Formal Synthesis in Circuit Design - A Classification and Survey

Ramayya Kumar[1],Christian Blumenröhr[2],Dirk Eisenbiegler[2]and Detlef Schmid[1,2]

[1] Forschungszentrum Informatik,
Department of Automation in Circuit Design, Karlsruhe, Germany
e-mail: kumar@fzi.de
[2]Institute of Computer Design and Fault Tolerance
University of Karlsruhe, Germany
e-mail: {blumen,eisen,schmid}@ira.uka.de

Abstract. This article gives a survey on different methods of formal synthesis. We define what we mean by the term formal synthesis and delimit it from the other formal methods that can also be used to guarantee the correctness of an implementation. A possible classification scheme for formal synthesis methods is then introduced, based on which some significant research activities are classified and summarized. We also briefly introduce our own approach towards the formal synthesis of hardware. Finally, we compare these approaches from different points of view.

1 Introduction

In everyday use, synthesis means putting together of parts or elements so as to make up a complex whole. However in the circuit design domain, synthesis stands for a stepwise refinement of circuit descriptions from higher levels of abstraction (specifications) to lower ones (implementations), including optimizations within one abstraction level.

Synthesis can be performed by hand for small circuits. Nowadays more and more automated synthesis tools are in use to harness the complexity of the circuits[McPC90]. Both hand-made and automatically computed hardware implementations may be incorrect with respect to the given input specification. By correctness we mean that the implementation satisfies the specification, in a formal mathematical sense. However, it is possible that the specification itself may be incomplete or erroneous. These anomalies in the specification will have to dealt with separately, either via simulation or by checking certain properties such as safety and liveness. In the rest of the paper, it is assumed that the specification is the reference with which the implementation is checked and the correctness aspects dealt with assume correct specifications.

Examining the correctness of the implementation with respect to a given correct specification, the errors in the hand-made circuits are created by the designer, whereas the errors in synthesized implementations depend on the correctness of the synthesis tools involved. The use of automated synthesis tools significantly increases the reliability as compared to hand designs. During such synthesis, the designer interacts with the system only in a controlled manner. The implementations cannot be created manually and hence the amount of errors are reduced. Furthermore, such systems offer the capability of design space exploration which has a great impact on the quality of designs, especially in the context of designing complex systems.

The ‘‘correctness by construction” paradigm that has been propagated in the synthesis domain is nevertheless questionable. Synthesis programs have become more and more complex especially at higher levels of abstraction (algorithmic level, system level). Sophisticated algorithms and data structures are used and they usually programmed as hundreds of thousands of lines of code, in imperative programming languages like C. Due to their complexity, such programs can only be partially tested. Added to that, these programs are implemented by a team of programmers. This scenario enforces careful thought about the complex interfaces between the modules which manipulate the complex data structures representing the hardware. Yet another source of bugs in synthesis programs are wrong interpretations of HDLs (hardware description languages) that are used for specifications. To embed HDLs, one has to implement procedures for converting hardware descriptions given in the HDL into the internal representation and vice versa. To guarantee the overall correctness of a synthesis tool, one also has to consider the correctness of these transformations with respect to the semantics of the HDLs and the semantics of the internal hardware representation. All these reasons motivate us to closely examine the approaches for the correct synthesis of hardware.

We will first give a definition of formal synthesis and delimit it from related approaches towards correct synthesis. In section 3, we will classify formal synthesis approaches. Section 4 briefly summarizes some formal synthesis approaches. We then look at some criteria for comparing approaches for formal synthesis and conclude.

2 Definition and Delimitation of Formal Synthesis

On examining the synthesis process we observe that we can ascertain its correctness ***before***, ***during*** or ***after*** the synthesis process. Hence we classify the approaches towards formal correctness of synthesis into *pre-synthesis verification*, *formal synthesis* and *post-synthesis verification* (figure 1). All approaches have the same overall aim: proving the correctness of the synthesis output (implementation) with respect to the synthesis input (specification). They all use some kind of logic for deriving proofs within some calculus. However, the time when the logical argumentation takes place differs — in pre-synthesis verification, the proof is derived even *before* synthesis happens, in formal synthesis the proof is formally derived *during* synthesis, and in post-synthesis, the proof is derived after *each* synthesis run.

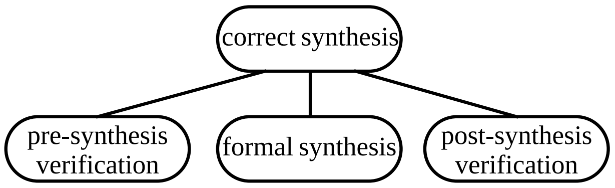

Figure 1. Approaches towards correct synthesis results

Pre-synthesis verification means proving the correctness of the synthesis procedure, i.e. it has to be proven, that the synthesis *program* always produces correct synthesis results with respect to the synthesis input. This is to be done by means of software verification. In pre-synthesis verification the correctness is proven once and for all, whereas in post-synthesis verification, the correctness proof has to be derived for each synthesis run separately. However, software verification is extremely tedious especially for large sized programs such as synthesis tools. Therefore, there is only little work done in this area with respect to hardware.

Formal synthesis means formally deriving the synthesis result within some logical calculus. In conventional synthesis hardware is represented by arbitrary data structures and there are no restrictions as to the transformations on these data structures. In formal synthesis, hardware is represented by means of terms and formulae, and only correctness preserving logical transformations are allowed. Restricting synthesis to only correctness preserving logical transformations guarantees the correctness of the synthesis procedure in an implicit manner. In contrast to conventional synthesis, the result is not only some hardware implementation but also a proof of its correctness, with respect to the specification.

