Design Validation by Symbolic Simulation and Equivalence Checking: A Case Study in Memory Optimization for Image Manipulation

Kong Woei Susanto, Tim Todman, Jose Gabriel Coutinho, and Wayne Luk

Department of Computing, Imperial College London,
180 Queen's Gate, London SW7 2BZ, United Kingdom
{kws,tjt97,jgfc,wl}@imperial.ac.uk

Abstract. Design optimization exploration is a key element in finding an optimal resource utilization. The exploration process applies optimizations iteratively; after applying each optimization, the result has to be validated. The research challenge for formal verification is to develop an efficient design validation flow and increase the quality of the validation. In this paper, we propose an automated validation flow to check the functional equivalence of the source design and its optimized version. This approach is based on a symbolic simulation technique to obtain the design properties and automatically check them using an equivalence checker. The novelty of this approach includes the use of model simplification techniques, such as if-conversion and loop-conversion, and state encoding to ease validation analysis.

1 Introduction

The rapid advancement of digital imaging technology has led to the availability of high resolution digital images. A wave of new applications has emerged in many fields, such as in photography, entertainment, medicine, and surveying. A RAW image taken with a Canon EOS-1Ds Mk-III [2] contains more than 14M (5616 by 3744) pixels, each of 14 bit color depth. Manipulating such high resolution data uses a lot of resources and time. One of the challenges is to develop a system that can be used to optimize designs so that they can utilize limited resources in the most efficient ways. We address this first challenge by developing a framework where a set of rules is used to optimize the designs [21,23]. Using the design framework, designers can explore various optimizations by mixing and matching the rules to find the combination that best uses the available resources. A new challenge emerges: how we ensure that the optimized designs will perform the intended tasks.

The verification of design optimization tools for large-scale designs is still a research challenge. As a consequence, designers for such designs will have to re-validate the target design to ensure that it still preserves the same functional behavior as its source; this validation process will have to be done every time the designers optimize the source. It is performed iteratively during the optimization

M. Nielsen et al. (Eds.): SOFSEM 2009, LNCS 5404, pp. 509–520, 2009.
© Springer-Verlag Berlin Heidelberg 2009

exploration stage to find the optimum balance. The continuous need to revalidate the target design inspires us to automate the design validation processes.

Validating design optimization equivalence is an undecidable problem [13]. It is a difficult proposition to have a complete formal equivalence checker for all application domains. However, a formal verification framework for a specific application domain can be defined. Specific information about the target domain can be exploited and the problem domain can be limited. One approach is not to verify the full semantic correctness, but only to verify the functional equivalence of the source and target designs.

In this paper we present our approach for design optimization validation which is based on a combination of symbolic simulation and equivalence checking techniques. The key features are the use of a variable table for each model and model simplification techniques such as if-conversion and loop-conversion. The system automatically generates formal models for each pair of source and target designs. The models are analyzed using a symbolic simulation technique. A decision procedure is used to validate the functional equivalence of the symbolic simulation results between the source and target designs. We demonstrate our validation approach with case studies in the optimization exploration of an image manipulation algorithm. In comparison to validation by simulation, validation by symbolic simulation has a higher degree of assurance. Validation based on symbolic constants eliminates the need of test-cases. As the result, the validation coverage is higher than the numerical simulation-based methods.

The rest of the paper is organized as follows. Section 2 discusses related work. The design validation methodology is discussed in section 3. We present the formal modeling approach in section 4, followed by a discussion of our experiments and case studies in section 5. Finally in section 6, we present the conclusion and the future work of this paper.

