FELIX: USING REWRITING-LOGIC FOR GENERATING FUNCTIONALLY EQUIVALENT IMPLEMENTATIONS

C. Morra[1] *J. Becker*[1] *M. Ayala-Rincón*[2] *R. Hartenstein*[3]

[1]ITIV
Universität Karlsruhe (TH)
Karlsruhe, Germany
morra@itiv.uni-karlsruhe.de
becker@itiv.uni-karlsruhe.de

[2]Departamento de Matemática
Universidade de Brasília
Brasília, Brazil
ayala@mat.unb.br

[3]Fachbereich Informatik
TU Kaiserslautern
Kaiserslautern, Germany
hartenst@rhrk.uni-kl.de

ABSTRACT

FELIX is a new design space exploration tool and graphical Integrated Development Environment (IDE) for the programming of coarse-grained reconfigurable architectures. Its main and novel advantage is the use of rewriting rules and logical strategies for the automated generation of alternative functionally equivalent implementations from a single mathematical specification. The user selection of the rewriting logic strategies to be applied determines the resulting implementations, making it possible to quickly generate, simulate and evaluate alternative implementations that are logically equivalent. The FELIX System includes an interface to the KressArray Xplorer for hardware design-space exploration. The current version of the tool is targeted for the Pact eXtreme Processing Platform (XPP), with support for additional architectures planned in future versions.

1. INTRODUCTION

When prototyping a given mathematical function, it is necessary to check different equivalent alternatives of computing it, in order to obtain the best trade-offs for the desired application. The exploration of implementation alternatives is a complex and time consuming process which usually requires a lot of expertise from the developer. The exploration of alternative implementations on reconfigurable hardware is an even more challenging task because of the added complexity resulting of simultaneous hardware/software co-design.

There is growing interest in coarse-grained reconfigurable arrays because they offer increased power and better area efficiency than fine-grained reconfigurable arrays[1]. Reconfigurable arrays can be classified by level of reconfiguration capability in fine- and coarse-grained. A coarse-grained array is defined as an array of functional elements with pathwidths greater than 1 bit; In other words, reconfiguration is done at the function element level instead of the gate level. Working at the functional level allows an easier application of approaches taken from mathematics and theoretical computer science, like formal methods and term-rewriting.

Term-rewriting has very simple operational semantics, which are based on matching and simplification (reduction); It offers a natural mechanism for manipulation of algebraic expressions and is considered the formal framework for reasoning about the functional programming paradigm[2][3]. Because of its simplicity, describing computational (hardware-software) behavior using term rewriting avoids the inclusion of unnecessary operational semantics that are required in programming languages.

In the mathematical context, term rewriting allows for reduction of algebraic expressions until reaching the "simplest" equivalent forms. This simplicity is measured according to the intuition that when applying a reduction step to a term this is being transformed into a simpler one. By combining rewriting with logic strategies, rewriting(-logic) allows for different paths of simplification. This is done by selecting different strategies for controlling the application of the rewriting rules. This enables the generation of different alternative equivalent "simplified" versions of the original mathematical expression, which are by definition “correct-by-design” and can be further synthesized and tested using other tools.

By expanding term-rewriting with logic, which is called rewriting-logic, it is possible to obtain a natural mechanism of discriminately representing the behavioral changes introduced by reconfiguration over flexible architectures. The foundational mathematics of rewriting theory combined with the versatility of logic (plus the type manipulation included in rewriting-logic computational environments) have been proved of great usefulness in the prototyping of dynamically reconfigurable architectures developed for space efficient computation of non trivial algebraic operator such as the FFT[4] and for

implementing run-time efficient dynamical programming methods for sequence comparison[5].

2. RELATED WORK

Diverse tools and approaches have been used to generate code for coarse-grained architectures. The Pact XPP VCC Compiler[6] uses automated loop unrolling and loop vectorization to parallelize a C program and map it into the XPP architecture. An interesting approach is the retargetable dataflow compiler, which uses a description of the target architecture in a separate hardware file in order to allow the use of the same compiler for different coarse-grained architectures. Two relevant examples of such retargetable compilers include the ALEX Compiler included in the KressArray Xplorer[7] and the DRESC Compiler[8]. A different approach is taken by the COMPAAN/LAURA[9] toolset, which extracts the dataflow and control code from a mathematical model created in Matlab/Simulink and converts it into synthesizable VHDL code. The KressArray Xplorer allows hardware design-space exploration for reconfigurable coarse-grained arrays[10].