Post-synthesis verification is the most frequently used approach today. In post-synthesis verification the synthesis step is first performed in a conventional manner and then the correctness of the synthesis output with respect to the synthesis input is proved. It is independent of the synthesis procedure.The only information available is the synthesis input and the synthesis output. There is no information on *how* the output was derived from the input. However, general hardware verification techniques are pretty time consuming or even undecidable. This makes verification for large sized circuits tedious or even impossible. [Gupt92] gives an excellent survey of the major trends in post-synthesis verification and we do not discuss them within this paper.

When comparing the above-mentioned approaches towards correct synthesis, one can distinguish between the tool developer’s and the circuit designer’s points of view.

2.1 Circuit Designer’s Viewpoint

For the circuit designer, the characteristics that are relevant are — time taken for synthesis, automation and the requirement of skills in formal methods.

- The *synthesis time* has an additional dependency on the size of the circuit under design and the number of synthesis steps that are performed.
  - With respect to the *size of the design*, the synthesis time is lowest for pre-synthesis verification. Since formal synthesis is restricted to transformations within some calculus, this results in slower synthesis tools. The post-synthesis technique on the other hand suffers from the problem of NP-completeness or undecidability, depending upon the levels of abstraction.
  - The synthesis time with respect to the *number of synthesis steps* is smaller for the pre-synthesis techniques as compared to the formal synthesis approaches, due to the reasons mentioned above. Post-synthesis approaches however, do not depend on the synthesis process and therefore the number of synthesis steps does not have any impact.
- In examining the characteristic of *automation* — it is possible to achieve it at all levels of abstraction in the pre-synthesis approach, since it is possible to write such programs. As far as formal synthesis is concerned, it depends upon the system as seen in section 4. Automation can be achieved only at lower levels of abstraction in the post-synthesis approach via model-checkers and tautology checkers. At higher levels of abstraction, general automation cannot be achieved due to the complexity of the task.
- From the point of view of the circuit designer, the *skills required in formal methods*

is an important factor in its acceptance as a tool. Pre-synthesis verification is the most convenient since the entire logical argumentation is of no concern to the designer. An argumentation similar to the automation holds for the other two approaches.

2.2 Tool Developer’s Viewpoint

The criteria to be examined from the tool developer’s viewpoint are different — the complexity of tool development, the maintenance of the tool when changes are made, and the skills in formal methods required for tool development.

- The *complexity of the development* of pre-synthesis verification is stupendous and hence it is practically impossible to develop such a tool for realistic synthesis programs, since software verification can be performed only for very small programs (≈ 1000 lines of C code). As far as formal synthesis is concerned, the complexity will increase with the size of the synthesis tool. Although post-synthesis verification tools do not depend upon the size of the synthesis tool, they can only be developed without problems at lower levels of abstraction. The development of post-synthesis verification tools at higher levels of abstraction is beyond decidability.
- As regards the *maintenance of the tool with respect to small changes* are concerned, pre-synthesis verification entails a complete reverification in general. The changes in the formal synthesis tools will be minimal and post-synthesis verification needs none, since changes in the synthesis process does not affect it.
- The *logical skills* required in developing such tools are software verification skills for pre-synthesis approach, and hardware verification skills for formal synthesis and post-synthesis approaches.

2.3 Summary

Summarizing the discussion above, pre-synthesis verification is the most acceptable one for circuit designers and least acceptable for tool developers. The situation is the opposite for post-synthesis verification tools. Formal synthesis is a good compromise between these two extremes (Fig. 2).

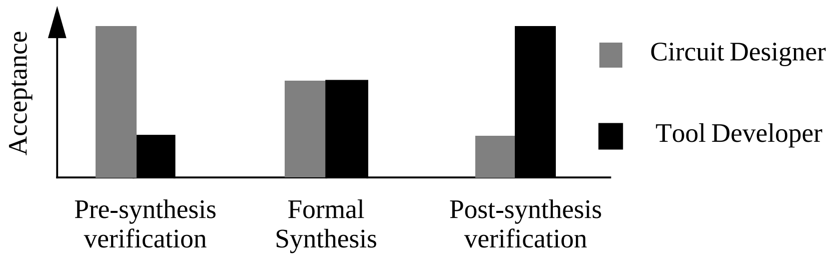

Figure 2. Acceptance of methods for correct synthesis

3 Classification of Formal Synthesis

Formal synthesis can be generally classified into two methods — synthesis step specific verification techniques and transformational derivation (Fig. 3). The former is really a specialized verification technique that has been tuned towards a specific synthesis step. The latter corresponds to a forward derivation of the implementation from a specification within some calculus.

Elaborating further, *transformational derivation* means that the specification will be transformed into an implementation according to a given *core* of elementary transformation rules. This can be further divided into two — transform the specification based on a *hardware-specific calculus* or on a *general purpose calculus*. In a transformational derivation based on hardware-specific calculus, the *core* transformations are circuit transformations whereas in a general purpose calculus they are logical transformations.