2 Related Work

Although industrial formal verification tools have a relatively short history [8,9], program transformation has been an active research area for more than four decades [4]. There are two common approaches used in this verification domain. The first approach is to verify the correctness of the transformation compiler [7,10,15,16,20]. This approach guarantees that the compiler will always produce a correct transformation. However, most formal verification of the compiler only targets at algorithmic level, not the actual compiler code itself. Hence it cannot guarantee that its implementation will be error free. We have learnt from our previous work [20], that this approach is not suitable for our current dynamic setting. The second approach is to check that the target design is semantically equivalent to the source [3,11,13,14,18,25]. The drawback of this approach is that it will add substantial run time to the optimization. Singh and Lillieroth [18] demonstrated an approach to automatically validating the functional equivalence of a design synthesis result with its source using Prover tools [22]. They minimize the run time validation penalty by targeting a

specific application domain. We adopt a similar approach in this paper and focus our work on loop optimizations. The validation provides the assurance that the functional behavior of an efficient but possibly non-obvious design is the same as the one which is obvious (and already proven correct) but possibly inefficient.

The work closest to ours is by Saito et al. [17] and by Siegel et al. [19]. Both use a combination of symbolic simulation techniques to analyze the system and an equivalence checker either by a decision procedure or a model checker. Saito et. al. validate the equivalence of two C descriptions by identifying their cut points of textual differences and selecting the functions which have the textual differences. The cut points localize the scope of the analysis for the validation. Using this approach, they are able to validate two large C descriptions. One key element to the success of their approach is that their target C descriptions are close. Siegel et.al. validate the equivalence of parallel C code with respect to its sequential one. The approach involves an expression table which is constructed during the sequential evaluation. The uniqueness of this approach is that because of the way the transformation is captured, any resulting expressions from simulating the parallel code already existed in the table.

Similar to Saito and Siegel, our approach is also based on a combination of symbolic simulation and equivalence checker. Our application domains differ from the ones explored by Saito and Seigel. In our work, the source and target codes only have the same functional behavior but the codes are not close or similar as in Saito's. The cut point for our codes will be the entire code. Our system covers more language features, such as arrays and loops, than Saito's work. The regularity expected in Siegel's approach does not exist in ours. Almost all our optimizations introduced new variables. The internal operations of the optimized code which depend on these new variables may not have the consistency expected by Siegel's approach.

3 Methodology

The correctness of design optimization depends on the optimized design having behavior equivalent to the original source. Our approach is to assure the correct functional behavior of the system which is investigated by equivalence checking in conjunction with the symbolic simulation technique. We assume that the source design has been verified. The optimization validation assures that the optimization process does not change the functional behavior of the optimized design, hence it maintains the correct behaviour of the source.

The core of our system is a library of functions for automating translations and evaluating symbolic expressions. Two translation libraries are implemented. The first one contains functions to translate C code into a formal model. The second one contains functions to translate the symbolic simulation results into properties for the equivalence checker. Fig. 1 shows the validation scheme.

We use the ACL2 theorem prover [12] to implement the symbolic simulation environment. ACL2 is a theorem proving system and a programming environment. The logic of ACL2 is based on quantifier-free *First Order Logic*. In

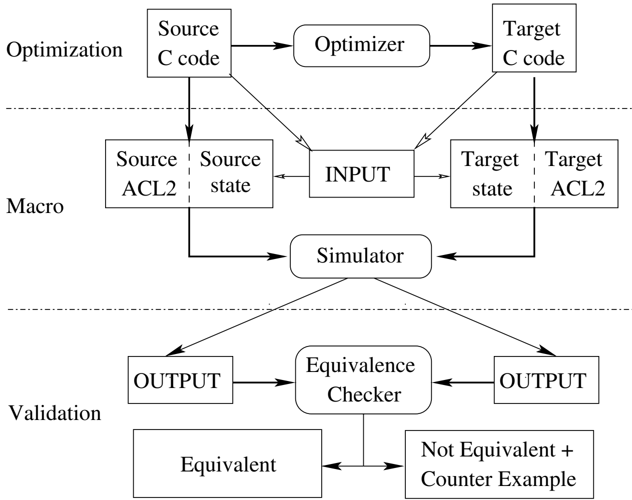

Fig. 1. The Optimization Validation Scheme

a programming environment, functions and formal models can be executed or simulated. This makes ACL2 a unique environment to perform both symbolic simulation and theorem proving. In this paper, we present our work on validation by symbolic simulation.

