ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 4, Issue 12, December 2016

A Study on Self Verifying Compilers for Synthesizing Behavioural Style Code Scripts

Harsh J Sitapra

Visiting Professor, Dept. of Electrical Engineering, Tolani F. G. Polytechnic, Adipur, Kachchh, Gujarat, India

ABSTRACT: The chief interest of this distinction lies in the fact that structural code block is considered easier to implement. The efficient implementation of behavioral languages imposes a formidable challenge. Behavioral code block has not been so clearly tackled yet, essentially because it touches aspects governing the semantics of programs. To make our point widely applicable, we avoid fine grain technicalities. Rather we emphasize the common denominator of all we propose a clear and general problem statement, and we set up a wide-ranging research agenda. When confronted with behavioral code block, to which most language implementers adopt interpretive techniques. Unfortunately, the lack of a precise understanding of the issues involved in this evolution pushes implementers to limit the behavioral models. The use of hardware compilers to generate complex circuits from a high-level description is becoming more and more prevalent in a variety of application areas. These proposals have been classified generically as Model-Driven Engineering (MDE) and share common concepts and terms that need to be abstracted, discussed and understood. However, this introduces further risks as the compilation process may introduce errors in otherwise correct high-level descriptions of circuits. Next, we suggest new avenues towards more efficient implementation techniques for full behavioral code block. This paper presents a survey on MDE based on a unified conceptual model for synthesis that clearly identifies and relates these essential concepts, namely the concepts of system, model, modeling language, transformations, software platform, and software product in general.

KEYWORDS: Model driven synthesis, Model-driven engineering, Behavioral silicon model, Self verifying compilers, Algorithmic synthesis.

I. INTRODUCTION

Behavioral data flow modelling is used to describe the behavior of digital circuits. Designer describes the functionality of design by writing algorithm kind of code. The goals of this exercise are: (1) to give the picture of the state of the art in the efficient implementation of behavioral domain, (2) to review the main issues in going from interpreter-based to compiler-based implementations and (3) to propose new avenues towards the realization of this objective.

A model is an abstraction of a system often used to replace the system under study. In general a model represents a partial and simplified view of a system. That means that our models are thus language-based in nature and tend to describe or prescribe some system as opposed, for example, to models in Mathematics which are understood as interpretation of a theory. An implementation of a programming language is the realization of its syntax and semantics, which comprises a translator and a run-time system.

First, through the relationship between Model and Modeling Language, a modeling language is a set of models (or a model is an element of a modeling language). So, the success of a modeling language will depend on the right balance between simplicity and expressiveness, and in particular the following concerns should be addressed when designing a concrete syntax: writability, readability, learnability and effectiveness. The effective computation is obtained by translating the program to a suitable low-level sequence of actions to be executed on a computer in conjunction with the language run-time system. MDE is a relatively new engineering approach with some expectations and challenges to be addressed in the next years.

As discussed throughout this paper there are some concrete proposals and many more tools and platforms that, in some way would achieve that general vision and relevance. For an effective implementation of MDE several features should be supported and integrated into appropriate tools, typically classified as IDEs. Our effort here is intended to simplify behavioral modeling and to improve synthesis accuracy and efficiency. Experienced engineers and representatives of EDA vendors have wrestled to define enhancements that will offer increased design productivity, enhanced synthesis capability and improved verification efficiency. The guiding principles behind proposed enhancements included: (1) do not break existing designs, (2) do not impact simulator performance, (3) make the language more powerful and easier to use.

EDA vendors opposed this enhancement because of the perceived difficulty in efficiently implementing this enhancement, and its potential impact on compile-time performance. The size, and level of complexity of hardware has increased dramatically over these past years. This has led to the acceptance of high-level hardware synthesis | allowing the compilation of program-like descriptions into hardware circuits. As in the case of software compilers, correctness of synthesis tools is crucial.