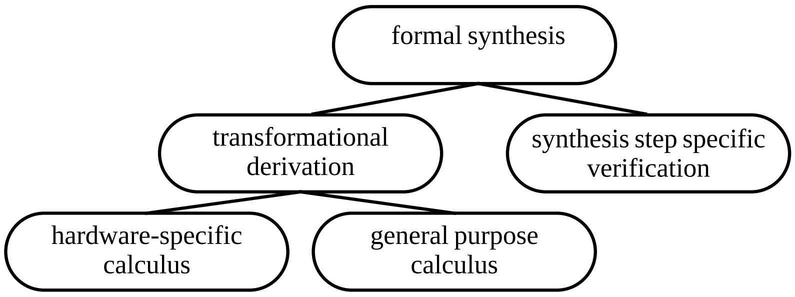
formal synthesis
transformational derivation
synthesis step specific verification
hardware-specific calculus
general purpose calculus

Figure 3. Formal synthesis and related approaches

3.1 Hardware-specific Calculus vs. General-purpose Calculus

In comparing these two approaches for transformational derivation, there are two main criteria — efficiency and security. The efficiency of synthesis is directly determined by the efficiency of circuit transformations, whereas its security is determined by the number and sizes of the core transformations.

The size of the core transformations in the hardware-specific calculus approach is large since the core transformations are the circuit transformations themselves. In contrast, the general purpose calculus approach has a small core of logical transformations. Hence the implementational efforts are high for hardware-specific calculus and low for a general purpose calculus. Additionally, the core transformations are strongly dependent on the application domain and/or the levels of abstraction handled when using a hardware-specific calculus, whereas the core is static for a general purpose calculus. For example, the core transformations for a high-level hardware-specific calculus is completely different from those required for logic synthesis.

Before proceeding further we shall shortly ruminate on the statements in the previous paragraph. It should be noted, that formal derivations within some calculus can only be sound if the transformations in the calculus — a piece of software — are “correct”. Verifying the correctness of the calculus itself means the comparison of the derivation within the calculus with the semantics of its logic. This argumentation itself is to be performed within some “meta logic”. This exercise is one way to increase the reliability of the calculus. An objective correctness criteria cannot be achieved due to the fact, that the meta logical argumentation itself may be error prone. Therefore, the correctness of the calculus is considered to be more of a theoretical activity which has no practical impact on its usage. Nevertheless, there is at least one major requirement — the calculus should be consistent. It should, for example, be guaranteed, that it is not possible to derive a proof for the equivalence of two circuits as well as the proof for their nonequivalence.

For small sized calculi, such errors may easily be figured out, but for calculi with a big set of complex transformations guaranteeing safety becomes a tedious goal. Due to the smaller core, general purpose calculi can be considered to be more reliable. Furthermore, logical (multi-purpose) calculi are based on some well understood mathematical theories, which also increases their reliability as compared to hardware specific calculi. Usually, multi-purpose calculi are also better tested since they have been applied to different domains.

The major drawback of general purpose calculi is their efficiency. For every hardware transformation step, a series of basic mathematical transformations has to be performed. Figuring out the right series of basic mathematical transformations for implementing some specific synthesis step may be demanding. Hardware specific calculi are much better adapted to the synthesis domain. Hardware transformations are directly implemented, which also results in better runtimes.

3.2 Transformational Derivation vs. Synthesis-Step Specific Verification

Besides transformational derivation, one can also perform *synthesis step specific verification* to guarantee correct synthesis. The difference to post-synthesis verification (see section 2) is that the knowledge about *which* synthesis step has been performed, is exploited during verification. However, in contrast to formal synthesis, the knowledge about *how* the synthesis was performed is unknown. Such verification techniques are optimized with respect to a specific synthesis step thus avoiding some disadvantages of post-synthesis verification. Generally, synthesis step specific verification is independent of the heuristic that is applied during a synthesis step. According to its nature, i.e. (a verification technique) only specification and implementation are considered together with the knowledge about the performed synthesis step. A very important point is that synthesis step specific verification is not applicable to all synthesis steps, which can be easily seen for e.g. state encoding step, where the knowledge that the states were encoded cannot prevent one from having to guess the code. Due to this fact, very few research activities have been started in this verification area ([EiJe96], [HuCC96]).

4 Examples

Based on the classification described in section 3, we shall give a brief overview of some systems that have been developed for formal synthesis. We shall first start with an example of pre-synthesis verification – although it does not fall into the category of systems based on formal synthesis, we deem that it deserves a mention here, due to its uniqueness. Later we describe three systems based on HW-specific calculus and general purpose calculus, respectively. The list of references have been organized into general references and specific references for the described systems. Additionally, a set of miscellaneous references have been listed for the two kinds of formal synthesis approaches. However, these have not been referenced in the text and have been included for the sake of completeness only.

4.1 Proven Boolean Simplifier – PBS

The proven boolean simplifier (PBS) is a verified logic synthesis system developed at the Cornell University [AaLe91, AaLe94, AaLe95]. PBS is a part of the high-level synthesis system called BEDROC, which translates a behavioural description given in the HDL - HardwarePal into an implementation suitable for FPGAs [LeAL91, LCAL+93]. Although some abstract formalization have been performed for the scheduling problem within BEDROC, PBS is the only piece of code that has been totally verified.

PBS has been implemented in a purely functional subset of Standard ML and has been verified using the Nuprl proof assistant [Lees92]. PBS consists of about 1000 lines of SML code and implements the weak-division algorithm for logic minimization [BrMc82]. Weak-division performs logic minimization by removing redundant logic. This is achieved by replacing common sub-expressions among the divisors of functions by new intermediate values, thus realizing them only once within the whole circuit.

In proving the correctness of PBS, the subset of SML used in PBS was embedded in Nuprl and the implementation was then interactively verified by proving over 500 theorems. These theorems involve proofs ranging from the correctness of simple functions, such as membership of an element in a list, up to complex theorems that the input combinational function and the minimized function are equivalent. Theorems were also proven regarding the minimality of the weak-division algorithm. The overall effort required was about 8 man months.

