Fully Automatic Verification and Error Detection for Parameterized Iterative Sequential Circuits

Tiziana Margaria*

ABSTRACT The paper shows how iterative *parametric sequential* circuits, which are most relevant in practice, can be verified *fully automatically*. Key observation is that monadic second-order logic on strings provides an adequate level for hardware specification and implementation. This allows us to apply the corresponding decision procedure and counter-model generator implemented in the Mona verification tool, which, for the first time, yields 'push-button' verification, and error detection and diagnosis for the considered class of circuits. As illustrated by means of various versions of counters, this approach captures hierarchical and mixed mode verification, as well as the treatment of varying connectivity in iterative designs.

1 Motivation

A clear trend towards reuse of existing designs is the emergence of parametric designs in standard libraries [13]. While such "families of circuits" have been already popular in the hardware verification community for years, where they are the best examples for induction-based reasoning in the hardware application domain, industrial practice did not feature parametric designs in standard libraries because of lack of consolidated methods for their *full-automatic* treatment in the design lifecycle. Thus the pressure towards fully automated methods for the analysis, verification, and fault detection of parametric circuits and their specifications grows with the increasing demand for design reuse. Unfortunately, the standard automata theoretic techniques of the hardware community fail here, because the treatment of automata of unbounded size is required. At this stage, Basin and Klarlund discovered that monadic second-order logic on strings, although 'hopeless' from the complexity point of view, is well-behaved in many practical applications, and, in particular, well-suited for the fully automatic verification of

*Fakultät für Mathematik und Informatik, Universität Passau, Innstr. 33, 94032 Passau (D), tel: +49 851 509.3096, fax: +49 851 509.3092, tiziana@fmi.uni-passau.de

parametric *combinational* circuits and of the timing properties of a D-type flip-flop [3]. This general observation led to the development of the Mona verification tool [18], whose kernel consists of a non-elementary decision procedure for monadic second-order logic on strings.

In this paper we concentrate on *sequential* circuits, the class of circuits with the most practical relevance for CAD design, and in particular on the uniform treatment of whole families of the parametric iterative kind.

- *Sequential* circuits differ from combinational ones for the presence of clocked registers, or, more generally, state holding devices. Their input/output behaviour over time is usually modelled as a Finite State Machine [19], describing the values taken by the state holding elements and by the outputs over time. The standard model of time is discrete, with one observation point for each clock cycle.

- *Parametric* circuits are actually families of circuits ranging over one or more dimensions, like the width of the datapath as in the n-bit synchronous counter of Section 4.

- *Iterative* circuits exibit regularities of their structures along one dimension, e.g. pipelines or one-dimensional systolic arrays.

Parametric circuits are called iterative when the parametrization involves one dimension only. Examples include n-bit counters, which are sequential, the n-bit ALU of [3], which is combinational, but also the class of systolic arrays called "cellular automata", pipelines, etc..[1]. For this class of circuits, we

- show how they can be *fully automatically* verified,

- loosen the standard hypothesis of constant connectivity, which is inherent to most of the known approaches (see next section), showing how to deal with *variations* of the connectivity between adjacent basic cells as long as these are regular in an informal sense, and finally

- illustrate the impact of the diagnostic features offered by the Mona system on fault detection and diagnosis for hierarchical parameterized systems.

Fully automatic approaches can be integrated within existing design environments as 'black boxes' that can be used without additional expertise from the user. In particular, this also applies to Mona, which we integrated in METAFrame, a general environment for the analysis, verification and construction of complex systems.

[1] Grid structures where parameters interact (needed e.g. to handle parameterized multipliers) exceed this schema and cannot be handled in this approach.

The remainder of the paper is organized as follows. After a summary of related literature in Section 2, Section 3 sketches the background for the proposed verification method. Sections 4 and 5 illustrate respectively the verification and the fault detection and diagnosis for sequential parametric iterative designs, and finally Section 6 presents a first evaluation.

2 Related Work

Traditional approaches to parametric hardware verification with formal techniques resort to general purpose methods based on inductive reasoning within either first or higher-order logic. First order logic based theorem provers such as Boyer's and Moore's NQTHM [28, 17, 29], or the Larch Prover [1] have been used for years to handle parametric designs, as well as several higher order logic systems like HOL [14, 8, 27], VERITAS [16], or NUPRL [10, 4], and there are studies which compare both approaches [2]. In all of them, verification becomes an activity bound to interactive, computer assisted theorem proving. Thus the user must employ tactics (as in the HOL system) or suggest appropriate lemmas. While such approaches are general and quite powerful, their use is mainly restricted to verification experts as opposed to engineers and CAD designers. Also the approaches of [35, 32, 7, 20, 36] for model-checking, language containment, and process verification require interaction.

We know of two approaches to fully automate the verification of parametric circuits. They are based on symbolic methods:

Rho and Somenzi propose methods to automatically derive invariants from the structure of iterative sequential circuits by resorting to automata-based methods and to BDD-based symbolic manipulation of automata [30, 31]. The derivation process proceeds by greatest fixpoint iteration over state-minimal deterministic representations of the corresponding regular languages. It is successful whenever it stabilizes up to isomorphism. As no characteristic criterion for stabilization is given, the precise class of circuits that can be handled is unclear. The authors only consider circuits with constant connectivity.

Gupta and Fisher [15] introduce a canonical representation for inductive boolean functions (IBF) which resembles an inductive extension of BDDs covering certain classes of inductively-defined hardware circuits. Verification is carried out by symbolic tautology checking on the IBF representation of the circuits. Their results are the most similar to ours, as they cover essentially the same class of circuits, which they call *linear* parameterized sequential circuits. However, besides being complicated, the required coding in a special inductive format has the following drawbacks:

- The coding is bound to a very fine-grained analysis of the recursion

pattern and does not directly reflect the structure of the original description. Thus diagnostic information would require to re-establish the connection to the original description.[2]

- The treatment of parametric outputs is very tricky and leads to complicated representations of hierarchical structures.