The high-level synthesis languages provide a number of complex composition operators. The interaction of which can be difficult to understand and ensure the correctness of. The verification of the synthesis procedures proved to be tedious, and in some cases very difficult to demonstrate. Compositional compilation techniques are usually verified using structural induction over the language constructs, the individual cases of which usually turn out to be of a finite nature. The circuit is then passed onto the model-checker to ensure that it outputs a valid output or not as intended by the behavioral code. A generic circuit, working for any number of inputs, can then be defined by recursion over the structure of this list. Again, we use pattern matching and recursion to define the circuit.

II. Related Work

In [2] “Survey of HDL Compiler Optimization Techniques”, by M.Joseph, Narasimha B.Bhat and K.Chandra Sekaran, International Journal of Information Processing Vol. 1, No. 1, March/April 2007, the work presents that the hardware description languages adhere to a simple, sequential programming style, which mimics the HLL programming model. They are not capable of expressing the synchronous, con-current processing nature of the hardware circuits. This gap between the HDLs and hardware circuitry is called semantic gap. This problem leads to sub optimal design of digital systems.

This paper presents HDL compiler optimization techniques comprehensively and also some open issues. General approach uses a typical syntax-directed translation + global optimization + local optimization + code generation method. It does not address the implementation issues of these techniques. VLSI design as a whole can be thought as an optimization problem but recent findings up to this presents only compiler transformations not the other issues like gap between the CAD tools and target technologies, which is a major cause for suboptimal hardware generation as mentioned.

III. Compiling Expressions Into Hardware Circuits

It is quite easy to generate a circuit which accept only input strings which are admitted by a given regular expression. The circuits we generate will have one input start and one output match: when start is set to high, the circuit will start sampling the signals and set match to high when the sequence of signals from a received start signal until just before the current time is included in the language represented by the regular expression. A synthesis procedure is nothing but a function from a program to a circuit: requirement here is we managed to prove that the compilation procedure satisfies standard axioms of regular expressions, hence effectively verifying the compiler.

One weakness with the above proof, is that if the sub-circuits `break' for some time but then resume to work correctly, the top-level circuit is expected to resume correctly. This is a strong property which compilation procedures which encode some form of state usually fail to satisfy. To strengthen induction to deal with this adequately we need to add temporal induction assuming that the sub-components always worked correctly, the top-level component works as expected. Finite state model-checkers can be used to verify properties of hardware compilers. Broadly speaking, the history of software development is the history of ever late binding time . . . "

Pushing this idea to its limit by postponing the binding time of at least part of the language syntax, semantics and implementation as well as the program itself, to the run-time. In the extreme case, this possibility challenges our capability to efficiently implement languages (i.e., compile them). But if behavioral code indeed includes such an extreme possibility, we claim that most of the time, typical programs will behave in such a way that most of their code

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 4, Issue 12, December 2016

can be compiled using standard techniques, or at least techniques that are within the reach of the current research on the implementation of modern dynamic programming languages. When the translation is done on-line, expression by expression, and interleaved with the actual computation, the implementation is said to be an interpreter for the language; when it is done all at once, usually prior to the execution, the implementation is said to be a compiler for the language. We consider that language designers must give as much freedom as possible to programmers in terms of late-binding, but their implementers must use appropriate techniques to extract static computations from programs in order to compile and optimize them prior to run-time.

Our goal is to precisely characterize the static case in order to enable the development of new compiler tools and techniques to tackle static nature, even in languages allowing dynamic behavioral nature. Behavioral style of coding language designers and implementers will then face two possible paths. Indeed, one of them is to provide highly efficient languages with only static binding. But we also propose that dynamic behavioral late binding can be introduced, at the price of dynamic compilation.

IV. Demands For Behavioural Code Compilation

The pursuit of ever late binding time had a great impact on the history of software development in general but also of programming languages in particular. The underlying principle is that all options should be kept open until the last possible moment. Binding means translating an expression in a program into a form immediately interpretable by the machine on which the program is run; binding time is the moment at which this translation is done. This restrictive definition has the advantage of being very clear. An example of a binding is the one of a variable name to a memory location, which happens at program design time in machine language (and in early programming languages) but has been postponed to compile time or run-time in modern programming languages. Another example is the binding in a procedure call of the procedure name to the address of the code to be run. In procedural languages, this binding is done at compile time while in object-oriented languages it is postponed to the run-time. The class of binding times includes the design, specification or implementation of the programming language, or the design, specification, implementation, translation or running of the program. Furthermore, we distinguish formal binding times from actual ones.