We define LISP macros to automate the formalization of the model from C to ACL2 and the validation process [1]. The validation method is as follows. A source design is optimized and transformed by the optimizer algorithms. The result is an optimized design described in the same language as the source. LISP macros take these two designs and generate the equivalent formal models. In addition, a state environment is generated for each of the designs. The state environments are initialized with a common set of input variables. The models were symbolically simulated with their respective state environment.

The simulation results from both models are then paired and translated into properties and checked using an equivalence checker. We use Yices [5] as the equivalence checker to validate the functional equivalence between the source and target designs. When the equivalence checker fails to validate the properties, counter examples are generated for debugging.

4 Formal Models and Validation

4.1 Formal Models

C Intermediate Language

We use an intermediate language which is a C abstract syntax tree (AST) to describe the C model. The goal of using these form of C code is to simplify the analysis of the code. Most of the conversions are converting the argument into

```
TYPE             ::=   _int
IO               ::=   _in | _out | _inout
CONSTANT         ::=   number
VAR              ::=   name [:: array-size]
VAR-DECL         ::=   TYPE :: VAR [:: IO]
ARITH-OP         ::=   _+ | _- | _* | _/ | _< | _> | _<> | _% | _not | _or | _and
SUB-STMT'        ::=   VAR
                   |   CONSTANT
                   |   ARITH-OP :: SUB-STMT' :: SUB-STMT'
FOR-CONDITION    ::=   name :: SUB-STMT' :: SUB-STMT' :: number
SUB-STMT         ::=   SUB-STMT'
                   |   _if :: SUB-STMT :: SUB-STMT [:: SUB-STMT]
                   |   _== :: SUB-STMT :: SUB-STMT
STATEMENT        ::=   _= :: VAR :: SUB-STMT
                   |   _for :: FOR-CONDITION :: STATEMENTs
                   |   _if :: SUB-STMT :: STATEMENTs [:: STATEMENTs]
                   |   _par :: STATEMENTs
PROGRAM          ::=   VAR-DECLs
                       STATEMENTs
```

Fig. 2. A subset of C-like language

left operators (prefix) arguments and adding the parenthesis. For example, a C code assignment $a = b$*;* in AST is defined as *(_= (a) (b))*.

We have implemented a subset of C like grammar with LISP macros. The macro expansions will generate the formal model of the design and its operating environment in ACL2 logic. The syntax of C subset used in the analysis is presented in Fig. 2.

Our application domain predominantly contains expensive arithmetic calculations targeted for Field Programmable Gate Array (FPGA) implementation. This is reflected in the choice of the subset of the language. Similar to C, all operations are based on integer data-type. Only this data-type is implemented in the system. Our work is focused on validating the optimization of core algorithms. The AST structure flattens the algorithm. As a result, we do not need to implement function definitions. We also do not allow pointers data type throughout the designs. The equivalence checker requires additional knowledge about the system: users must provide informations about inputs and outputs. The inputs are used as the common initialization value for the designs; the outputs are used by the checker as the validation targets.

The image summation (*Isum*) code shown in Fig. 3 is used to illustrate the stages of LISP macro expansions in generating its corresponding formal model. The source design for the macro functions is shown in Fig. 4.

State Encoding

A state environment contains all variables used in the design. These variables consist of those defined by the variable declaration statements and the loop-variables of the loop statements. The state environment is a linear list of

```
sum = 0;
for (jc = 0; jc < hc; jc++)
    {for (ic = 0; ic < wc; ic++)
         {sum = 0;
          for (j1 = 0; j1 < h1; j1++)
              {for (i1 = 0; i1 < w1; i1++)
                    sum = sum + imageI[(jc + j1)][ic + i1];}
          imageO[jc][ic] = sum;}}
```