Remarks:
Since PBS falls into the category of pre-synthesis verification, we do not compare it with the other approaches as described in section 5. Hence we summarize some important characteristics before moving on to the other examples.

- Since PBS has been verified, this piece of software can be used with an increased degree of confidence.
- PBS is restricted to the use of weak-division algorithm for logic minimization.
- PBS was conceived with the thought of achieving verified software.
- Verification of PBS was eased due to the use of a functional subset of SML for implementation.

To conclude – although PBS is a first step towards achieving correct synthesis, verifying large synthesis programs written in imperative languages is practically impossible.

4.2 RLEXT – Register Level EXploration Tool

RLEXT has been developed at the University of Illinois for the formal synthesis of register-transfer level datapaths [KnWi89, KnWi92]. This tool falls into the category of hardware-specific calculus for formal synthesis.

The hardware specificity arises from the fact that the design, at the register-transfer level, is looked at from different points of view (dataflow, timing and structure). A large set of types and subtypes defines the objects that make up a design and plausibility constraints have been formulated for defining the well-formedness of a design. The overall design is represented by a 4-tuple – dataflow, timing, structure and binding. The dataflow model captures the behaviour of the design and is conceptually viewed as a tri-partite graph which represents the connection of values to parameters and parameters to operations. The timing model gives the abstract states and the state transitions of the design.The structural model gives the schematic of the design. The binding defines the relationships between the behaviour, timing and structure. For each model a set of plausibility constraints is given to determine the well-formedness of the model. Given that the three models (dataflow, timing and structure) are well-formed, another set of constraints is given for the binding relationship to ensure correct relationships between these models.

The entire program which implements these types and plausibility constraints has been implemented in about 20,000 lines of Lisp code. The major drawback of this approach is the large set of axiomatized plausibility constraints. A positive feature of this approach is that RLEXT automatically fixes small bugs in the design.

4.3 RUBY

Ruby is a language based on relations and functions for specifying VLSI circuits. The language and the synthesis tools around Ruby have been developed at the Programming Research Group - Oxford, University of Glasgow and Technical University of Denmark, Lyngby [Josh90, JoSh91a, JoSh91b, ShRa93, ShRa95]. Although Ruby is only a language, we choose to call the formal synthesis approach as Ruby too. This approach is based on the paradigm of hardware-specific calculus for formal synthesis.

The specification is given as a relation between the terminals of the circuit. Simple relations can be combined into more complex ones by a variety of higher-order functions. The typical derivation of an implementation in Ruby involves the use of term-rewriting using equivalences or conditional equivalences. Since constructs in Ruby have a a graphical interpretation (figure 4), abstract floorplans can also be generated.

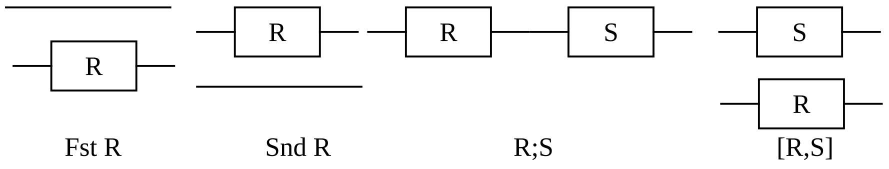

Figure 4. Graphical interpretation of a few constructs in Ruby

“Fst R”, “Snd R” and correspond to the circuit diagrams in figure 4. “R;S” corresponds to a serial connection and [R,S] to a parallel one. Since all these constructs also have a formal basis, it is possible to derive equations such as “Fst R; Snd S = [R,S] = Snd S; Fst R”.

Starting from specifications, it is possible to derive equivalent circuits using the transformations that are available in the system. Examples for some transformations are: parallel compositions, retiming, and conversions of regular structures into other forms, e.g. rows to columns. Additionally, it is possible to derive bit-serial and/or parallel circuits from specifications. During such conversions, additional constraints are generated which have to be satisfied.

4.4 DDD — Digital Design Derivation

The DDD system has been developed at the University of Indiana [John84, JoBB88, JoBo91, BoJo93]. The system is based on functional algebra and falls into the category of hardware-specific calculus for formal synthesis.

Starting from a specification in a lisp-like syntax (a derivative of the functional language *Scheme*), a series of transformations are applied to refine them into an implementation which is also represented in a dialect of the same language. DDD can be thought of as an S-expression editor for hardware purposes, with the objective of finding an implementation that satisfies certain constraints. The behaviour is specified as an iterative system using tail recursive functions. This is translated into a sequential description which can be regarded as a network of simultaneous signal definitions comprising of variables, constants, delays and expressions involving operations. Then a concept called *factorization* is used to abstract away multiple instances of common subexpressions which are allocated to data path operators such as ALUs, adders, etc. Complex factorizations ranging from the abstraction of a register and an incrementer into a counter up to the introduction of co-processors are also possible. Failures in factorizations lead to refinements in control, which are accomplished by serializing transformations on the behavioural descriptions. Although all these transformations are based on functional algebra, none of them have been formally verified.

4.5 LAMBDA/DIALOG

LAMBDA/DIALOG has been developed at Abstract Hardware Limited, Uxbridge [FFFH89, FoMa89, FiFM91, MaFo91, HFFM93]. This is the only commercial tool for performing formal synthesis and belongs to the category of general purpose calculus. This tool has also been used in an industrial environment to perform synthesis, as reported in [BoCZ93, BBCC+95].