Recent work on rewriting based treatment of hardware design includes the work from Kapur, who used the well-known Rewriting Rule Laboratory - RRL for verifying arithmetic circuits[11]. Arvind applied rewriting in the specification of processors with simple architectures, the rewrite-based description and synthesis of simple logical digital circuits and the description of cache protocols over memory systems[12]. In his work, terms and rewriting rules were used to describe hardware states and behavior. Rewriting-logic has been shown to have greater flexibility than pure rewriting for the discrimination between fixed and reconfigurable elements of reconfigurable architectures, allowing for a natural and quick conception and simulation of implementations of new reconfigurable computing paradigms. Applications of rewriting-logic in this architectural design context include formal verification of processors (using model checking implemented by rewriting) [13] and [4][5] mentioned previously.

2.1. REWRITING LOGIC

In the architectural context, rewriting rules are of the form $s \Rightarrow t$ if C, where s and t are terms over a given signature, the left- and right-hand sides of the rule, and C is its premise. Usually, these rules are syntactically restricted in such a way that variables occurring in the right-hand side of the rule and in the premise occur in the left-hand side too. A given term can be reduced using such a rewriting rule, whenever the left-hand side of the rule matches a sub-term of the given term and the corresponding instance of the premise holds. Thus, the given term is reduced by substituting the matched sub-term by the corresponding instance of the right-hand side of the rule. These operational semantics are the same as those involved in functional environments and has been promoted in functional programming languages since the well-known McCarthy LISP of the 1950s[14].

In the algebraic context, pure rewriting is useful for reaching canonical or normal forms of a given term; that is, reaching the "simplest" representation corresponding to a given algebraic expression. But in the architectural context, it is well-known that the "simplest" expression does not necessarily coincide with the more adequate form to implement the associated operator or function. Thus, combining rewriting with logic strategies increases the capabilities for determining different alternatives of representing these operators; the logic strategies control the application of the rewriting rules, allowing the generation of different canonical forms for the same mathematical expression. In this way, we obtain a formal mechanism for producing different canonized versions of a mathematical operator, which can be quickly prototyped using other design tools.

The most popular rewriting-logic computational environments are Maude[15], ELAN[16] and CafeOBJ[17]

3. THE FELIX TOOL.

The *Functional Equivalent Logical Implementations eXplorer* (FELIX) is a new rapid prototyping tool and integrated development environment for the generation of functionally equivalent implementations from a single specification. It uses rewriting-logic rules and strategies to transform a given specification of a function into alternate ways of computing it, which are then compiled using a retargetable dataflow compiler and mapped into the given reconfigurable hardware architecture. The simulation of these alternative implementations, that are by definition logically equivalent, allows a faster exploration and evaluation of the different resulting performance and resource utilization trade-offs of each approach. The use of rewriting-logic also allows the developer to work using a higher level of abstraction, and later automatically convert the result into a function level closer to the hardware implementation. The result is translated into code for a retargetable dataflow compiler with support for multiple coarse-grained architectures. The current version of the tool is targeted for the Pact XPP[1], with support for additional architectures planned in future versions.

[1] Pact XPP[18] is an array of coarse-grain, data-driven, runtime reconfigurable ALU and RAM elements. The ALUs can perform arithmetic and logical functions, including addition, subtraction, multiplication, shifting, AND, OR, complex addition and multiplication, swap, merge, demux and other specialized functions. Each ALU Processing Array Element has also a Forward Register (FREG) and Backward Register (BREG) that are used for routing. Additionally the BREG can

FELIX can also be used for coarse-grained hardware design space exploration using its interface to the KressArray Xplorer.

The FELIX design flow is presented in Figure 1.