Fig. 3. Image summation (Isum) algorithm in C-code

```
(_= (sum) 0)
(_FOR ((_= jc 0) (_< jc (hc)) (1))
      ((_FOR ((_= ic 0) (_< ic (wc)) (1))
             ((_= (sum) 0)
              (_FOR ((_= j1 0) (_< j1 (h1)) (1))
                    ((_FOR ((_= i1 0) (_< i1 (w1)) (1))
                           ((_= (sum)
                                (_+ (sum)
                                    (imageI (_+ (jc) (j1)) (_+ (ic) (i1)))))))))
              (_= (imageO (jc) (ic)) (sum)))))))
```

Fig. 4. Image Summation (Isum) algorithm in ACL2-code

variables. Macros expand array variables and generate a new variable based on the variable name and its index. The state environment for *Isum* is as follows:

```
(list Isum.imageI.0.0 ... Isum.imageI.(hc + h1 - 2).(wc + w1 - 2)
      Isum.imageO.0.0 ... Isum.imageO.(hc + h1 - 2).(wc + w1 - 2)
      Isum.sum Isum.hc Isum.wc Isum.jc Isum.ic Isum.j1 Isum.h1)
```

The state contains two array variables (*imageI* and *imageO*) and seven integer variables. The arrays have two integer indices which are separated by a dot (.).

The state is a local object to the design. A name from a different design code refers to a different object. Along with its state environment, accessor functions are also generated for each design. For example, a design named *Isum* will have *Isum-put* and *Isum-get* as the function to update a specific state element and to read a specific element of the state. In every validation process, two sets of these functions are generated: one set for the source design, the other for the target design.

Statement Modelling

A design is represented by a *PROGRAM*. It is interpreted as a function over the state environment. Similarly, each *STATEMENT* of a *PROGRAM* is a function over the state environment. We only allow four type of statements: variable assignment, loop statement, conditional statement, and parallel computation.

The modelling algorithm contains two main features. First, all loop statements are preprocessed. The body of a loop statement is split as a separate function. The loop itself is replaced as a function call to the new function. Second, all top level if statements are normalized using *if-conversion*. With this normalization, we transform predicated evaluation into non-branching/linear evaluation. The model for variable assignments, conditional statements, and parallel computations are very similar. Every variable operand is an accessor of the state environment. The result is either used for further computations or updated the state environment. In a parallel computation, every statement within the same parallel block uses the same state environment. In this section, variable assignments and loop statements will be elaborated.

A variable assignment updates the content of its current state value with a new value. The first assignment statement *(_= (sum) 0)* is the assignment of variable *sum* with a constant number *0*. The macro generated an ACL2 code for the statement with the accessor function *Isum-put*. The result is *(Isum-put 0 (Isum.n) st-1))*. This ACL2 statement has the meaning that it updates the value of variable *Isum.sum* of state *st-1* with a constant value of 0.

After the state encoding phase, macros analyze the program for any loop statement. Consider the most inner loop of the *Isum* code.

```
for (i1 = 0; i1 < w1; i1++)
    sum = sum + imageI[(jc + j1)][ic + i1];
```

This LISP macro assigns the loop statement as *Isum.6*. The index 6 denotes the position of the loop within the algorithm. For each loop statement two additional functions are generated to represent loop iterations and its body statements. The first function is the loop's iteration control *Isum.6-loop-it*.

```
(defun Isum.6-loop-it (st loop-list)
       (if (consp loop-list)
          (Isum.6-loop-it (Isum.6-loop-fn st (car loop-list)) (cdr loop-list))
          st))
```

The *Isum.6-loop-it* function has two arguments *st* and *loop-list*. The first argument *st* is the state that will be updated and the second argument is the loop sequence. The second function is for the statements in the loop's body *Isum.6-loop-fn*.

```
(defun Isum.6-loop-fn (state-1 loop-val)
       (let ((state-2 (Isum-put loop-val (Isum.i1) state-1)))
            (Isum-put ('+ (Isum-get '(Isum.sum) state-2)
                          (Isum-get (Isum.imageI
                                       ('+ (Isum-get (Isum.jc) state-2)
                                           (Isum-get (Isum.j1) state-2))
                                       ('+ (Isum-get (Isum.ic) state-2)
                                           (Isum-get (Isum.i1) state-2)))
                                     state-2))
                      (Isum.sum)
                      state-2)))
```

The *Isum.6-loop-fn* function consists of micro operations of the body. It involves updating the loop variable, getting the value of array argument from the state table, using them to obtain the content of *Isum.imageI*, adding it with the content of *Isum.sum*, and storing back the result to the location of *Isum.sum* in the state table.

