Input parameters can be categorized by the following three types:

- ***Functional*** (P_{Func}^{i} in Fig. 1)

These parameters control optional features/options of design. They affect the design structure by means of inclusion or exclusion of certain design components in the output RTL.

- ***External interface*** (P_{Exlf}^{i} in Fig. 1)

They parameterize HDL identifiers (module/wire/reg/instance) in a RTL description for customization of the generated RTL design to an external interface.

- ***Scalability*** (P_{Scal}^{i} in Fig. 1).

These parameters affect the design structure by increase or decrease of certain design characteristics including but not limiting the register bit-width, number of words (in memories), number of cores (e.g. in SoCs), etc.

It is difficult to perform the separation of parameters formally. Meantime a compiler developer without extra efforts can perform that during the compiler design.

To accelerate the separation, a special tool, which parses the templates of a RTL compiler, then extracts and roughly pre-categorizes all template parameters via analysis of templates logic and RTL fragments, has been developed. Further, this categorization can be revised by a compiler developer to provide more accurate separation. This can be considered as a step in the design process, and we assume further that such a separation already exists.

Actually, these parameters are the basis on which the set of compiler templates is built. Control of the design hierarchy and the design features from a template space is performed via them.

The process of functional verification for the built compiler outputs comprises two steps:

- RTL instances generation,
- RTL instances functional verification.

A big number of output RTL instances for all possible input vector values and, correspondingly, unacceptable total time of functional verification for generated instances (either via simulation or formal verification) brings a real challenge to reach a 100% coverage of functional verification for compiler outputs within a reasonable verification time.

The paper considers a possible solution of the problem focusing on two issues: reduce of the number of generated instances and a common input for both simulation and formal verification methods during the functional verification step.

The list of input parameters for a given RTL compiler contains parameters which do not influence directly on the functionality of the generated instance. A way is discussed below how to exclude them from consideration during the functional verification of a RTL compiler outputs.

Simultaneously, the key factor at the functional verification step of a given RTL instance is a choice of a verification method.

Usually, it is a simulation of the generated instance. This approach is time consuming and can take weeks to complete the verification [5, 6]. Besides, the traditional simulation can explore only a small percentage of the reachable design state space due to the fact that the number of values of input vectors required for exhaustive state coverage rises exponentially with the number of input bits and state bits in the design [7].

The other used method is a formal verification [7]. It does not rely on traditional logic simulation or test vectors and utilizes a symbolic simulation and symbolic values instead of the logic zero and one values used by traditional logic simulators [8, 10] for increasing the considered state coverage of the design [9, 11].

Meantime, the formal verification also has time and resource consuming steps, specifically, assertion formulation and, finally, does not replace simulation – it rather complements it.

The suggested input in a form of automatically generated assertions can be used both in simulation and formal verification and, thus, it will mutually reduce time consuming steps as well as will stimulate usage of any flexible combination for these verification methods.

2. Methodology. Paths in a given compiler template which do not affect directly functionality of output RTL instances are redundant for considerations connected with functional verification of these instances and can be eliminated from the consideration, e.g., by assigning fixed values to parameters which activate these paths.

Specifically, per the characterization above, the whole set P_{Exlf}^{i} of external interface parameters is redundant and can be eliminated from the template logic by assigning fixed values to them. This can be considered as a first step of our approach. The elimination means an assignment of constant values from their value set to all external interface parameters. The step is illustrated in *Fig 2* via assigning the constant value **MemWrapper** to the parameter *wrapper_name*.

The second step is unrolling loops related to scalability parameters from the P_{Scal}^{i} list in the template logic.

Each of these parameters has its own value set which is specific for a given compiler. During the unrolling an assignment of the list of possible values to the considered parameter is performed replacing the corresponding loop. An example of a loop unrolling for the parameter *wr_read_portid_list* is adduced in *Fig 2*.

Both two steps can be completely automated.

Fig. 1. A compiler template in Tcl

The third step is a separation of the RTL description fragments from the template logic. This process results in RTL fragments which become a subject of verification for the compiler developers and a “pure” template logic which does not contain RTL descriptions. It is not difficult to automate the mentioned separation too.