The core of LAMBDA/DIALOG is its theorem prover based on higher order logic. The theorem prover is called LAMBDA and the GUI (Graphical User Interface), which is a schematic editor, is called DIALOG. The design state is always represented as a formal logical rule. Starting from the trivial rule — "*the overall specification fulfils the overall specification*", transformations are applied which lead to the rule defining the intermediate stage of design — "*some components and a modified, reduced specification fulfils the overall specification*". The final design corresponds to the rule — "*some components and some timing conditions fulfil the overall specification*". The entire process is facilitated by the use of the GUI - DIALOG. Given a specification in higher-order logic, the input and output ports appear on the schematic window. The user can then either interactively apply basic instantiations of modules in the DIALOG library, or apply compound tactics for automatically performing several such steps. For example, each **if** statement in the specification can be individually refined into some kind of a multiplexer, or all of them can be removed in one go by applying a tactic. During this process, the specification is simplified into a set of subgoals which have to be realized and the design continues until all subgoals have been eliminated. A drawback of this tool is that under certain complex situations, direct interactions at the theorem prover level may be necessary in order to continue with synthesis.

4.6 VERITAS

VERITAS is a theorem prover based on an extension of typed higher order logic and has been developed at the University of Kent [HaLD89, HaDa92]. Due to its core, it is a member of the general purpose calculus category.

In VERITAS, synthesis is a recursive interactive design process. Starting from the specification, transformations, called techniques, are interactively applied for converting the specification into a implementation. A technique consists of a pair of functions called the subgoaling function and the validation function. The subgoaling function generates a (possibly empty) set of design subgoals, theorem subgoals and term subgoals. Each of these subgoals must be discharged during the design process. The validation function, which is a derived inference rule in VERITAS logic, constructs the theorem that the implementation satisfies the specification. There are general purpose techniques for splitting conjunctions, rewriting, stripping an existential quantifier, strengthening a specification, etc. An oft used technique is called *claim* — one can claim that an arbitrary specification is an implementation of the original specification. This leads to two subgoals; the first is to implement the new specification (design subgoal) and the second is to prove that the new specification satisfies the original specification (theorem subgoal).

4.7 HASH (Higher order logic Applied to Synthesis of Hardware)

HASH is under development at the University of Karlsruhe [EiKu95a,EiKu95b, EiBK96] and is based on the theorem prover HOL [GoMe93]. This tool falls into the category of general purpose calculus for formal synthesis.

HASH is a toolbox for implementing formal synthesis programs comprising of circuit transformations that correspond to those of conventional synthesis programs. For example, *scheduling* is a transformation in HASH. This transformation gets the dataflow graph and the schedule (calculated outside the logic using ASAP, ALAP, Force-directed, etc.) and transforms the original dataflow graph into a scheduled dataflow graph [EiKu95b, EiBK96]. In a similar manner there are various transformations provided for each synthesis step at the algorithmic and the RT-level.

The main concept of HASH is to split the synthesis process along two dimensions — *design space exploration* and *design transformation*. The design space exploration aspect determines the quality of the synthesized design with respect to the constraints that have been specified, while the design transformation determines the correctness of the design. Due to this split, the heuristics for exploring the design space can be performed outside the logic and the results imported into the design transformation. This implies that for each synthesis step, e.g. scheduling, state minimization, retiming, etc., there exists a unique transformation which performs that synthesis step. It is to be noted here, that these transformations are specific to a synthesis step but independent of the heuristics. This basic concept is shown in figure 5.

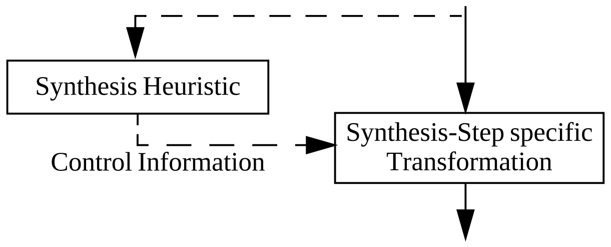

Figure 5. The basic concept of HASH

In this way, the two most important points of circuit design are met independently: the quality and the correctness of the implementation. The quality only depends on the synthesis heuristics involved. The implementations derived within HASH are the same as those achieved with conventional tools. On the other hand, correctness is guaranteed since the circuit transformations are performed inside HOL. From circuit designer's point of view, however, there is no difference between formal synthesis tools based on HASH and conventional synthesis tools. Since all the logical argumentation is performed fully automatically and above all no formal skills are demanded from the user.

5 Comparison

Having presented a brief summary of some systems in the area of formal synthesis, we shall now compare them. We classify the characteristics for comparison into three main criteria — Hardware Representation, Hardware Transformation and Synthesis Control.

5.1 Hardware Representation

This criterion relates to the representation of the specification, the intermediate stages of design and the final implementation. In the systems described in section 4, RLEXT, RUBY and DDD fall into the class of hardware-specific calculus and LAMBDA, VERITAS and HASH are based on general purpose calculus. Additionally, RLEXT is restricted only to RT-level datapaths. The other systems are capable of handling designs at all levels of abstraction, namely from the system level down to the gate level. Due to the strong reliance on the concept of iterative structures, RUBY is mostly suited towards signal processing and module generation applications. All other systems do not have any particular bias towards their application domains. Except RLEXT, all other systems are capable of exploiting hierarchy during the design process.

5.2 Hardware Transformation

This measure compares the conversions used in going from specifications to implementations. Using a theorem prover as a basis has two basic advantages namely, increased reliability and the small number of core transformations. The systems RLEXT, RUBY and DDD are not based on theorem provers. However, due to the use of an algebraic foundation, RUBY and DDD do not have a large number of basic transformations. Before we elaborate this statement, we shall look at the realization of the transformations.