In contrast, our approach allows a quite straightforward structure-preserving representation of the argument circuits and therefore good support for error diagnosis (Section 5). Moreover, as will be illustrated in the examples, no additional effort is required to treat parametric outputs. This is achieved by using Monadic Second-Order logic on Strings (M2L(Str)) for specification, which Alonzo Church proposed already 30 years ago as an appropriate formalism for reasoning about bitvectors [9]. In fact, this logic is decidable, however, only in non-elementary time: the worst-case complexity is a stack of exponentials of height proportional to the size of the formula, a good reason for it having being considered impractical so long. Fortunately, relevant problems are usually far better behaved and can be solved automatically in reasonable time[3]. The examples of this paper required CPU times ranging from fractions of a second to a few minutes.

3 Background

3.1 The Verification Scenario

The architecture of our verification setup (illustrated in Figure 1) is based on the cooperation between Mona [18] and the METAFrame environment [33]. As circuit descriptions given in a Hardware Description Language can be first translated in the target logic, along the lines introduced in [11, 23, 25][4], proofs can be carried out at the *logic* level and entirely automatically by means of Mona. The result, in form of an automaton or of a minimal counterexample, can be visualized as a graph within the METAFrame environment. The user may interact with both systems simultaneously as illustrated by Figure 2. Currently, METAFrame provides the graphic facilities for the display of the results delivered by Mona. It also allows the user to investigate properties of the graphs by means of hypertext inspectors for nodes and edges.

Mona is a verification tool for a second order monadic logic. Predicates are defined as logic formulas and transformed into minimal automata. The

[2]Perhaps this is the reason for not being considered in the paper.

[3]The so popular BDD encodings are also in general exponential, but large classes of hardware circuits have manageable polynomial representations.

[4]In these approaches the semantics of Register-Transfer and gate level descriptions was expressed in terms of first-order logic formulas.

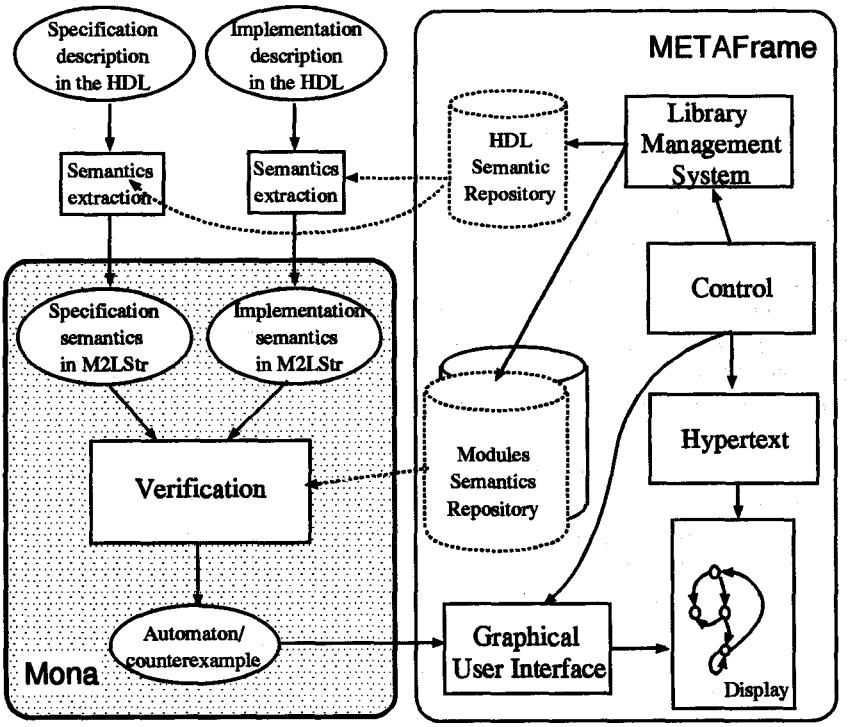

FIGURE 1. The verification scenario

computed automata are persistent objects, that is they are computed only once and stored in a library with their BDD structure. Later references to the predicate cause the precomputed automaton to be loaded and reused. Hence definitions constitute a library of reusable components which remain available for later sessions and the persistency of the precomputed automata supports compositionality with no additional cost: since the automata constitute the semantics of the hardware objects, once the equivalence between two logic objects has been proved they become semantically indistinguishable. In particular, a specification can be then replaced by its implementation (compositional verification) and structural descriptions of a module can be replaced by their behavioral counterpart (mixed-mode verification).

3.2 The Specification Language

The *monadic second-order logic on strings* (M2L(Str)) is one of the most expressive decidable logics known, and it precisley captures regular languages (see [34]). For its syntax we follow [18]. The basic operators are reported in Figure 3, where we distinguish logic terms denoting positions t, string