The fourth step is the development of assertions for RTL fragments obtained at the previous step. This process is manual and should be performed by the compiler developers. This is not difficult because these fragments are usually self-developed, small and, thus, it is easy to formulate the corresponding assertions for mentioned fragments. These assertions also can be obtained independently from separate items of initial requirements for the considered compiler.

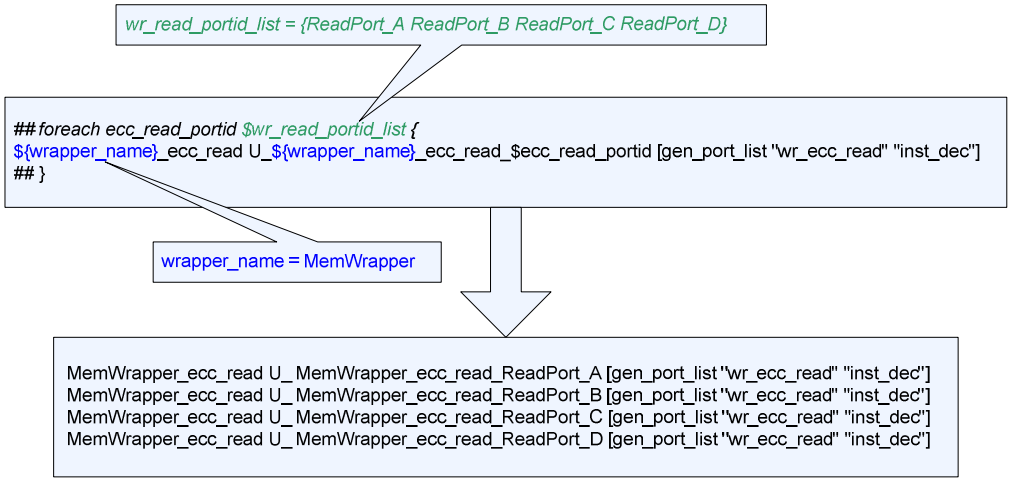

Fig. 2. Assigning constant values and unrolling loops in the template

The next step is the construction of a *Multidirectional Decision Diagram* (MDD) for description of the template logic. The MDD nodes corresponding to variables controlling the template logic and the edges originating from such a node correspond to the value set for the corresponding variable.

The implementation in a form of a MDD leads to an automatic generation of input vectors covering all paths of the template. This can be used to provide 100% path coverage of the template. Each path in MDD starting from the root to one of the MDD end nodes corresponds to an input vector for a given RTL template and an output RTL instance. The MDD for the input template in *Fig. 1* is adduced in *Fig. 3*.

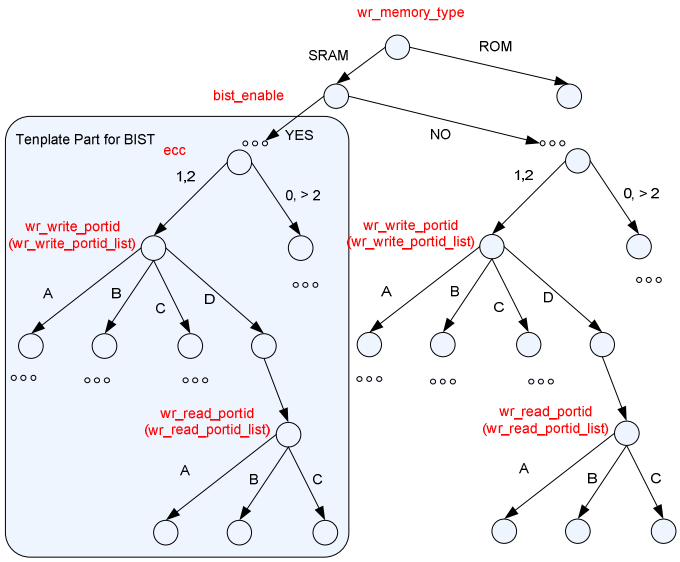

Fig. 3. A MDD for building an assertion generation template

At the last step a new template should be developed per already built assertions for RTL fragments and the built MDD. This template provides generation of complete assertions for RTL output instances generated by the compiler. The generation of an assertion occurs simultaneously with the generation of a RTL instance.