Hardware transformations can be axiomatized or derived from some small set of basic transformations. In the RLEXT system, all the transformations are axiomatized. In RUBY and DDD, some transformations are axiomatized and some of them are derived from these axiomatized transformations based on general algebraic rules. Hence the statement, that these systems do not have a large number of basic transformations. In LAMBDA, VERITAS and HASH all the hardware transformations are derived from the logical transformations provided within the respective theorem prover and hence the size of the core (basic transformations) is small. The user's of the hardware transformations are circuit designers. Hence, we can compare the systems based on the nature of transformations, i.e. are they hardware oriented or logic oriented. RLEXT, LAMBDA and HASH offer transformations which are similar to those used in the conventional synthesis process. The transformations in DDD and VERITAS are more from the logician's point of view. Due to the floorplanning aspect of RUBY, it partially offers a circuit designer's perspective.

5.3 Synthesis Control

This yardstick relates to the use and the practicality of the systems. RLEXT, LAMBDA and HASH provide automation for the circuit designer. Except for RLEXT and HASH, the use of all other systems requires a good knowledge in mathematical notation. Although LAMBDA offers a GUI, there are situations where one has to interact with its theorem prover core. A unique feature of the HASH system is the use of existing heuristics in the synthesis domain and a clear split between the design space exploration task and the actual application of a design transformation.

6 Conclusions

The acceptance of formal methods within the circuit designer community is inversely proportional to the amount of formal knowledge required! The circuit designers would ideally love to have verified implementations for free. Fully automated post-synthesis verification techniques are therefore used wherever possible. However, they are too weak and too time consuming for real sized circuits. For the circuit designer, verified synthesis programs (pre-synthesis verification) would be the best. However, the gap between what can be verified using software verification techniques and the complexity of existing synthesis tools, cannot be bridged today.

In this paper we have presented a summary and classification of formal synthesis techniques. Formal synthesis is constructive in the sense that the proof is derived along with the synthesis process rather than guessed afterwards. Therefore also very expressive logics can be used for formal synthesis. In contrast to post-synthesis verification, the undecidability of a logic is not a handicap. If good tools based on formal synthesis can be developed, hardware verification can be restricted to purely the verification of safety and liveness properties.

Acknowledgements

The authors are grateful to the anonymous referees whose constructive comments have improved the quality of the paper.

References

General

[BrMc82] R.K. Brayton, C. McMullen, “Decomposition and factorization of boolean expressions”, Proc. International Symposium on Circuits and Systems, IEEE Computer Society, 1982

[Evek87] H. Eveking, “Verification, Synthesis and Correctness-Preserving Transformations — Cooperative Approaches to Correct Hardware Design”, From HDL Descriptions to Guaranteed Correct Circuit Designs, D. Borrione (Ed.), Elsevier Science Publishers, North-Holland, 1987

[GoMe93] M. Gordon, T. Melham, “Introduction to HOL: A Theorem Proving Environment for Higher Order Logic”, Cambridge University Press, 1993

[Gupt92] A. Gupta, “Formal Hardware Verification Methods: A Survey”, Formal Methods in System Design, Kluwer Academic Publishers, Vol. 1, 1992, pp.151-238

[Lees92] M. Leeser, “Using Nuprl for the verification and synthesis of hardware”, Phil. Trans. R. Soc. Lond., Vol. 339, 1992, pp. 49-68

[LeAL91] M. Leeser, M. Aagaard, M. Linderman, “The BEDROC High Level Synthesis System”, Proc. ASIC’91, IEEE, 1991

[LCAL+93]M. Leeser, R. Chapman, M. Aagaard, M. Linderman, S. Meier, “High level Synthesis and Generating FPGAs with the BEDROC System”, Journal of VLSI Signal Processing, Kluwer Academic Publishers, Vol. 6, No.2, Aug. 1993, pp.191-214

[McPC90] M. C. McFarland, A.C. Parker, R. Camposano, “The high-level Synthesis of Digital Systems”, Proc. of IEEE, Vol. 78, 1990, pp. 301-318

Pre-Synthesis Verification

[AaLe91] M. Aagaard, M. Leeser, “A Formally verified system for logic synthesis”, Proc. ICCD-91, IEEE, Oct. 1991

[AaLe94] M. Aagaard, M. Leeser, “PBS: Proven Boolean Simplification”, IEEE Trans. on Computer Aided Design, Vol. 13, No.4, Apr. 1994

[AaLe95] M. Aagaard, M. Leeser, “Verifying a logic synthesis algorithm and implementation: A case study in software verification”, IEEE Trans. on Software Engineering, Oct. 1995

Synthesis Step Specific Verification

[EiJe96] C.A.J. van Eijk, J.A.G. Jess, “Exploiting Functional Dependencies in Finite State Machine Verification”, Proc. European Design & Test Conference 1996, Paris, pp. 9-14, IEEE Computer Society Press

[HuCC96] Huang, Cheng, Chen, “On Verifying the Correctness of Retimed Circuits”, Great Lakes Symposium on VLSI 1996, Ames, USA

hardware-specific calculus

DDD

[John84] S.D. Johnson, “Synthesis of Digital Designs from Recursion Equations”, MIT Press, Cambridge, 1984

[JoBB88] S.D. Johnson, B. Bose, C.D. Boyer, “A Tactical Framework for Digital Design”, VLSI Specification, Verification and Synthesis, G. Birtwistle and P. Subrahmanyam, Eds., Kluwer Academic Publishers, Boston, 1988, pp.349-383