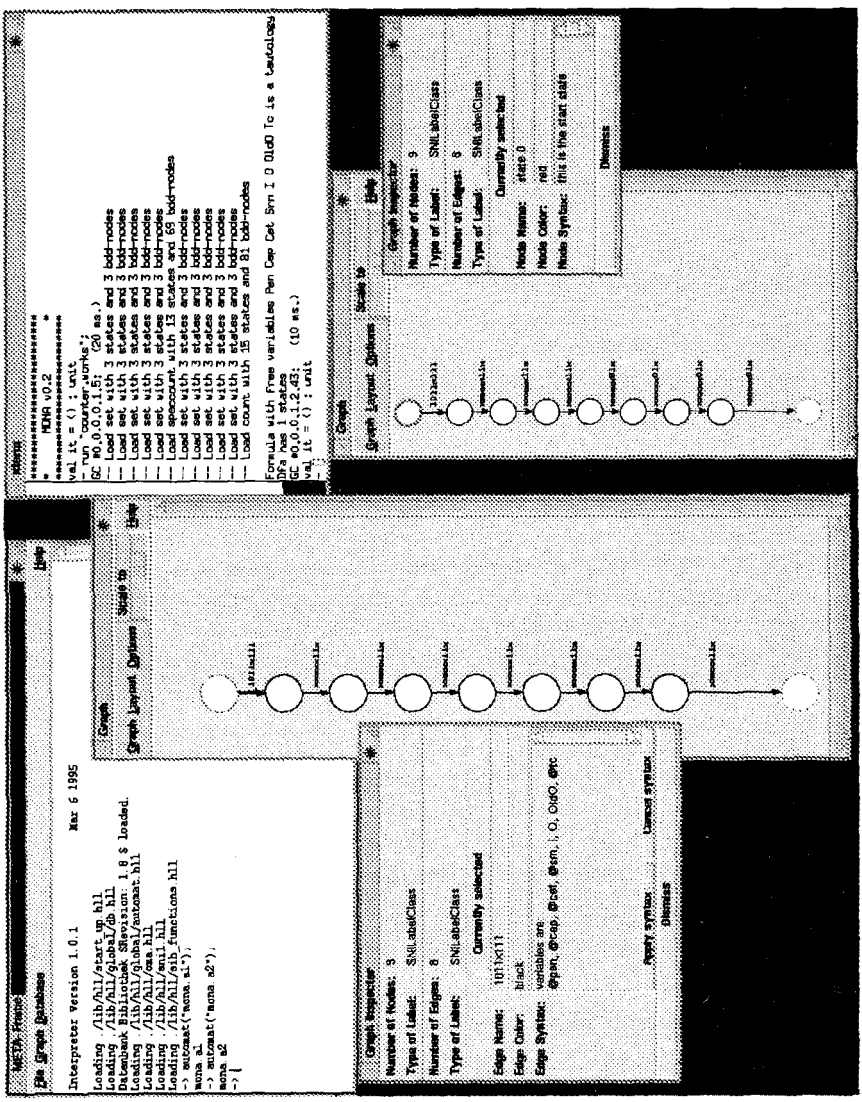

FIGURE 2. Snapshot of a user session

$$
\begin{array}{lll}
t & ::= & 0 \mid \$ \mid p \mid t + o\, i \\
T & ::= & all \mid P \mid \mathsf{C}(T) \mid T_1 \cup T_2 \\
F & ::= & tt \mid t_1 = t_2 \mid t_1 < t_2 \mid T_1 = T_2 \mid t \; in \; T \mid \neg F \mid F_1 \; \& \; F_2 \mid \\
 & & All\; p{:}\; F \mid All\; P{:}\; F
\end{array}
$$

FIGURE 3. A basic syntax for M2L(Str)

expressions T, and formulas F:

- First-order expressions t describe string positions (i.e. bitvector addresses) and are built using constants 0 (the starting position), \$ (the final position), first-order variables p, and the operator $+o$ which denotes addition modulo the string length. Here i ranges over natural numbers.

- Second-order expressions T describe strings (i.e. bit vectors) in terms of sets of positions whose value is 1 in the string. For instance, *all* represents strings whose elements are all 1 and `union` computes the union of the positions of two strings where elements are 1 and C*(T)* denotes the complement of string T. Here, P ranges over string variables.

- Formulas are interpreted over strings, which correspond to bitvectors of finite, but not necessarily precised length. They have their expected meaning. For example, the conjunction of two formulas is true when both formulas are true. The formula t *in* T is true when the position denoted by the interpretation of t is "set" (i.e., 1) in the string interpreting T. Quantification is possible over positions and over strings.

Note that many connectives given here, like e. g. position variables and their connectives, are only included for convenience since they may be encoded within the logic using second-order variables. Internally, Mona uses such an encoding. Similarly, dual connectives like, e.g. *ff*, the empty string *empty*, string disjunction $\cup$, implication $\Rightarrow$, equivalence $\Leftrightarrow$, existential quantification $\exists$, are available, as well as a short form for Boolean variables, represented as `@p`, over which quantification is possible too. Predicate definitions are equalities terminated by semicolons, and comments are introduced by the symbol `#`.

As an example, the following formulas

```
if(@cond, @then, @else) = (@cond => @then) & (~@cond => @else);

inc(Old, New) =
```

```
if(Old = all,
   New = empty,
   Ex j : j notin Old  & (All k : k < j => k in Old) &
          All l : (l < j => l notin New) &
                  (l = j => l in New) &
                  (l > j => (l in Old <=> l in New)));
```

define, respectively, a predicate for the usual $if - then - else$ construct, and a predicate `inc` corresponding to the bitvector increment operation by a simple case analysis. If `Old`, the bitvector we are incrementing, is all ones (i.e., `all` positions are set) then the result, `New`, is all zeros (i.e., `empty` positions are set). Alternatively, there exists a least position `j` which is zero (i.e. `notin Old`). In this case the increment operation should clear all the smaller positions in `New`, set this position, and leave the rest unchanged. This predicate will be used in the specification of the counter in the following section.

Interpretations of formulas are constructed by converting formulas to automata as described in [18]. For any formula ϕ that is not a tautology, a minimal length counterexample can be extracted from the corresponding automaton. This feature is exploited in Sect. 5 for fault detection, diagnosis and testing.

4 Verification of Parametric Iterative Sequential Circuits

We illustrate the flexibility of parameterization allowed in M2L(Str) by giving a behavioral specification, formalizing several architectures of counters based on the well known 74LS163 4-bit counter with enable and synchronous clear, and verifying their correctness.

The same family of counters had been already considered in [24, 26, 25] to study the impact of semantic data abstraction and compositionality in the hierarchical verification of counters of increasing but given word size. There, the semantics had been given in first order logic, and the (fully automatic) verification had been carried out by rewriting and resolution with the OTTER theorem prover. However, this approach was unable to deal with a parametric version of the implementation, a limitation which can be overcome in the fashion described in this paper.