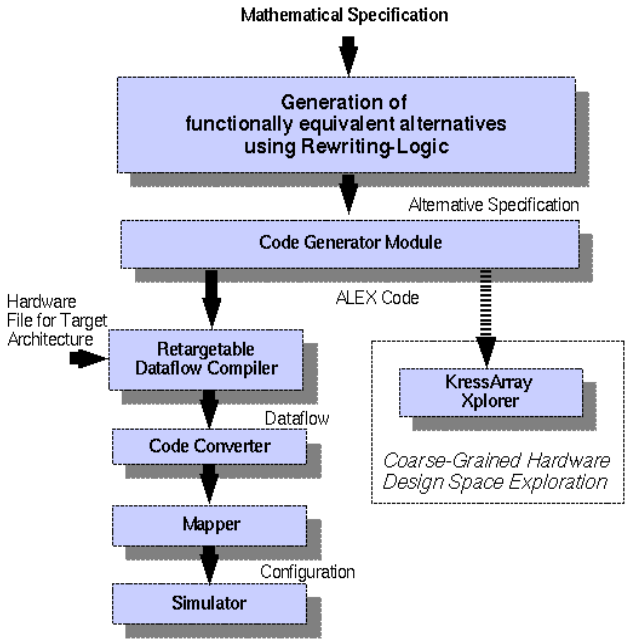

Fig. 1. FELIX Design Flow

The automated mathematical manipulation/ optimization is done using the ELAN rewriting logic system [16] with a set of custom rules and strategies. Applying the rewriting rules using different strategies produces alternative implementations that are logically equivalent to the original specification. The rewriting rules can include any combination of symbolic mathematical manipulation, element transformations and substitution for functions specified in a external module library. Different data types and their operators can be specified independently. Data-types can also be specified as a subset or superset of other data-types. Operators can be defined to be also associative (to the left or to the right) and/or commutative. Conflicts between operators are controlled by giving them priority values to indicate their hierarchy:

The code generator module automatically converts the previous stage result into compilable ALEX code, which is similar to C, for the ALEX compiler of the KressArray Xplorer. This retargetable dataflow compiler allows the generation of implementations for different target architectures by changing the target hardware description in the hardware file. The Hardware File for the XPP architecture consists of the definition of the arithmetical and logical operations of the ALU blocks (as operators) and external modules (as functions). The output of the compiler is a dataflow file consisting of a list of linked operator nodes and/or functions. The operator nodes correspond to an XPP ALU or BREG, while function nodes correspond to external modules. A NML converter translates the dataflow result into NML code for the mapping and simulation using the PSDS-M64 Software Development Suite for the XPP.

also perform addition and subtraction. The XPP is available as a synthesizable IP core with arbitrary sizes and data-widths. Pact also sells a XPP device called the XPP64A, which offers an 8x8 array of ALU elements, 16 RAM elements and 4 I/O elements, all of them with a 24 bit data-width. The XPP is programmed using Pact's Native Mapping Language (NML), and a Vectorizing C to NML Compiler is also available.

4. APPLICATION EXAMPLE #1: FIR FILTER

In this case, the goal is to automatically obtain a timing-optimized implementation from the definitional mathematical equation of the FIR Filter. The direct-form of a four-tap FIR Filter is defined by equation (1), which can also be written as (2):

$$y_i = \sum_{k=0}^{4} b_k * x_{i-k} \quad (1)$$

$$y_i = b_0 * x_i + b_1 * z(x_i) + b_2 * z(z(x_i)) + b_3 * z(z(z(x_i))) + b_4 * z(z(z(x_i)))$$

$$\text{where } z(x_i) = x_{i-1} \quad (2)$$

A graphical representation of this equation can be seen in figure 2.

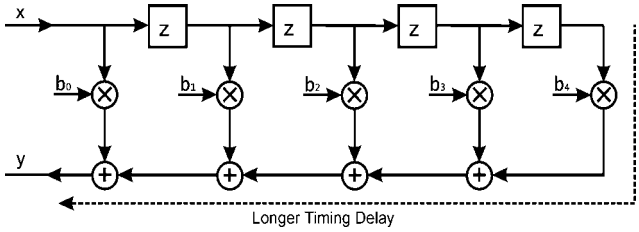

Fig. 2. Four Tap FIR Filter: direct form

To obtain a timing optimized version, the "collapse" rules from the rule library were selected. The "collapse" rules for function *z* can be seen in figure 3.

```
a : constant;
x,y : variable;
...
[collapse] a+z(x)    => z(a+x) end
[collapse] z(y)+z(x) => z(y+x) end
[collapse] a*z(x)    => z(a*x) end
```

Fig. 3. Definition of the "collapse" rewriting rules for function z

The application of these rules to equation (2) result in equation (3):

$$y = (b0*x) + z((x*b1) + z((x*b2) + z((x*b3) + z((x*b4)))))) \quad (3)$$

Equation (3) corresponds to a timing optimized version of the FIR filter, as can be seen in figure 4.