A formal binding time is the latest moment at which a binding can occur in general, while an actual binding time is the actual moment at which the binding occurs for a particular element of a particular program. The notion of binding time is crucial to understand, compare and contrast the design and implementation of programming languages. Later binding times introduce more flexibility, at the expense of transferring the costs of the bindings towards later stages of processing. The general trend in the evolution of programming languages has been to postpone formal binding times towards the running of programs, but to use more and more sophisticated analyses and implementation techniques to bring actual binding times back to the earlier stages, where the costs of bindings are less harmful. Behavioral coding style relays on postponing the binding of almost all elements of programs and languages to the run-time.

However, in the rest of this paper, we will claim that if formal binding times are postponed to run-time, the research should focus on new tools and implementation techniques aiming at bringing the actual binding times back to compile time. Static information is the mainspring of compilation. The notion of statically known information in programs is central to compilers. Compilers serve two main purposes. First, they translate a computation expressed in a high-level source language (or model of computation) into an equivalent computation expressed in a (usually) lower-level language. The lower-level language is usually more efficient because it is directly implemented in hardware. Second, compilers process the static semantics of the computation, that is code that can be executed at compile time because it is known and it processes statically known data (it typically ranges from compile time checks for contextual properties to constant folding).

In particular, it is perfectly conceivable, and even legitimate to be able to dynamically change the target language. This would be interesting for example when processes are migrated across a network of perhaps statically unpredictable heterogeneous processors, a typical application of behavioural reflection. Unless we want to write programs to produce hardware implementation of high-level programming languages, acceptable performance can only be achieved by translation of programs into assembly code. Hardware compilers (also known as syntheses tools) are compilers whose output is a description of the hardware configuration instead of a sequence of instructions. The output of these compilers target computer hardware at a very low level, for example a field-programmable gate array (FPGA) or

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 4, Issue 12, December 2016

structured application-specific integrated circuit (ASIC). Such compilers are said to be hardware compilers, because the source code they compile effectively controls the final configuration of the hardware and how it operates. The output of the compilation is only an interconnection of transistors or lookup tables. Compilers bridge source programs in high-level languages with the underlying hardware.

A more insightful answer is expressed in the following hypothesis. The major point here is that languages with static behavioural code can be both powerful and efficient. Moreover, in a language allowing dynamic behavioral program code binding can still exhibit static computations that can be compiled. The above hypothesis is purposely vague about what exactly we mean by “information" and “known at compile time". Both heavily depend on the language itself. This in turn depends on a lot of information about its processor that must be determined even at language design time, or at least at its implementation time. Open compilers are essentially compilers that are refined and that can be modified from within the language. All synchronous digital machines can be reduced FSM model. A finite state machine (FSM) consists of memory elements to hold the machine’s current state and combinational logic, specified as Boolean logic functions, to compute the next state and outputs.

A machine can be described as a single FSM or a collection of communicating FSMS. The target of a register-transfer compiler is a set of logic gates and memory elements; its job is to compile the most efficient hardware for the architecture. A register transfer compiler can be used as the back-end of an architectural compiler. A compile time MOP (meet over paths) is the protocol provided to the user for implementing modifications. Despite their name, compile time MOPs do not imply in our view that everything is done prior to program execution, but rather leaves the door open to dynamic compilation.

The first projection says that partially evaluating an interpreter with respect to a known program will do function of compiler" the program by removing the level of interpretation. The second says that partially evaluating the partial evaluator itself with respect to a specific interpreter yields a compiler". There are two major ways to know that something is static in a program: syntactically or by a binding-time analysis. For example, the syntax of applications in first-order functional languages forces the applied function to be known at compile-time. In higher-order languages, the applied function is the result of an expression, but a closure analysis can show that, in particular cases, the result of this expression is in fact known at compile time.