The following four subsections discuss verification problems of increasing difficulty:

1. the parameterized behavioral specification of the counter, which is used to verify the correctness of all the proposed design architectures,
2. the implementation of the 4-bit basic cell, which illustrates how to deal with standard sequential circuits,

3. a parameterized n-bit version of it, which defines a family of iterative sequential circuits. This modelling requires the additional ability to handle variable connectivity, and finally

4. the hierarchical design of a 4×n-parameterized counter constructed as a cascade of n 4-bit modules, where iteration is needed at the hierarchical level, and ranges over the modules.

While induction-based methods are tailored for case 3, but have difficulties with case 2, automata-based methods are tailored for case 2, but fail for case 3. Case 4, being a mixture of 2 and 3, requires a specific and often intricate user interaction in all known approaches. – The handling of faulty designs is delayed to Section 5.

4.1 The Behavioral Specification

The specification defines the behavior of the circuit in terms of its input/output function independently of the word length. Its control part selects one of the four possible operations (*clear, parallel_load, increment, no_op*) according to the values of the control signals, and its data path simply consists of a data register with synchronous clear. Due to the possibility in any state (or data register configuration) of loading any other configuration, transitions are possible between any pair of states.

In [12] the specification is given by the following mode select table and expression for the terminal count TC.

SRn	PEn	CET	CEP	Action (Effect)
L	X	X	X	Reset (Clear)
H	L	X	X	Load ($I \rightarrow O$)
H	H	H	H	Count (Increment)
H	H	L	X	No Change (Hold)
H	H	X	L	No Change (Hold)

$$\mathtt{tc} = \mathtt{I}_o \wedge \mathtt{I}_1 \wedge \mathtt{I}_2 \wedge \mathtt{I}_3 \wedge \mathtt{CET} \qquad (*)$$

We may formalize this in M2L(Str) in a fairly direct manner. The functional behavior of the parameterized counter is defined by a direct encoding of this operation select table and of the additional logic equation. To model it, it is sufficient to require the consistency of its behavior over any two consecutive time units: for each operation, the value of the generated output `O` (which in this simple example coincides with the new state) and of the terminal count `tc` must be consistent with the previous output/state `OldO`, the current input `I` and the current values of the control signals as prescribed by the above table. Note that this modelling turns out to be similar to the one adopted in [18] for the dining philosophers with encyclopedia. Both model essentially an (infinite) family of state transition function. Hardware, however, usually requires an additional function in order express the

output values at any clock cycle as a function of the current state for Moore machines, and of the input and the current state for Mealy machines. For the counter, the output function is simply the identity.

The following specification is implicitely parameterized over the length of the datapath (strings I, O, OldO) and uses the predicates `if` and `inc` which belong to the basic library and have been already illustrated in the previous section.

```
speccount(@pen,@cep,@cet,@srn,I,O,OldO,@tc) =
   (if(~@srn,(All p: p notin O),
        if(~@pen,(All p: (p in I <=> p in O)),
    if(and(@cet,@cep),inc(OldO,O),
      (All p: (p in OldO <=> p in O)))))
   & (@tc <=> (All j: j in OldO & @cet)));
```

In the following we will concentrate on the relevant phenomena of each implementation style, and limit the Mona syntax to the minimum. The complete descriptions, essentially due to David Basin, are collected in Appendix 1.

4.2 Verification of the Basic Cell

The structure of the gate-level implementation of the 74LS163 4-bit counter, as found in [12], is directly reflected in the structure of the predicate `count4bit` defined in Appendix 1.1. Its automatic generation from a description in a Hardware Description Language such as, e.g., VHDL [21], ISPS [5] or CASCADE [6] would be straightforward.

Assuming that the input strings are all of length 4 (so their last position, `$`, is 3), the equivalence of the parameterized specification, restricted to 4-bit strings, with the 4-bit implementation is stated as

```
($ = 0+3 =>
    (count4bit(@pen,@cep,@cet,@srn,@tc,0 in I,0+1 in I,0+2 in I,
            0+3 in I,0 in O,0+1 in O,0+2 in O,0+3 in O,0 in OldO,
            0+1 in OldO,0+2 in OldO,0+3 in OldO) <=>
    speccount2(@pen,@cep,@cet,@srn,I,O,OldO,@tc)))
```

Mona proves this equivalence in 15 s CPU on a SparcStation 20.

4.3 The Iterative Counter: Describing Variable *Connectivity*

The structure of the n-bit generalization of the counter as shown in in Figure 4 individuates a parametric iterative family of modulo 2^n counters. Its formalization of in a top-down fashion is reported in Appendix 1.2.

Already looking at the description of the 4-bit implementation, we may have noticed that the bit slices do not have all the same inputs/outputs, as