Main steps of the proposed solution are summarized in *Fig. 4*.

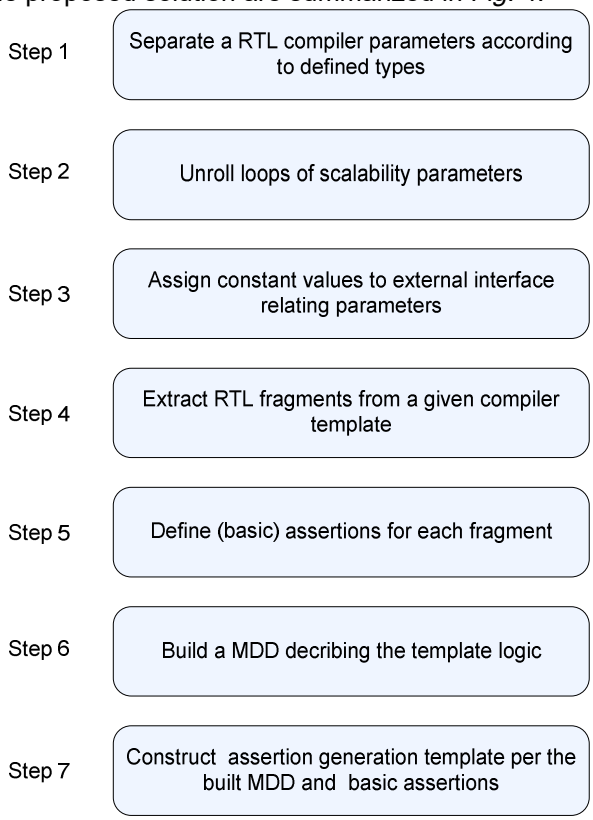

Fig. 4. A flow of the approach

In order to double check the correctness of hand-written basic assertions the following way can be used. A verification of a RTL design should ensure that a built RTL instance behaves according to a set of requirements (a specification) which contains a set of specific properties to be satisfied. These specific properties correspond to assertions for fragments outputs. Thus the double check can be in comparison of constructed basic assertions with mentioned properties described in the specification.

There are several languages for assertion description, which are supported by various tools. The CTL language was chosen in our case for the implementation of the approach due to many advantages [7, 13, 15]. The construction of CTL assertions is usually a major part of efforts in the formal verification of RTL instances [12]. The suggested approach avoids time consuming process of assertion construction for every design instance due to the assertion generation template described above.

3. A model for effectiveness estimation. As it was mentioned in the introduction, a functional verification of outputs of a RTL compiler is a verification of all instances generated by a RTL compiler. This is time consuming process and it requests essentially both computer and

human resources because generated instances should be verified using a formal verification tool on manually written verification assertions or simulation on multiple input vectors. The number of output instances is usually close to thousands.

A total verification time can be expressed by the formula:

$$T_{Ver} = \sum_{i=1}^{N} (T_{asser}^{(i)} + T_{ver}^{(i)} + T_{analys}^{(i)})$$

where: T_i is a verification time for the instance i, N is number of all instances generated by a RTL compiler. It is clear that N has an exponential dependence on the number of input parameters and the resulting verification time is unacceptable for verification of RTL compilers.

Verification time for the instance i can be expressed as:

$$T_i = T_i^{(asser)} + T_i^{(ver)} + T_i^{(analys)}$$

$$T_{Ver} = \sum_{i=1}^{N} (T_i^{(asser)} + T_i^{(ver)} + T_i^{(analys)})$$

Possible ways of reducing the verification time are the following:
a decrease of the number of considered paths in templates;
a decrease of the number of manually written assertions.
Per considerations in the methodology section the decrease of the number of paths depends on a power of the P_{Descr} set. Similarly the decrease of the number of manually written assertions depends on a power of the P_{Scal} set.

Some applications of the suggested methodology to specific RTL compilers are adduced in the next section. They illustrate an essential decrease of the verification time for the considered cases.

Meantime it is clear that a representation of a compiler for a given design in a form of compiler hierarchy could also reduce the verification time of the compiler. The issues connected with it will be considered in our further publications.