In general, syntactic staticness is much easier to deal with than the one brought to the fore by analysis, but it is also more restrictive. When everything is known statically, the result of the whole computation can be obtained at compile time, but this ideal case is indeed very rare. When code written in a language is compiled, its syntax is transformed into an executable form before running. There are two types of compilation. Machine code generation: Some compilers compile source code directly into machine code. This is the original mode of compilation, and languages that are directly and completely transformed to machine-native code in this way may be called "truly compiled" languages. Intermediate representations: When code written in a language is compiled to an intermediate representation, that representation can be optimized or saved for later execution without the need to re-read the source file. When the intermediate representation is saved, it may be in a form such as byte code. The intermediate representation must then be interpreted or further compiled to execute it. Virtual machines that execute byte code directly or transform it further into machine code have blurred the once clear distinction between intermediate representations and truly compiled languages.

Note that languages are not strictly "interpreted" languages or "compiled" languages. Rather, implementations of language behaviour use interpretation or compilation. When both the interpreter and the method are known, we can partially evaluate the interpreter with respect to the method or statically generate a compiler from the interpreter and compile the method. Where only the interpreter is known, we can generate the compiler statically, but this compiler will have to be used at run-time to compile methods. Interpretation does not replace compilation completely. It only hides it from the user and makes it gradual. In the last and most challenging case, the interpreter is not known until run-time, which means that even the generation of a compiler must be done at run-time. They are least upper bounds because specific apply methods may necessitate much simpler implementation techniques.

A hybrid of an interpreter and a compiler will compile the statement into machine code and execute that; the machine code is then discarded, to be interpreted anew if the line is executed again. Interpreters are commonly the simplest implementations of the behaviour of a language. Splitting a compiler up into small programs is a technique used by researchers interested in producing provably correct compilers. Proving the correctness of a set of small programs often requires less effort than proving the correctness of a larger, single, equivalent program. We have noticed that it takes great care to write interpreters and programs that perform well under partial evaluation. A deep knowledge of partial evaluation and of the particular partial evaluator under use is needed to achieve the goal of compiling away the levels of meta-interpretation. Part of the problem also comes from immoderate expectations put in partial evaluation. Partial evaluation performs a kind of compilation but has pointed out that programs resulting from partial evaluation essentially stick to the implementation choices made by the interpreter. This late-binding of the language semantics favors interpretive techniques, but compilers are absolutely necessary to make the languages efficient and therefore of real interest.

V. RESEARCH DIRECTIONS: SILICON CODE GENERATION

In this section, we first look at emerging techniques from the implementation of current advanced programming languages. Architectural compilers design a machine’s structure, paying relatively little attention to the physical design of transistors and wires on the chip. Silicon compilers concentrate on the chip’s physical design. The crux of the argument is that, although it costs something to run the compiler at run-time, run-time code generation can sometimes produce code that is enough faster to pay back the dynamic compile costs". The portion of our intended system that makes the most innovative use of compiler methods is the silicon code generator. Our circuits are implemented as a library of registers and gates that implement a subset of the Boolean functions-NAND, NOR, AND-OR-INVERT (AOI), etc. This library forms the silicon instruction set-it is an exact analog of a machine’s instruction set. Open compilers propose a methodical development of this technique.

Adaptative run-time systems: families of compilers often share an important part of their run-time systems. This kind of reuse does not only simplify the maintenance of compilers, it also paves the way to runtime adaptiveness of compilers to applications. The silicon code generator’s job is to translate the intermediate form produced by the global optimizer into the best silicon code-networks of the gates and registers available in the library. At the end of code generation the intermediate form has been compiled into an equivalent piece of hardware built from the available library components. Binding-time analysis: the ability to statically determine the binding time of the value of each expression in a program, and to segregate among those known at compile time (static) from those known only at run-time (dynamic).