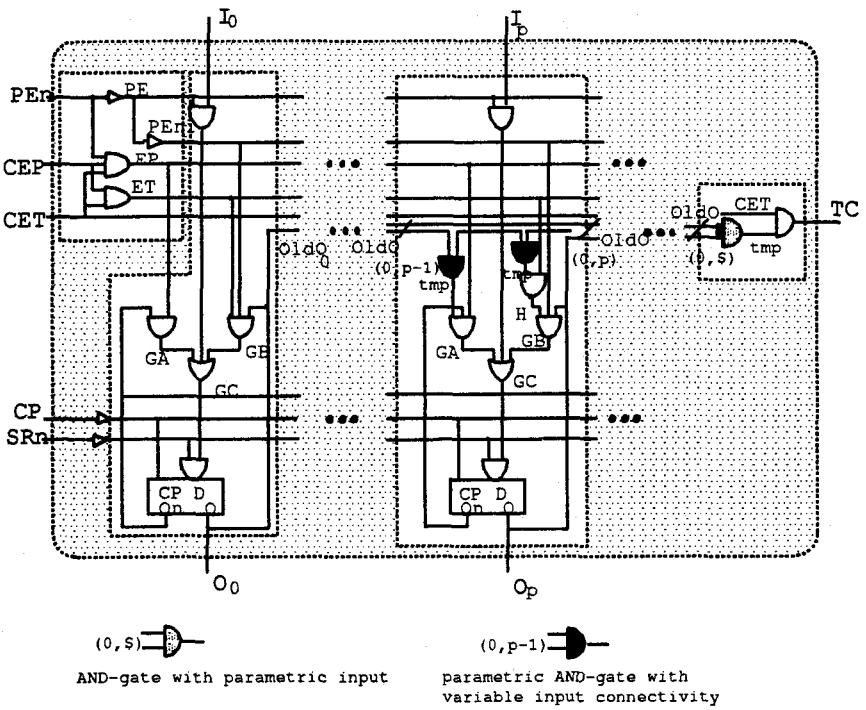

FIGURE 4. Implementation of the n-bit counter

it would be usual e.g. in systolic arrays. In the formalization of the parametric generalization, the ability to describe purely repetitive structures as e.g. in [31] would not suffice. However, since successive slices differ only in a regular manner, we are still capable to formalize this change of pattern within the logic in a natural way.[5]

According to Figure 4, each slice receives inputs from *all the previous* ones, i.e., the pth slice has one data input and one data output more than the $p-1$th. In particular, the two internal gates indicated in black have a variable number of input lines, which increases by one unit along the n dimension. This is captured by introducing the relation `n_and(p,I,@o)` which defines an *and* gate with a variable number of input lines controlled by the parameter p:

```
n_and(p,I,@o) = ((@o) <=> (All j : (j <= p => j in  I)));
```

[5]In fact, this circuit belongs to the *linear parametric* class of [15]. It may be modelled by means of their LIBF structures, providing an adequate inductive definition scheme is at hand.

This defines a *parameterized family* of AND gates, which is used in the body of the `computebit` relation and thus it occurs in the description of this family of counters.

We may now prove that this parameterized counter is equivalent to the specification.

```
(speccount(@pen,@cep,@cet,@srn,I,0,Old0,@tc)
   <=>  count(@pen,@cep,@cet,@srn,I,0,Old0,@tc))
```

Mona verifies this equivalence in 8 s CPU. In traditional approaches based on first or higher-order logic, this would require proof by induction over the (explicitly formalized) length of the string.

4.4 Hierarchical Parameterized Verification

In practice, the parametric scheme specified above is not a desirable implementation for a family of counters. Having a parameterized number of wires, the fan-in and fan-out[6] of some internal gates would become too large already for a small value of n. Real designs require fan-in and fan-out to be a small constant. A hierarchical implementation of a $4n$-bit counter in terms of a cascade of 4-bit modules (as reported in Figure 5) satisfies this need, and is easily expressed in the logic.

The formalization in Appendix 1.3 is now parameterized in the number of 4-bit units and reuses the `count4bit` predicate to implement the basic cells. Rather than using recursion, the iteration which allocates the modules in the `ripplecount` description is expressed by a predicate `fourth(p)` that is true when the position variable p takes values $3, 7, 11, \ldots$. Internal bit vector variables (`Cep`, `Cet` and `Tc`) represent the vectors of intermediate control values to be propagated between neighbouring 4-bit modules. The rippling of the terminal count `@tc` to the enabling control lines `@cep` and `@cet` of the next module follows the solution indicated in Figure 5.

The equivalence between this implementation and the behavioral specification `speccount` (or, equivalently, the parameterized implementation `count`) can be proven for all counters whose datapath is a multiple of 4-bit:

```
(fourth($) =>
    (ripplecount(@pen,@cep,@cet,@srn,I,0,Old0,@tc) <=>
       speccount(@pen,@cep,@cet,@srn,I,0,Old0,@tc)))
```

Mona verifies that this is a tautology in 100 s CPU.

Note that the design of the outer unit is truly hierarchical: we do not need

[6] Fan-in is the number of input lines to a gate, fan-out is the number of gates driven by its output.

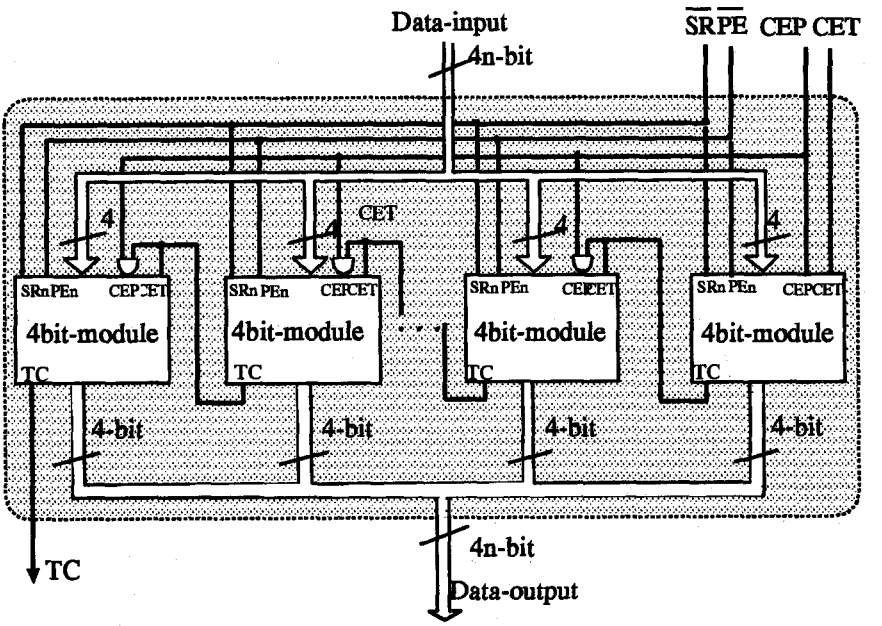

FIGURE 5. Hierarchical implementation of the $4n$ bit counter

to know any interna of the 4-bit cells. The verification at this stage ensures the correct connection of the global control and data inputs and outputs to and from each 4-bit module, independently e.g. also of parameter names. A change to an implementation in terms of 8-bit cells would only require a definition of an *eigth(p)* function and of the corresponding distribution of data inputs and outputs in bunches of 8. This could be generated as well from a HDL description.