[John89] S.D. Johnson, “Manipulation logical organization with system factorizations”, Hardware Specification, Verification and Synthesis: Mathematical Aspects, Leeser and Brown (Eds.), LNCS, Vol. 408, Springer, 1989

[JoWB89] S.D. Johnson, R. Wehrmeister, B. Bose, “On the Interplay of Synthesis and Verification”, Proc. IMEC-IFIP Intl. Workshop on Applied Formal Methods in Correct VLSI Design, L. Claesen (Ed.), North Holland, 1989, pp. 385-404

[JoBo91] S.D. Johnson, B. Bose, “DDD — A System for Mechanized Digital Design Derivation”, ACM/SIGDA Workshop on Formal Methods in VLSI Design, Miami, Florida, Jan. 1991. (Unfortunately, the proceedings of the workshop have not been officially published)

[BoJo93] B. Bose, S.D. Johnson, “DDD-FM9001: Derivation of a verified microprocessor. An exercise in integrating verification with formal derivation”, Proc. IFIP Conf. on Correct Hardware Design and Verification Methods, LNCS, Vol. , Springer, 1993

[RaBJ93] K.Rath, B. Bose, S.D. Johnson, “Derivation of a DRAM Memory Interface by Sequential Decomposition, Proc. Intl. Conf. on Computer Design - ICCD, IEEE, Oct. 1993, pp.438-441

[RaTJ93] K. Rath, M.E. Tuna, S.D. Johnson, “An Introduction to Behaviour Tables”, Tech. Rep. No. 392, Indiana University, Computer Science Department, December, 1993

[ZhJo93] Z. Zhu, S.D. Johnson, “An Example of Interactive Hardware Transformation”, Tech. Rep. No. 383, Indiana University, Computer Science Department, May, 1993

RLEXT

[KnWi89] D.W. Knapp, M. Winslett, “A Formalization of Correctness for Linked Representations of Datapath Hardware”, Proc. IMEC-IFIP Intl. Workshop on Applied Formal Methods in Correct VLSI Design, L. Claesen (Ed.), North Holland, 1989, pp. 1-20

[KnWi92] D.W. Knapp, M. Winslett, “A Prescriptive Model for Data-path Hardware”, IEEE Trans. on Computer Aided Design, Vol. 11, No.2, Feb. 1992

RUBY

[Shee88] M. Sheeran, “Retiming and slowdown in Ruby”, The fusion of hardware design and verification, G.J. Milne (Ed.), North-Holland, 1988, pp. 245-259

[JoSh90] G. Jones, M. Sheeran, “Circuit design in Ruby”, Formal Methods for VLSI design, J. Staunstrup (Ed.), North-Holland, 1990, pp.13-70

[Ross90a] L. Rossen, “Formal Ruby”, Formal Methods for VLSI design, J. Staunstrup (Ed.), North-Holland, 1990, pp. 179-190

[Ross90b] L. Rossen, “Ruby Algebra”, Designing Correct Circuits, Oxford, G. Jones and M. Sheeran (Eds.), Springer, 1990, pp. 297-312

[JoSh91a] G. Jones, M. Sheeran, “Deriving bit-serial circuits in Ruby”, Proc. VLSI-91, Edinburgh, A. Halaas and P.B. Denyer (Eds.), Aug. 1991

[JoSh91b] G. Jones, M. Sheeran, “Relations and refinements in circuit design”, proc. 3rd Refinement Workshop, C.C. Morgan and J.C.P. Woodcock (Eds.), Springer, 1991, pp. 133-152

[ShRa93] R. Sharp, O. Rasmussen, “Transformational Rewriting with Ruby”, Proc. CHDL-93, D. Agnew, L. Claesen and R. Camposano (Eds.), Elsevier Science Publishers, 1993, pp.243-260

[ShRa95] R. Sharp, O. Rasmussen, “The T-Ruby Design System”, Computer Hardware Description Languages and their Applications (CHDL'95), pp. 587-596, 1995

MISCELLANEOUS

[KrSK95] T. Kropf, K. Schneider, R. Kumar, “A Formal Framework for High Level Synthesis”, in TPCD-94, Bad Herrenalb, Germany, R. Kumar and T. Kropf (Eds.), LNCS, Springer, Vol. 901, 1995, pp. 223-238

[GoSS87] G.C. Gopalakrishnan, M.K. Srivas, D.R. Smith, “From Algebraic Specifications to Correct VLSI Circuits”, From HDL Descriptions to guaranteed Correct Design, D. Borrione (Ed.), Elsevier Science Publishers, North-Holland, 1987

[GrRa87] W.Grass, R. Rauscher, “CAMILOD - A Program System for Designing Digital hardware with Proven Correctness”, From HDL Descriptions to guaranteed Correct Design, D. Borrione (Ed.), Elsevier Science Publishers, North-Holland, 1987

[Ueha87] T. Uehara, “Proofs and Synthesis are Cooperative Approaches for Correct Circuit Designs”, From HDL Descriptions to guaranteed Correct Design, D. Borrione (Ed.), Elsevier Science Publishers, North-Holland, 1987

[Camp89] R. Camposano, “Behaviour-preserving Transformations for High-level Synthesis”, Proc. Workshop on hardware Specification, Verification and Synthesis: Mathematical Aspects, Leeser and Brown (Eds.), LNCS, Vol. 408, Springer, 1989

[DrHa89] D. Druinsky, D. Harel, “Using State-Charts for hardware Description and Synthesis, IEEE Trans. on CAD, Vol. 8, No.7, July, 1989, pp.798-807