While these programs may take a high-level description of the chip’s function, they map that description onto a restricted architecture and do little architectural optimization. Silicon compilers assemble a chip from blocks that implement the architecture’s functional units. They must create custom versions of the function blocks, place the blocks, and route wires between them to implement the architecture. Other static analyses: a lot of interest is currently raised by static analyses in general. Besides the constraint propagation approach developed in standard compiler technique and for object-oriented languages, abstract interpretation is now rapidly developing in functional and logic programming. We want an efficient implementation of the matching program, but we also want to be able to quickly change the matching program as we tune the instruction set. For example, if the interpreter uses a-lists to implement environments, partially evaluating it with respect to a program will yield a “compiled" program that still uses a-lists for its environments. Architectural compilers display a number of relatively direct applications of programming-language compilation techniques. Dynamic behavioral compilation happens when modifications depends directly or indirectly upon unknown inputs to the program. It doesn't mean however that no compilation can occur. Dynamic compilation techniques are already used in several languages.

The problem, as stated earlier, is that the compilers will have to be generated dynamically, currently a major drawback. The modularity afforded by describing the code generator as patterns, actions, and cost functions makes it easy to port our code generator to a new hardware library, just as code generators can be ported to new instruction sets. While this method, and its successors, has promise, we have found two limitations in its application to hardware. When compiling to a fixed architecture, resources are free up to the point of saturation and infinitely expensive after that. The cost of developing and maintaining the instruction set must be weighed against the improvement in the speed and area of hardware that can be compiled. This fact leads to the following observation. In the same way standard languages face a compromise between the costs of compiling versus interpreting, which depends on the number of times a portion of code is executed, languages with dynamic behavioral reflection face a compromise between compiling versus interpreting a portion of code that depends on the number of times it is executed between successive modifications that renders its recompilation mandatory.

Many compilation techniques developed for programming languages are applicable to compilation of register-transfer hardware designs. We call our system a hardware compiler because its strategy for generating hardware is very

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 4, Issue 12, December 2016

similar to the strategy used to compile programming languages: syntax-directed translation of the source into a simple hardware object followed by optimization to improve the hardware. In this perspective, we can foresee systems that mix interpretation and compilation, where it would be possible to compile part of the application and suspend the compilation when the necessary information will be known only at run-time. Suspended compilation will then be incrementally performed during run-time, perhaps depending on heuristics balancing the expected gain in performance versus the expected cost of performing this compilation. Hardware compilers produce a hardware implementation from some specification of the hardware. There are, however, a number of types of hardware compilers which start from different sources and compile into different targets. These systems jointly design the microcode to implement the given program and the datapath on which the microcode is to be executed.

For example, applications running for weeks on networks of heterogeneous computers may put a higher preference on the possibility to generate a new compiler to migrate processes towards newly added nodes than pure performance. Also, as incremental compiler generation will enhance, the balance between flexibility and performance will militate in favor of more practical applications. Register-transfer compilers, in contrast, implement an architecture: they take as input a description of the machine's memory elements and the data computation and transfer between those memory elements; they produce an optimized version of the hardware that implements the architecture. Our register-transfer compiler compiles an FSM description into a machine: logic gates implement the combinational logic functions and registers implement the memory elements. As for language compilation, one machine can be compiled into many different configurations of logic gates and registers. The compiler's job is to compile the FSM into hardware that is small and fast.

VI. CONCLUSION AND FUTURE WORK PERSPECTIVES

Most of the implementations tend to rely on interpretive techniques, which are easier to implement and by design more reactive to modifications. However, compiling is mandatory to get efficient languages. The compiler represents the design as a graph; successive compilation steps transform the graph to increase the level of detail in the design and to perform optimizations. Compilers can either be constructed by hand or generated from some specification of the language syntax and semantics. Modifications can then be made either directly in the code of the compiler or by altering the specification and automatically generating a new compiler. In the field of compiler generation, the kind of specications that have been used ranges from formal semantics (denotational, action or modular monadic semantics) to interpreters, either meta-circular or written in a low-level (assembly) language.