5 Fault Detection for Parametric Hierarchical Designs

Mona supports a natural fault detection and diagnosis also for parametric hierarchical hardware designs. This is known to be a difficult problem: the hierarchical structure, intended to yield clarity and transparency, complicates error detection, as the evidence of the faulty behaviour may lie deep inside the modules used in the hierarchical descriptions. This applies in particular to induction-based verification methods.

In order to demonstrate how our approach covers this class of errors, we try to simplify the structure of the parametric hierarchical counter: rather than rippling the control signals between the modules, we connect CEP directly to the TC of the preceding module. The attempt to verify the

Theorem	Verification	
	Flat	Library Based
speccount ⇔ count	15	4
speccount ⇔ count4bit	8	1
speccount ⇔ ripplecount	100	5

TABLE 1. Summary of verification results: CPU times in s

implication wrt. the specification

```
fourth($) =>
  (ripplecount-direct(@pen,@cep,@cet,@srn,I,0,Old0,@tc)
    => speccount(@pen,@cep,@cet,@srn,I,0,Old0,@tc)))
```

yields the counterexample shown in Figure 2 (right). A quick analysis immediately reveals the problem: according to the specification, the selected operation is *hold*, which is correctly executed by the first 4-bit slice, while the second (and all the subsequent ones) performs an *increment*!

The change from one legal operation to another legal operation in a portion of a hierarchical parametric circuit is known to be a hard to detect class of faulty behaviours, but it is here easily covered. The counterexample reports values which distinguish the behaviour of the implementation and of the specification. The problem is here due to the `Tc`, which is set by the first module and it is erroneously propagated to both `Cep` and `Cet` of the second; hence the second module receives the op-code of the increment operation.

Trying to establish *equivalence* rather than implication leads to another counterexample (Figure 2 (left)), which violates the converse implication.

Interestingly, since the verification of a formula works by construction of its full semantics in terms of the corresponding minimal automaton, the cost of the error detection is independent of the fault class. In particular, faults in the data path are not more expensive to detect than faults in the control part. A more detailed discussion of the automatic generation of counterexamples for test and diagnosis purposes can be found in [22].

6 Evaluation and Future Work

The verification results reported in Table 1 clearly show the advantages of a library-based verification approach. The semantics of logic predicates can in fact be computed separately and stored as BDDs in a library (see section 3.1), so that predicates must only be evaluated once, and can be referenced in the course of a verification at virtually no cost. Already in the case of the considered counters the availability of the precompiled predicates accounts for performance improvement factors (flat vs. library based

verification) up to 20. This feature suggests the scalability of the proposed approach to quite more complex verification problems.

The logic and the verification method presented here at the gate and register-transfer level allow us to capture in a common framework a wider spectrum of abstraction levels, ranging from switch to gate, register transfer, and architecture or protocol. In fact, by proceeding hierarchically in a library-based way, verification is only necessary relative to the few basic components of each abstraction level, thus avoiding completely the need of handling large circuits at low abstraction levels.

By modelling hardware primitives as relations, rather than functions, we capture both ends of the spectrum, where bidirectionality of the signals plays a central role. Our formalism can not only express the functionality of transistors, buses, and the like, but also diverse kinds of operators, functions, and predicates, useful in specifying their behavior. For example, one may define temporal operators and reason about time dependent specifications [3] or synchronization properties of distributed systems [18] whose modelling is similar to the one adopted for sequential parametric hardware. The possibility of hiding completely to the end user the interna of Mona and of its input language once they are integrated into a CAD design environment is a central feature for the practicability of this verification method in an industrial environment.

Acknowledgements

The author is indebted to Michael Mendler and Claudia Gsottberger for their precious contribution in discussions and in the final realization of the case studies, and to Falk Schreiber, who helped combining Mona and METAFrame. Thanks are also due to David Basin and Nils Klarlund for earlier discussions, help in getting acquainted with Mona, and for their initial implementation of the counter and other examples in Mona.

7 REFERENCES

[1] M. Allemand: "*Formal verification of characteristic properties*: Proc. TPCD'94 (Theorem Provers in Circuit Design - Theory, Practice, and Experience), Bad Herrenhalb (D), Sept.'94, LNCS N. 901, pp. 292-297.

[2] C. Angelo, L. Claesen, H. De Man: "*A Methodology for Proving Correctness of parameterized hardware Modules in HOL*," Proc. CHDL'91, Marseille (F), April 1991, IFIP Transactions, North-Holland, pp.63-82.

[3] D. Basin, N. Klarlund: "*Hardware verification using monadic second-order logic*," Proc. CAV '95, Liège (B), July 1995, LNCS N. 939, Springer Verlag, pp. 31-41.

[4] D. Basin, P. DelVecchio: "*Verification of combinational logic in Nuprl*," In "Hardware Specification, Verification and Synthesis: Mathematical Aspects", Ithaca, New York, 1989. Springer-Verlag.

[5] M. Barbacci, G. Barnes, R. Cattell, D. Siewiorek: "*The ISPS Computer Description Language*", Tech. Rep. CMU-CS-79-137, Carnegie-Mellon University, Computer Science Department, Aug. 1979.

[6] D. Borrione, C. Le Faou: "*Overview of the CASCADE multi-level hardware description language and its mixed-mode simulation mechanisms*," Proc. 7th Int. Conf. on Computer Hardware Description Languages (CHDL'85), Tokyo (Japan), Aug. 1985.

[7] M. Browne, E. Clarke, O. Grumberg: "*Reasoning about networks with many identical finite state processes*," Information and Computation, 81(1), Apr. 1989, pp. 13-31.