4.2 Validation

The validation flow in our proposed approach consists of two stages. In the first stage, each of the models is evaluated using symbolic simulation technique. Then, the results are checked using an equivalence checker.

The symbolic simulation is based on a state updating technique. Every instruction in the model is represented as a function over the state. Expressions of the statement access the state to obtain or update its current value. The result of evaluating an instruction is an updated state. This process is performed iteratively until the simulation terminates.

Symbolic simulation operates on symbolic expressions rather than an exact/integer values. Symbolic expressions can be considered as a tree like structure with operands as its leaf nodes. The expressions are denoted in infix notation. Numerical operations operate on these symbolic expressions resulting in new symbolic expressions. We use lazy operation where no interpretation is performed on the symbolic expressions during numerical operations when it is not necessary. As a result, symbolic expressions of *(0 + a + b)* is maintained as it is. The expression does not have to be the same as *(a + b)*. A limited symbolic expression interpretation may be performed on the conditional part of an *if* statement. In this operation, only total simplification is allowed. It is when the expression does not contain symbolic constants. When there is simplification, only one of the if statement will be used. If not, a complete expression will be maintained. Consider a symbolic expression *(if a-expr b-expr c-expr)*. If *a-expr* can be simplified, the final expression will be either *b-expr* or *c-expr*. If not, the original expression *(if a-expr b-expr c-expr)* will be maintained.

There is a limitation in this symbolic simulation approach. Only symbolic constant assignment can be applied to data variables. All control variables for the loop require the exact integer value of the variable. With this we would be able to perform analysis by loop unrolling.

The functional equivalence checking is performed on the symbolic simulation results of the source and target designs. The functional equivalence is expected to be satisfied at all output variables regardless of their local/internal state. We extract the output variables from each state environments and paired them as in-equality. For example, the imageO.0.0 *(Isum.imageO.0.0)* of the source *(state-1)* and target *(state-2)* states is represented as *(/= state-1(Isum.imageO.0.0) state-2(Isum.imageO.0.0)*. The formulae stated that the content of *Isum.imageO.0.0* in *state-1* is not the same as the content of *Isum.imageO.0.0* in *state-2*. The decision procedure (Yices) checks whether the formulae can be satisfied. If they can be satisfied, Yices will provide the counter examples for debugging. If not, the arguments of the in-equality formula are equivalent.

5 Case Studies

Manipulating large high resolution images requires a lot of resources but in many situations, only limited resources are available. Many algorithms, such as caching, have been developed to speed up the process. While good results have been achieved by optimizing the code, the complexity of the optimized code also grows. In this case study, we demonstrate our validation approach in validating a range of cache optimizations for image manipulation. All experiments are carried out on a laptop with an Intel Core^{TM} 2 Duo (2.10 GHz) processor and 4GB memory.

We have developed an image manipulation algorithm (Isum) that calculated a local sum of surrounding pixels of an image. the code is described in Fig. 3. It contains 7 lines of main code. The algorithm is an uncache version of the algorithm. It is used as the reference/specification in validating the optimized cache version of the algorithms.

Most image manipulation algorithms contain independent calculations. It is possible to speed up the process by optimizing cache usage and introducing parallelization. We develop two cache windowing optimization scenarios: column cache [24] and zigzag. **Column cache** is a caching scheme that partitions/divides the image on the column (vertically). Loop **zigzag**ing is used in nested loops by reversing the inner loop every other run. Without zigzagging, the cache needed to be refilled at the start of each line. By reversing the direction, part of the cache can be reused, and only the remainder needs to be refilled.