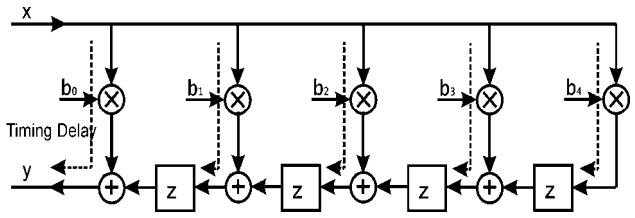

Fig. 4. Four Tap FIR Filter: timing optimized result

5. APPLICATION EXAMPLE #2: DISCRETE FOURIER TRANSFORM

This example is intended to illustrate the exploration of alternative implementations from a single mathematical specification of a well known algorithm. In this case, a Fast Fourier Transform implementation is automatically derived from the original Discrete Fourier Transform definition, by using the key idea of the FFT which is the evaluation of polynomials by their decomposition in polynomials of their pair and odd coefficients (evaluated in the squares of the original arguments). The use of a rewriting engine for the mathematical manipulation allows working on a higher abstraction level than it is available on the target architecture or even on the intermediate compiler. In this case, we will do all the mathematical manipulation on complex numbers to obtain a radix-2 Fast Fourier Transform, and afterwards use a rewriting strategy to convert the complex number operations into function calls from a external modules library, or as an alternative, convert all complex operations into real and imaginary part operations.

Let $n=2^k$, the n-DFT for a polynomial $p(x) = a_0 + a_1 x + \ldots + a_n\text{-}_1 x^{n-1}$ is defined by the vector of evaluation of this polynomial in the n complex n-roots of the unity $[p(w^i)]_{i=0..n-1}$, where w is a primitive n-root of the unity

To obtain the Fast Fourier Transform from the original DFT definition, we will use rewriting rules and strategies that decompose the input polynomial into even and odd parts which are then evaluated into the squares of the original root.

The supplied rules and strategies are parameterized regarding size (in this case for power of two), and are automatically converted by the preprocessor from ELAN to the desired size. A summary of the most important rules used in this example can be seen in Figure 5.

```
a_0,...,a_N,t1,t2    : term;
p,p1,p2,p_2,...,p_N : poly;
c1,c2               : complex;
…
[decomp] fft(p1)    =>
       [{app(p1,Root(logN,J))}_J=0...(N-1)] end
[decomp] fft([a_1])  =>  [a_1] end

[evalSqr] app(p,Root(n,n1)) =>
        app(p1,c1)+Root(n,n1)*app(p2,c1)
              where p1 :=() even(p)
              where p2 :=() odd(p)
              where c1 :=() sqr(Root(n,n1)) end

{[] odd(p_I)  =>
      [{p_I[2*J+1]}_J=0...(I/2-1)] end}_I=2...N
{[] even(p_I) =>
      [{p_I[2*J]}_J=0...(I/2-1)] end}_I=2...N

[]    sqr(Root(0,n1))  => n1 if n1 != 0 end
[]    sqr(Root(n,n1))  => Root(n2,n3)
            where n2 :=() n - 1
            where n4 :=() 2^n
            where n3 :=() ((2 * n1) % n4)/2 end

[reduceRoots] Root(n,n1)=>
          neg(Root(n,n2)) if n1>=(2^(n-1))
            where n2 :=() (n1-(2^(n-1))) end

[reduceMult] Root(n,0) => 1 end

[butterfly] t1+c1*t2 =>
            butterfly(t1,t2,c1,1)   end
[butterfly] t1-c1*t2  =>
            butterfly(t1,t2,c1,2)   end
[convertToReal] c1*c2  =>
            Re(c1)*Re(c2)-Im(c1)*Im(c2)+i*
            (Im(c1)*Re(c2)-Re(c1)*Im(c2)) end

[xppOper] c2*c1 => MUL2(c2,c1) end
[xppOper] c2+c1 => ADD2(c2,c1) end
[xppOper] c2-c1  => SUB2(c2,c1) end
```

Fig. 5. Summary of the main rules used in the DFT example

The above rules to the original DFT definition can be used to generate different FFT implementations with different trade-offs between resource utilization, speed and complexity, depending on the order of application and type of rules and used. A similar process can be used to generate other DFT implementations like the QFT, prime-factor, radix-4 FFTs, etc.