4. Application for specific RTL compilers. A software system which provides a development and integration flow for a critical component of systems-on-chip (SoC) implemented as hierarchical silicon aware IP is described below.

The case is a multi-layer Embedded Memory Test and Repair Infrastructure (EMTRI) based on a special standardized knowledge model covering:

- memory functionality, its structure, fault and repair models;
- adaptation of test and repair algorithms to customer specific peculiarities;
- basic operations and operations pipeline for test and repair, implemented in a form of processors for specific memories;
- mechanisms of building a network in the case of different processors;
- system testing of the built network;
- mechanisms of the infrastructure insertion in an existing RTL level functional design and its effectiveness evaluation;
- silicon debug, yield analysis and yield estimation for the infrastructure within the considered SoC.

Implementation of the infrastructure is done in a form of a hierarchy of RTL compilers and software tools exploring the database generated by the hierarchy. Each

specific instance of the infrastructure is defined via assignment specific values to parameters of compilers and generates of components and subcomponents for each level. The hierarchy has both hardware and software levels. Hardware levels will be considered further. They include wrapper, processor and server levels.

The Wrapper Compiler for memory component operates as an interface between processor and memory instance. Processor is aimed to perform different BIRA/BIST and related actions. Server compiler is assigned to build the top-level design infrastructure utilizing the generated low level processor, wrapper, and memory modules [13,14].

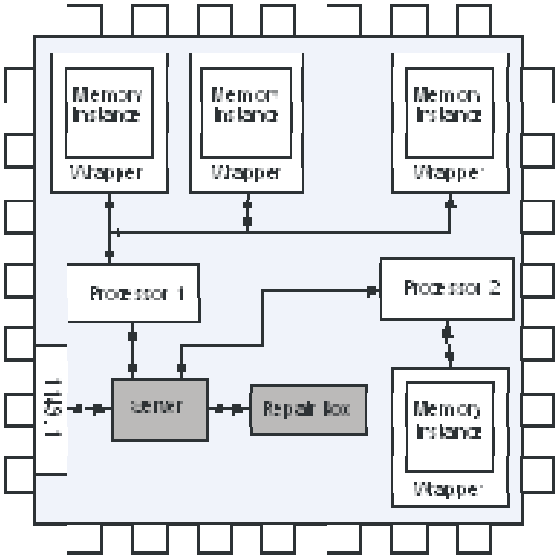

Fig. 5. EMT&RI infrastructure

This brings additionally a scalability and parameterization of the approach. The standardization of the knowledge leads to utilization of the approach for a broad range of memories satisfying the appropriate requirements.

A special process flow will be described briefly to illustrate building of the mentioned infrastructure with a possibility of independent development of separate components. Corresponding verification and characterization steps of the flow will be adduced too to demonstrate technology-wise advantages of the approach reducing time-to-market.

4.1 EMT&RI verification example. A verification of one level of the hierarchy, the processor level will be considered below.

For formal verification of the Processor we have to generate a set of assertions for all its instructions and then to justify that a given assertion holds for any value of parameters.

Assertion generation process via the template built according to the methodology above and further verification of the obtained assertion will be considered below for one of the Processor instructions: BIST_RUN which tests the memory array for faults. This instruction does not have operands and its binary code is 001010. A CTL assertion for verification of the functional behavior of Processor during interpretation of BIST_RUN should be connected with a status register of the Processor which holds information about the current state. Specifically, it should be connected with values of bits *fail* and *ready* of the status register, reflecting the following. *Fail* indicates that BIST_RUN instruction has detected faults in the memory array. *Ready* indicates that the Processor has finished the

execution of the last instruction and is ready to execute the next instruction. A part of assertion generation template relating to the generation of the assertion for BIST_RUN instruction is adduced in *Fig. 1* and the resulting assertion is adduced in *Fig. 5*.

```
Check_BIST: SPEC AG ((
        !U_SMS.U_proc_stp_p1500.WIR_r[0] &
        !U_SMS.U_proc_stp_p1500.WIR_r[1] &
        U_SMS.U_proc_stp_p1500.WIR_r[2] &
        !U_SMS.U_proc_stp_p1500.WIR_r[3] &
        U_SMS.U_proc_stp_p1500.WIR_r[4] &
        !U_SMS.U_proc_stp_p1500.WIR_r[5] =>
        AG (U_SMS.U_proc_stp.status_reg[3]  = >
         !U_SMS.U_proc_stp.status_reg[4]));
```