[8] A. Camilleri, M. Gordon, T. Melham: "*Hardware verification using higher-order logic*," In D. Borrione (ed.),"From HDL Descriptions to Guaranteed Correct Circuit Designs", pages 43–67. Elsevier Science Publishers B. V. (North-Holland), 1987.

[9] A. Church: "*Logic, arithmetic and automata*," Proc. Int. Congr. Math., Almqvist and Wiksells, Uppsala 1963, pp. 23-35.

[10] R. L. Constable et al.: "Implementing Mathematics with the Nuprl Proof Development System," Prentice-Hall, Englewood Cliffs, NJ, 1986.

[11] H. Eveking: "*Axiomatizing Hardware Description Languages*," Int. Journal of VLSI Design, 2(3), pp. 263-280, 1990.

[12] "Databook of Analog and Synchronous Components", Fairchild - 1993.

[13] A. de Geus: "High Level design: A design vision for the '90s," Proc. IEEE Int. Conf. on Computer Design, p.8, 1992.

[14] M. Gordon: "*Why higher-order logic is a good formalism for specifying and verifying hardware*," In G. J. Milne and P. A. Subrahmanyam, editors, "Formal Aspects of VLSI Design", North-Holland, 1986.

[15] A. Gupta, A. Fisher: "Parametric Circuit Representation Using Inductive Boolean Functions," Proc. CAV'93, Elounda (GR), June 1993, LNCS N. 697, pp.15-28.

[16] F.K. Hanna, N. Daeche: "*Specification and verification using higher-order logic: a case study*," In G.J. Milne and P.A. Subrahmanyam, editors, "Formal Aspects of VLSI Design", pp. 179–213. Elsevier, 1986.

[17] W. Hunt: "*Microprocessor design verification*," Journal of Automated Reasoning, 5(4):429–460, 1989.

[18] J. Henriksen, J. Jensen, M. Jørgensen N. Klarlund, R. Paige, T. Rauhe, A. Sandholm: "*Mona: Monadic second-order logic in practice*," Proc. of TACAS'95, Aarhus (DK), May 1995, LNCS 1019, Springer Verlag, pp. 89-110.

[19] Z. Kohavi: "Switching and finite automata theory", Computer Science Series, McGraw Hill, New York, NY (USA), 1970.

[20] R. Kurshan, K. McMillan: "*A structural induction theorem for processes*," Proc. 8th ACM PODC Symposium, Edmonton (CAN), Aug. 1989, pp. 239-247.

[21] IEEE: "Standard VHDL Language Reference Manual", 1988, IEEE Std. 1076-1987.

[22] T. Margaria, M. Mendler: "Automatic Treatment of Sequential Circuits in Second-Order Monadic Logic", 4th GI/ITG/GME Worksh. on Methoden des Entwurfs und der Verifikation digitaler Systeme, Kreischa (D), March 1996, Shaker Verlag.

[23] T. Margaria: "First-Order theories for the verification of complex FSMs," Aachener Informatik-Berichte Nr.91-30, RWTH-Aachen, Dec. 1991.

[24] T. Margaria: "Efficient RT-Level Verification by Theorem Proving", IFIP World Congr.'92, Madrid (E), Sept. 1992, North-Holland pp. 696-702.

[25] T. Margaria: "*Verifica formale della correttezza del progetto di sistemi digitali*," Dissertazione di Dottorato (in Italian), Politecnico di Torino, Turin (I), Feb. 1993.

[26] T. Margaria, B. Steffen: "Distinguishing Formulas for Free", EDAC–EUROASIC'93: IEEE European Design Automation Conference, Paris (France), February 1993.

[27] T. Melham: "*Using recursive types of reasoning about hardware in higher order logic*," In Int. Working Conf. on The Fusion of Hardware Design and Verification, pp. 26–49, July 1988.

[28] L. Pierre: "The Formal Proof of the "Min-max" sequential benchmark described in CASCADE using the Boyer-Moore theorem prover," Proc. IMEC-IFIP Worksh. on Applied Formal Methods for Correct VLSI Design, Leuven (B), Nov. 1989, pp. 129-149.

[29] L. Pierre: "*An Automatic Generalization Method for the Inductive Proof of Replicated and Parallel Architectures*," Proc. TPCD'94, Bad Herrenhalb (D), Sept.'94, LNCS N. 901, pp. 72-91.

[30] J.K Rho, F. Somenzi: "Inductive Verification for Iterative circuits", Proc. DAC'92, Anaheim (CA), June 1992, pp. 628-633.

[31] J.K Rho, F. Somenzi: "Automatic Generation of Network Invariants for the verification of Iterative sequential Systems", Proc. CAV'93, Elounda (GR), June 1993, LNCS N. 697, pp.123-137.

[32] A. Sistla, S. German: "*Reasoning with many processes,*" Proc. LICS'97, Ithaca, NY, June 1987, pp. 138-152.

[33] B. Steffen, T. Margaria, A. Claßen. "The META-Frame: An Environment for Flexible Tool Management," Proc. TAPSOFT'95, Aarhus (Denmark), May 1995, LNCS N. 915, Springer Verlag.

[34] W. Thomas: "*Automata on infinite objects,*" In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, vol. B, p. 133–191. MIT Press/Elsevier, 1990.

[35] P. Wolper: "*Expressing interesting properties of programs in propositional temporal logic,*" Proc. POPL'86, St. Petersburg, Jan. 1986, pp. 184-192.

[36] P. Wolper, V. Lovinfosse: "*Verifying properties of large sets of processes with network invariants,*" Proc. Automatic Verification Methods for Finite Systems, LNCS 407, Springer Verlag, 1989, pp 68-80.