A graphical flow chart of the rewriting-logic strategies and their results, can be seen in Figure 6. Each block of the chart corresponds to a normalization step using a set of labeled rules.

The generated alternatives with the above rules include:

- A FFT using external butterfly modules, where resolution is defined by such modules.
- A slower FFT with decreased resource utilization, using a dynamically reconfigurable interconnect [4]
- A fast parallel FFT with 12 bit resolution, using XPP complex operators
- A higher resolution (24 bit), resource intensive FFT where all operation are done using normal XPP operators.

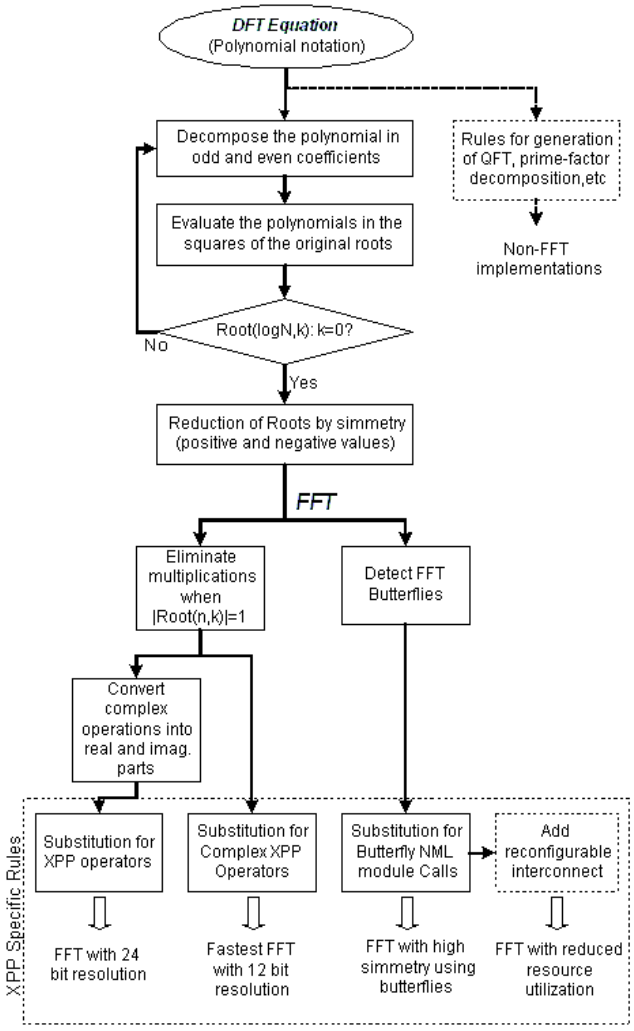

Fig. 6. Rewriting-logic strategies for the DFT and their outcomes

6. CONCLUSION AND FUTURE WORK

The *Functional Equivalent Logical Implementations eXplorer* (FELIX), a novel tool based on the combination of rewriting-logic with a retargetable dataflow compiler, is presented in this paper. With this tool, the user can generate and quickly evaluate alternative, correct-by-design implementations from a single mathematical specification. This alternative implementation generation is controlled by rewriting-logic strategies, allowing the user to compare without many effort several implementation approaches quickly, while avoiding human errors being introduced during the mathematical manipulation and simplification stages. The manipulation is done in an abstraction level close to the original mathematical specification, and is later transformed into a level close to that of the reconfigurable architecture before being converted into code for the retargetable compiler. The expressive power of the rewriting-logic framework was illustrated with the non-trivial algebraic operations of the DFT and the FFT.

Future work includes the expansion of the FELIX System to support additional architectures, which will require the writing of additional rewriting rules, new hardware description files for the dataflow compiler and new code converters at its output.

The writing of rewriting rules and strategies for managing dynamic reconfiguration and control flow structures, and the addition of more parameterized function blocks to the module library will greatly increase the capabilities and usefulness of the tool. In this way, new and more complex rules and strategies can be implemented. An obvious example, is a strategy to trade resource utilization for performance, based on a rule that will select, from the module library, a faster but bigger implementation of the sub-terms. Other more complex strategies could target power optimization and tradeoffs based on the available hardware resources.