[GrMT94] W. Grass, M. Mutz, W.D. Tiedemann, “High-level-Synthesis Based on Formal Methods”, Proc. EUROMICRO ‘94, Liverpool, 1994, 83-91

General Purpose calculus

LAMBDA

[FFFH89] S. Finn, M.P. Fourman, M. Francis, R. Harris, “Formal System Design — Interactive Synthesis based on computer-assisted formal reasoning”, Proc. IMEC-IFIP Intl. Workshop on Applied Formal Methods in Correct VLSI Design, L. Claesen (Ed.), North Holland, 1989, pp.97-110

[FoMa89] M.P. Fourman, E.M. Mayger, “Formally based systems design — Interactive hardware scheduling”, VLSI-89, G. Musgrave and U. Lauther (Eds.), Munich, North Holland, 1989

[FiFM91] S. Finn, M.P. Fourman, G. Musgrave, “Interactive Synthesis in higher order logic”, Proc. Intl. Workshop on the HOL Theorem Prover and its applications, Davis, Calif., IEEE Press, 1991

[MaFo91] E.M. Mayger, M.P. Fourman, “Integration of Formal Methods with System Design”, Proc. VLSI-91, Edinburgh, A. Halaas and P.B. Denyer (Eds.), Aug. 1991

[BoCZ93] M. Bombana, P. Cavalloro, G. Zaza, “Specification and Formal Synthesis of Digital Circuits”, Proc. Higher Order logic Theorem Proving and its Applications, L.J.M. Claesen and M.J.C. Gordon (Eds.), Elsevier Publishers, Vol. A-20, 1993, pp.475-484

[HFFM93] R.B. Hughes, M.D. Francis, S.P. Finn, G. Musgrave, “Formal Tools for Tri-State Design in Busses”, Proc. Higher Order logic Theorem Proving and its Applications, L.J.M. Claesen and M.J.C. Gordon (Eds.), Elsevier Publishers, Vol. A-20, 1993, pp.459-474

[BBCC+95]G. Bezzi, M. Bombana, P. Cavalloro, S. Conigliaro, G. Zaza, “Quantitative evaluation of formal based synthesis in ASIC Design”, Proc. TPCD’94, R. Kumar and T. Kropf (Eds.), Bad Herrenalb, Germany, LNCS, Vol. 901, Springer, 1995, pp. 286-291

VERITAS

[HaLD89] F.K. Hanna, M. Longley, N. Daeche, “Formal Synthesis of Digital Systems”, Proc. IMEC-IFIP Intl. Workshop on Applied Formal Methods in Correct VLSI Design, L. Claesen (Ed.), North Holland, 1989, pp.532-548

[HaDa92] F.K. Hanna, N. Daeche, “Dependent types and formal synthesis”, Phil. Trans. R. Soc. Lond., Vol. 339, 1992, pp. 121-135

HASH

[EiKu95a] D. Eisenbiegler, R. Kumar “An Automata Theory Dedicated towards Formal Circuit Synthesis”, Higher Order Logic Theorem Proving and Its Applications HUG’95, Aspen Grove, Utah, USA, September 11-14, 1995

[EiKu95b] D. Eisenbiegler, R. Kumar, “Formally embedding existing high level synthesis algorithms”, Proc. CHARME’95, Springer, LNCS, Vol. 987, pp. 71-83

[EiBK96] D. Eisenbiegler, C. Blumenröhr, R. Kumar, “Implementation Issues about the Embedding of Existing High Level Synthesis Algorithms in HOL”, International Conference on Theorem Proving in Higher Order Logics, TPHOL’96, Turku, Finland, August 27 - 30, 1996

MISCELLANEOUS

[BaBL89] D.A. Basin, G.M. Brown, M. Leeser, “Formally Verified Synthesis of Combinational CMOS Circuits”, “Formal Synthesis of Digital Systems”, Proc. IMEC-IFIP Intl. Workshop on Applied Formal Methods in Correct VLSI Design, L. Claesen (Ed.), North Holland, 1989, pp. 251-260

[Basi91] D. Basin, “Extracting circuits from Constructive Proofs”, ACM/SIGDA Workshop on Formal Methods in VLSI Design, Miami, Florida, Jan. 1991. (Unfortunately, the proceedings of the workshop have not been officially published)

[Busc92] H. Busch, “Transformational Design in a Theorem Prover”, Theorem Provers in Circuit Design, V. Stravidou, T.F. Melham and R.T. Boute (Eds.), Elsevier Science Publishers, North-Holland, 1992, pp. 175-196

[Lars94] M. Larsson, “An Engineering Approach to Formal System Design”, in Higher Order Logic Theorem Proving and Its Applications, T.F. Melham and J. Camilleri (Eds.), pp. 300-315, Valetta, Malta, September 1994, Springer

[Lars95] M. Larsson, “An Engineering Approach to Formal Digital System Design”, The Computer Journal, Oxford University Press, Vol. 38, No.2, 1995, pp. 101-110

[SGGH92] J.B. Saxe, S.J. Garland, J.V. Guttag, J.J. Horning, “Using Transformations and Verification in Circuit Design”, Designing Correct Circuits, J. Staunstrup and R. Sharp (Eds.), Elsevier Science Publishers, North-Holland, 1992

[Wang93] L. Wang, “Deriving a Correct Computer”, Proc. Higher Order logic Theorem Proving and its Applications, L.J.M. Claesen and M.J.C. Gordon (Eds.), Elsevier Publishers, Vol. A-20, 1992, pp. 449-458