Using the combination of these memory optimization techniques, we generate six versions of image manipulation algorithms with cache. Fig. 5 shows the optimized image manipulation algorithm with column cache. All cached algorithms contain more lines of code. In one case, the size of an optimized code is more than 19 times longer than the original specifications. The comparison for the algorithms is shown in Fig. 6.

```
int sum = 0;
for (jc = 0; jc < hc; jc++)
    {for (j1 = 0; j1 < h1; j1++)
        {for (i1 = 0; i1 < nCols; i1++)
            {cache[j1][i1] = image2[jc + j1][i1];}}
     for (ic = 0; ic < wc; ic++)
        {sum = 0;
         for (j1 = 0; j1 < h1; j1++)
            {for (i1 = nCols; i1 < w1; i1++)
                {sum += image2[jc + j1][ic + i1];}}
         imageC[jc][ic] = sum + cache.sumImage();
         for (j1 = 0; j1 < h1; j1++)
            {for (i1 = 0; i1 < nCols - 1; i1++)
                {cache[j1][i1] = cache[j1][i1 + 1];}
                 cache[j1][nCols - 1] = image2[jc + j1][ic + nCols];}}}
```

Fig. 5. Image Summation (Isum) algorithm with column cache in C code

Algorithm	Code Size (lines)	Validation time (seconds)
Uncached (specification)	8	-
Column cache	15	1.95
Column cache + zigzag	38	1.52
Column cache + one partial Column Cache	23	2.31
Column cache + one partial Column Cache + zigzag	56	3.01
Large Cache zigzag	100	3.38
L-shape cache + zigzag	152	4.25

Fig. 6. Comparison of algorithm code size and validation time

The table contains the size of each algorithm measured in the number of lines of code and the time to validate the optimized algorithm with the source (reference) algorithm. The image size selection is based on two factors. First, the validation system resources are not yet scalable to handle large image. It is currently limited to deal with maximum image size of 30x30. Second, we argue that it is sufficient to use a small image to perform a quick sanity check of the optimization. Furthermore, a large image only adds the validation complexity while the core features of the algorithms remain the same.

The algorithms are set to manipulate image of size 10x10. The size of cache memory in the optimized algorithms are approximately 25% of the image size. Each process, such as loading the library, generating formal models, simulating symbolically, and validating their functional equivalence, takes less than 5 seconds each. During the validation process, we find one error from incorrect usage of loop variable outside its scope. The error is not detected by the gcc compiler during the compilation and no warning is generated. The symbolic simulation validation proposed in this paper eliminated the need in using a set of images to check the correctness of the target algorithm.

6 Summary and Future Work

We have adopted a pragmatic approach in developing a formal validation framework to ensure our design optimization algorithm produces correct results. The proposed validation framework performs the validation every time a source design is optimized. This process ensures that the optimization and transformation do not change the design's functional behavior.

The framework is implemented in the ACL2 theorem prover. ACL2 LISP macros are used to automatically generate the formal model and the analysis environment for the design. Key features in the modelling are the state encoding which represents all variables of the model and model simplifications by if-conversion and loop-conversion. The correctness is assured by checking that the results of evaluating the models using ACL2 symbolic simulator are equivalent using Yices. A debugging facility is provided for the user to manually check the symbolic simulation results.

We have demonstrated through our case studies that the technology is sufficient to achieve our goals. It also provides a useful environment for designers to perform manual debugging by evaluating the symbolic simulation results of the designs.

We plan to extend the system with the capability to analyze unflattened abstract syntax tree. We also plan to integrate the validation system as part of the hArtes [6] framework. The framework is an end-to-end framework for the development of a real-time embedded system design on a heterogeneous reconfigurable platform.

Acknowledgment

The authors thank the ACL2 developers and SRI for making the ACL2 and Yices system available. This research is supported by the European Framework 6 Programme, Project grant 035143, Holistic Approach to Reconfigurable Real-Time Embedded Systems (hArtes).