The compiler itself can also be viewed as a low-level specification of the language semantics. Translation of the source into Boolean gates plus registers is syntax-directed, much as in programming language compilers, but with some differences because the target is a purely structural description of the hardware. Unfortunately, a tension exists between high level specifications, that eases the modifications, and the implementation, which needs a low-level assembler model to be run efficiently. The two can hardly coincide because low-level models are too detailed and tend not to be modular.

We add the caveat that while programming-language compilation techniques for fixed architectures can often give us a good start on many problems in hardware compilation it is important to remember that language-to-hardware compilations are harder precisely because the hardware architecture is not fixed. An alternative is to use high-level models oriented towards the user but to automatically propagate the modifications from the high-level specification to the low-level model. A good software compiler must optimize its code for the target of its code. A good hardware compiler must be able optimize its code for a target architecture, but must also explore resource tradeoffs across a potentially wide range of such targets.

We hope that hardware and programming-language compilation techniques become even more closely related as we discover more general ways to compile software descriptions into custom hardware. For the moment we continue to experiment with and extend our hardware compiler technology to make it more general, while still retaining its ability to create industrial-quality designs.

REFERENCES

1. G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529-551, Apr. 1955.
2. Survey of HDL Compiler Optimization Techniques, by M.Joseph, Narasimha B.Bhat and K.Chandra Sekaran, International Journal of Information Processing Vol. 1, No. 1, March/April 2007.
3. IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Std P1364/D5.
4. Douglas L. Perry, VHDL, McGraw-Hill, Inc., 1994, p. 1.
5. Don Mills and Clifford E. Cummings, "RTL Coding Styles That Yield Simulation and Synthesis Mismatches," SNUG'99 (Synopsys Users Group San Jose, CA, 1999) Proceedings, section-TA2 (1st paper), March 1999.
6. Koen Claessen and Gordon J. Pace. "An embedded language framework for hardware compilation. In Designing Correct Circuits" '02, Grenoble, France, 2002.
7. "Synopsys FPGA Synthesis Reference Manual", Synopsys, December 2012, https://solvnet.synopsys.com.
8. Sutherland, Davidmann and Flake, "SystemVerilog for Design Engineers, second edition", Springer, Boston, Massachusetts. Copyright 2006. ISBN: 978-0-387-36495-7.
9. "HDL Compiler™ for SystemVerilog User Guide, Version G-2012.06-SP4", Synopsys, December 2012.
10. D.Weiss and C. Lai. Software ProductLine Engineering. AddisonWesley, 1999.
11. U. Asmann. Invasive Software Composition. SpringerVerlag Heidelberg, February 2003.
12. Kenneth L. Short, VHDL for Engineers, Pearson, 2009.
13. Dave Landis, Ph.D., P.E., Programmable Logic and Application Specific Integrated Circuits.
14. S. Gupta, N. Dutt, R. Gupta, and A. Nicola. SPARK : A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations. In *International Conference on VLSI Design*, New Delhi, India, January 2003.
15. J. Guy L. Steele. Rabbit: A Compiler for Scheme. Technical report, Cambridge, MA, USA, 1978.
16. S. D. Johnson. Formal derivation of a scheme computer. Technical Report Technical Report 544, Indiana University Computer Science Department, September 2000.
17. K. Compton, S. Hauck, "Configurable Computing: A Survey of Systems and Software", *Northwestern University, Dept. of ECE Technical Report*, 1999.

BIOGRAPHY

Harsh J Sitapra is a Researcher at the Department of Electrical Engineering, Tolani Foundation Gandhidham Polytechnic Adipur, Kachchh, Gujarat, India. He received Master in Engineering (ME) degree in 2011 from Faculty of Technology and Engineering, Maharaja Sayajirao University of Baroda, Vadodara, Gujarat, India. His research interests are Computer Architecture, VLIW processors, DSP core optimizations, Vector processors, SIMD platforms, etc.