Future versions of the presented tool may integrate an interface in development (from ELAN) to PVS[18] to allow the semi-automated verification using formal methods instead of requiring simulation. This integration includes strategies for applying rewriting proving methods (such as Knuth-Bendix-Huet critical pair lemma, detection of critical pairs, verification of joinability, etc.) in PVS.

7. REFERENCES

[1] R. Hartenstein. "A Decade of Reconfigurable Computing: a Visionary Retrospective". *Design, Automation and Test in Europe Conference (DATE'01)*, Germany, 2001.

[2] F. Baader and T. Nipkow. *"Term Rewriting and all That"*. Cambridge University Press, 1998.

[3] C. Terese and Van Rijsbergen (Editor) *"Term Rewriting Systems"*. Cambridge Univ. Press 2003.

[4] M. Ayala-Rincón, R. Nogueira, R. Jacobi, C. Llanos and R. Hartenstein. "Modeling a Reconfigurable System for Computing the FFT in Place via Rewriting-Logic". *Proceedings SBCCI'03, IEEE CS*, 2003.

[5] M. Ayala-Rincón, R. Jacobi, L. Carvalho, C. Llanos and R. Hartenstein. "Modeling and Prototyping Dynamically Reconfigurable Systems for Efficient Computation of Dynamic Programming Methods by Rewriting-Logic". *Proceedings SBCCI'04, ACM*, 2004.

[6] J. M. P. Cardoso and M. Weinhardt. „XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture". *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL'02)*, Springer-Verlag

[7] R. Hartenstein, M. Herz, Th. Hoffmann, U. Nageldinger. "KressArray Xplorer: A New CAD Environment to Optimize Reconfigurable Datapath Array Architectures". *5th Asia and South Pacific Design Automation Conference (ASP-DAC'00)*, Japan

2000.

[8] B. Mei, S. Vernalde, D. Verkest, H. De Man, R. Lauwereins. “DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures”. *Int’l Conference on Field Programmable Technology*, Hong Kong, 2002.

[9] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, E. Deprettere. “System Design Using Kahn Process Networks: The Compaan/Laura Approach”. *Design, Automation and Test in Europe Conference (DATE'04)*, France 2004

[10] U. Nageldinger. *“Coarse-grained Reconfigurable Architectures Design Space Exploration”. PhD Dissertation*, Universität Kaiserslautern, Germany, 2001.

[11] D. Kapur, and M. Subramaniam. “Using and Induction Prover for Verifying Arithmetic Circuits”. *Journal of Software Tools for Technology Transfer 3(1):32-65.*

[12] Arvind and Shen, X. 1999. “Using Term Rewriting Systems to Design and Verify Processors”. *IEEE Micro 19(3):36-46.*

[13] J. Meseguer. “Executable Computational Logics: Combining Formal Methods and Programming Language Based System Design”. In *Proc. First Int’l Conf. on Formal Methods and Models for Co-design 2003*, IEEE CS, 3-12.

[14] H. Kirchner and P. Moreau. ”Promoting rewriting to a programming language: a compiler for non-deterministic rewrite programs in associative-commutative theories”. *Journal of Functional Programming 11(2) 2001*.

[15] M. Clavel, F. Durán, S. Eker, et al. ”Maude: specification and programming in rewriting logic“. *Theoretical Computer Science 285(2):187-243*.

[16] P. Borovansky, C. Kirchner, H. Kirchner, and P. Moreau.“ELAN from a rewriting logic point of view”. *Theoretical Computer Science 285(2).*

[17] R. Diaconescu and K. Futatsugi “Logical foundations of CafeOBJ”. *Theoretical Computer Science 285(2):289-318.*

[18] V. Baumgarte, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. „PACT XPP – A Self-Reconfigurable Data Processing Architecture”. *Proc. Int’l Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'01)*, USA, 2001.

[19] S. Owre, J. M. Rushby, N. Shankar, and M. K. Srivas. “A tutorial on using PVS for hardware verification”. In R. Kumar, editor, *Theorem Provers in Circuit Design: Theory, Practice, and Experience*, LNCS 901, pages 258-279. Springer, 1995.