Fig. 6. Generated assertion

Generated CTL assertion in *Fig. 6* can be read as: it is always the case that if BIST_RUN instruction is loaded into Instruction Register, then for all computation paths there should be state, where *ready* is true (test always is finite, all FSM paths starting from current state should lead to the ready state) then eventually *fail* will be false (memory array should be fault-free). We need two verification models for this test case, one with fault injected memory model, and the other for fault-free memory model. For correct functioning design we must get false with fault injected memory model and, true for fault-free memory model.

All assertions targeted to functional verification of output RTLs should be implemented in the same way and according flow presented in *Fig. 4* combine in a single template.

The application results for the EMTRI are presented in *Table*.

Table

ER&R application results

Total Number of RTL Instances	$T^{(assertion\ formulation)}$ (Hours)		$T^{(verification)}$ (~5 min per instance)	$T^{(analysis)}$		Total (Hours)		Ratio %
	Before (~1.5 per instance)	Our Approach		Before (~6 min per instance)	Our Approach (~3 min per instance)	Before	Our Approach	
16	24	80	1.44	1.6	0.8	27	82.24	304
32	48	80	2.88	3.2	1.6	54	84.48	156
64	96	80	5.76	6.4	3.2	108	88.96	82
128	192	80	11.52	12.8	6.4	216	97.92	45
256	384	80	23.04	25.6	12.8	432	115.84	27
512	768	90	46.08	51.2	25.6	865	161.68	19
1024	1536	90	92.16	102.4	51.2	1730	233.36	13
2048	3072	100	184.32	204.8	102.4	3461	386.72	11

Note: Assertion generation time is not considered in the Table because it is too small (1,2 seconds) related to other time periods.

To improve analyzing cost of obtained results, a special tool which reads symbolic simulation trace files, checks results for predefined constraints and results displays in user friendly way has been developed.

5. Conclusion. The presented approach can be used not only for assertion based verification, but also for simulation based verification: instead of verification assertions parameterized verification tasks with input vectors will be used.

Results presented in the *Table* show that the application of the approach to RTL compilers is acceptable when the total numbers of verification instances are more than 50. In industrial RTL compilers this number is thousands, so for most of cases the approach can reduce the verification period.

It is planned to extend the suggested approach in four directions.

First, it is to increase the range of the application of the suggested approach by applying it to parameters which were not engaged into the consideration in this paper.

Second, it is to define some standard requirements to compilers which can successfully be passed and effectively through the suggested flow.

Third, it is to optimize time duration of the suggested verification steps. One point of improvement is to increase the accuracy of the parameters consideration, i.e. do not consider parameters which are not in the mentioned group but meantime their changes do not influence the verification process.

Fourth, it is to consider directly the impact of a hierarchy of compilers on the verification time.