References

1. Borrione, D., Georgelin, P., Rodrigues, V.: Using Macros to Mimic VHDL. In: Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers, Dordrecht (2000)
2. Canon, `http://www.canon.co.uk`
3. Cupak, M., Catthoor, F.: Verification of Loop Tranformations for Complex Data Dominated Applications. In: High Level Design Validation and Test, La Jolla, California, November 1998, pp. 72–79 (1998)
4. Dave, M.A.: Compiler verification: a bibliography. ACM SIGSOFT Software Engineering Notes 28(6) (November 2003)
5. Dutertre, B., Moura, L.: System Description: Yices 1.0, SRI International (2006)
6. hArtes, `http://www.hartes.org`
7. Hoare, C.A.R., He, J., Sampaio, A.: Normal form approach to compiler design. ACTA informatica 30(8), 701–739 (1993)
8. Krishnamurthy, N., Abadir, M.S., Martin, A.K., Abraham, J.A.: Design and Development Paradigm for Industrial Formal Verification CAD Tools. IEEE Design and Test of Computers 18(4), 26–35 (2001)
9. Kurshan, R.P.: Formal Verification in a Commercial Setting. In: Proc. Design Automation Conference, Anaheim, pp. 258–262 (1997)
10. Lerner, S., Millstein, T., Chambers, C.: Automatically Proving the Correctness of Compiler Optimisations. In: Proc. Programming Language Design and Implementation, SanDiego, California (June 2003)
11. Tristan, J.B., Leroy, X.: Formal verification of translation validators: A case study on instruction scheduling optimizations. In: Proc. 35th symposium Principles of Programming Languages, January 2008, pp. 17–27 (2008)
12. Moore, J.S.: Symbolic Simulation: An ACL2 Approach. In: Gopalakrishnan, G.C., Windley, P. (eds.) FMCAD 1998. LNCS, vol. 1522, pp. 334–350. Springer, Heidelberg (1998)

13. Necula, G.C.: Translation Validation for an Optimizing Compiler. In: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, Vancouver, British Columbia (June 2000)
14. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, p. 213. Springer, Heidelberg (2002)
15. Oliveira, M., Woodcock, J.: Automatic generation of verified concurrent hardware. In: Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 286–306. Springer, Heidelberg (2007)
16. Perna, J.I., Woodcock, J.: A Denotational Semantics for Handel-C Hardware Compilation. In: Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 266–285. Springer, Heidelberg (2007)
17. Saito, H., Ogawa, T., Sakunkonchack, T., Fujita, M., Nanya, T.: An Equivalence Checking Methodology for Hardware Oriented C-based Specifications. In: Proc. High-Level Design Validation and Test Workshop, October 2002, pp. 139–144 (2002)
18. Singh, S., Lillieroth, C.J.: Formal Verification of Reconfigurable Cores. In: Proc. Field-Programmable Custom Computing Machines, Napa Valley, California, pp. 25–32 (1999)
19. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Using Model Checking with Symbolic Execution to Verify Parallel Numerical Programs. In: Proc. International Symposium on Software Testing and Analysis, Portland (2006)
20. Susanto, K.W., Melham, T.: Formally Analysed Dynamic Synthesis of Hardware. Journal of Supercomputing 19(1), 7–22 (2001)
21. Susanto, K.W., Luk, W., Coutinho, J.G., Todman, T.: Validating Design Optimisation. In: Proc. Tools and Techniques for Verification of System Infrastructure, London, March 2008, p. 36 (2008)
22. Stalmarck, G.: A System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from a Formula, Swedish Patent No. 467 076 (1992), U.S. Patent No 5 276 897 (1994), European Patent No 0403 454 (1995)
23. Todman, T., Coutinho, J.G., Luk, W.: Customisable Hardware Compilation. Journal of SuperComputing 32 (2005)
24. Todman, T., Luk, W.: Memory Optimisations for High Resolution Imaging, in Proc. In: Proc. International Conference on Field-Programmable Technology, Brisbane, Australia (December 2004)
25. Zuck, L., Pnueli, A., Fang, Y., Goldberg, B., Hu, Y.: Translation and Run-Time Validation of Optimized Code. Electronic Notes in Theoretical Computer Science 70(4) (2002)