REFERENCES

1. **Douglas J.S.** HDL Chip Design. Doone Publications 1996. ISBN 0-9651934-3-8.
2. **Cummings C.E.** Fsm_perl: A Script to Generate RTL Code for State Machines and Synopsys Synthesis Scripts.- SNUG, 1999.
3. **Horstmannshoff J., Meyr H.** Efficient Building Block Based RTL Code Generation form Synchronous Data Flow Graphs // Annual ACM IEEE Design Automation Conference: Proceedings of the 37th Conference on Design automation. - 2000. - P. 552 – 555.
4. **Kejariwal A., Mishra P., Astrom J., Dutt N.** HDL Generation Method for Configurable VLIW processor // CECS Technical Report #03-04, Center for Embedded Computer Systems/ University of California.- February, 2003.
5. **Bening L. A** Two-State Methodology for RTL Logic Simulation, Hewlett-Packard// Proceedings of the 36th ACM/IEEE conference on Design automation.- New Orleans, Louisiana, United States, 1999.- P. 672 - 677, ISBN:1-58133-109-7.
6. **Evans A., Silburt A., Vrckovnik G., Brown T., Dufresne M., Hall G., Ho T., Liu Y.** Functional Verification of Large ASICs // Proceedings of the 35th annual conference on Design automation. -San Francisco, California, United States, 1998.- P. 650 – 655, ISBN:0-89791-964-5.
7. **Katoen J.-P.** Concepts, Algorithms, and Tools for Model Checking, 1999.
8. **Narasimhan N., Kalyanaraman R., Vemuri R.** Validation of Synthesized Register Transfer Level Designs Using Simulation and Formal Verifcation // High Level Design Validation and Test Workshop.- 1996.
9. **Bavonparadon P., Chongstitvatana P.** RTL Formal Verification of Embedded Processors // Industrial Technology, 2002. IEEE ICIT '02. 2002 IEEE International Conference. ISBN: 0-7803-7657-9. -2002. -Vol.1.- P. 667- 672.
10. **Kropf T.** Introduction to Formal Hardware Verification, Springer, 1999.
11. **Barjaktarovic M.** The State-of-the-Art in Formal Methods AFOSR Summer Research Technical Report for Rome Research Site // Formal Methods Framework-Monthly Status Report, F30602-99-C-0166, WetStone Technologies, 1998.
12. **Nguyen H. N.** Hierarchical Assertion-Based Verification, Bull S.A. – METASymbiose S.A.S. IP-Based SoC Design Intl. Workshop, 2004.

13. **Zorian Y.**, **Shoukourian S.** Embedded-Memory test and repair: Infrastructure IP for SoC yield. IEEE Design and Test of Computers, May-June 2003.
14. **Shoukourian S., Vardanian V., Zorian Y.** SoC yield optimization via embedded-memory test and repair infrastructure // IEEE Design and Test of Computers.- May, 2004.
15. **McMillan K.** Symbolic Model Checking: Ph.D. Thesis, Carnegie Mellon University.

SEUA. The material is received 15.03.2009.

Պ.Ս. ՄԱՐԳԱՐՅԱՆ

ՊՆԴՈՒՄՆԵՐ ԳԵՆԵՐԱՑՆԵԼՈՒ ԱՎՏՈՄԱՏԱՑՎԱԾ ՄՈՏԵՑՈՒՄ ՇԱԲԼՈՆՆԵՐԻ ՀԻՄԱՆ ՎՐԱ ԿԱՌՈՒՑՎԱԾ RTL ԿՈՄՊԻԼՅԱՏՈՐՆԵՐԻ ՀԱՄԱՐ

Ներկայացված է RTL կոմպիլյատորների ելքային RTL նկարագրությունների վերիֆիկացիայի մի մոտեցում: Ներկայացված մոտեցումը թույլ է տալիս ավտոմատ կերպով գեներացնել տվյալ RTL նկարագրության վերիֆիկացիայի համար անհրաժեշտ պնդումների բազմությունը: Մոտեցման արդյունավետությունը լուսաբանելու համար հատուկ RTL կոմպիլյատորների համար ներկայացված է կիրառության օրինակ:

Առանցքային բառեր. RTL կոմպիլյատոր, RTL նկարագրությունների ֆունկցիոնալ վերիֆիկացիա, սիմվոլիկ սիմուլյացիա, ձևական վերիֆիկացիա, ձևական վերիֆիկացիայի պնդումների գեներացիա:

П. С. МАРГАРЯН

ПОДХОД К АВТОМАТИЧЕСКОЙ ГЕНЕРАЦИИ УТВЕРЖДЕНИЙ ДЛЯ RTL КОМПИЛЯТОРОВ, ОСНОВАННЫХ НА ШАБЛОНАХ

Рассматривается подход к генерации утверждений, которые используются для функциональной верификации RTL описаний, сгенерированных RTL компилятором, основанным на шаблонах. Представлено применение этого подхода к специальным RTL компиляторам, иллюстрирующим эффективность этого метода.

Ключевые слова: RTL компилятор, функциональная верификация RTL описаний, символическая симуляция, функциональная верификация, формальная верификация.