1 Descriptions of the Synchronous Counters

In the following examples gates and modules are encoded as relations over booleans as in this fragment of the basic library:

```
notrel(@a,@b) = (~@a) <=> @b;
andrel(@a,@b,@c) = (@a & @b) <=> @c;
orrel(@a,@b,@c) = (@a | @b) <=> @c;
```

1.1 The 4-bit Basic Cell of the Counter

The structural implementation of the 4-bit basic cell of the synchronous counter is formalized in mona as the following relation over its external ports `@pen,@cep,@cet,@srn,@tc,@i0,@i1,@i2,@i3, @o0,@o1,@o2,@o3` and the previous state values `@oo0` ... `@oo3`.

```
count4bit(@pen,@cep,@cet,@srn,@tc,@i0,@i1,@i2,@i3,
          @o0,@o1,@o2,@o3,@oo0,@oo1,@oo2,@oo3) =

 (Ex @pe: Ex @ep: Ex @pen1: Ex @et:                  # decode operation
          (notrel(@pen,@pe) & and3rel(@pen,@cep,@cet,@ep) &
```

```
          andrel(@cep,@cet,@et) & notrel(@pe,@pen1)) &
 (Ex @ga0: Ex @gb0: Ex @gc0: Ex @gi0: Ex @h0:      # set 1st output
          andrel(@pe,@i0,@gi0) & notrel(@et,@h0) &
          andrel(~@oo0,@ep,@ga0) & and3rel(@h0,@pen1,@oo0,@gb0) &
          or3rel(@ga0,@gi0,@gb0,@gc0) & andrel(@srn,@gc0,@o0)) &
 (Ex @ga1: Ex @gb1: Ex @gc1: Ex @gi1: Ex @h1:      # set 2nd output
          andrel(@pe,@i1,@gi1) & nandrel(@oo0,@et,@h1) &
          and3rel(~@oo1,@oo0,@ep,@ga1) & and3rel(@h1,@pen1,@oo1,@gb1) &
          or3rel(@ga1,@gi1,@gb1,@gc1) & andrel(@srn,@gc1,@o1)) &
 (Ex @ga2: Ex @gb2: Ex @gc2: Ex @gi2: Ex @h2:      # set 3rd output
          andrel(@pe,@i2,@gi2) & nand3rel(@oo0,@oo1,@et,@h2) &
          and4rel(~@oo2,@oo0,@oo1,@ep,@ga2) &
          and3rel(@h2,@pen1,@oo2,@gb2) &
          or3rel(@ga2,@gi2,@gb2,@gc2) & andrel(@srn,@gc2,@o2)) &
 (Ex @ga3: Ex @gb3: Ex @gc3: Ex @gi3: Ex @h3:      # set 4th output
          andrel(@pe,@i3,@gi3) & nand4rel(@oo0,@oo1,@oo2,@et,@h3) &
          and5rel(~@oo3,@oo0,@oo1,@oo2,@ep,@ga3) &
          and3rel(@h3,@pen1,@oo3,@gb3) &
          or3rel(@ga3,@gi3,@gb3,@gc3) & andrel(@srn,@gc3,@o3)) &
 (Ex @w0: Ex @w1: Ex @w2: Ex @w3:                  # set tc
          notrel(~@oo0,@w0) & notrel(~@oo1, @w1) & notrel(~@oo2,@w2) &
          notrel(~@oo3,@w3) & and5rel(@w0,@w1,@w2,@w3,@cet,@tc)));
```

This formula directly encodes the structure of the circuit and could be automatically generated from a description of this module in a gate-level hardware description language.

1.2 The Family of Parametric Counters

The parametric structure of the family is formulated in a top-down fashion and organized according to functional units:

```
count(@pen,@cep,@cet,@srn,I,O,OldO,@tc) =
  (Ex @pe: Ex @ep: Ex @pen1: Ex @et:
     notrel(@pen,@pe) & and3rel(@pen,@cep,@cet,@ep)
     & andrel(@cep,@cet,@et) & notrel(@pe,@pen1)
     & computebit(@pe,@ep,@et,@pen1,@cet,@srn,I,O,OldO)
     & Ex @tmp: n_invand($,compl(OldO),@tmp) & andrel(@tmp,@cet,@tc));
```

The 4 parts of the above definition 1) declare the ports as booleans, which correspond to values on internal wires; 2) compute internal signals which decode the selected operation (this is identical to the 4-bit version); 3) compute the new value of the output/state by means of the `computebit` relation, and 4) compute the terminal count `@tc` using the auxiliary relation

```
n_invand(I,@o) = @o <=> All j : (j notin I);
```

which defines an *n-bit inverted and* gate.

Dealing with the first slice as a special case (having no predecessors implies a different internal structure, which could be reduced to the generic one with the introduction of 1-input gates), the computation in a generic slice is defined as follows:

```
computebit(@pe,@ep,@et,@pen1,@cet,@srn,I,O,OldO) =   #compute one bit
   (All p: Ex @h: Ex @ga: Ex @gb: Ex @gc:
       if(p = 0, (notrel(@et,@h)),
                 (Ex @tmp: n_and(p -o 1,OldO,@tmp) &
                           nandrel(@tmp,@et,@h)))
     & if(p = 0, (andrel(@ep,notsetp(0,OldO),@ga)),          # set GA
                 (Ex @tmp: n_and(p -o 1,OldO,@tmp) &
                  and3rel(@tmp,notsetp(p,OldO),@ep,@ga))) &
     & and3rel(@h,@pen1,setp(p,OldO),@gb)                    # set GB
     & (Ex @gi: andrel(setp(p,I),@pe,@gi)                    # set GC
               & or3rel(@ga,@gi,@gb,@gc))
     & andrel(@gc,@srn,setp(p,O)));                          # set O
```

1.3 A Hierarchical Family of 4−bit Counters

The partitioning of the datapath in 4-bit portions is described by the predicate `fourth(p)` that is true when the position variable p takes values $3, 7, 11, \ldots$.

```
fourth(p) = (Ex S : (0 notin S & 0+1 notin S & 0+2 notin S &
              (All p : (p >= 0+3 =>
                       (p in S <=> (p -o 1 notin S & p -o 2 notin S &
                                    p -o 3 notin S)))))
  & p in S);

ripplecount(@pen,@cep,@cet,@srn,I,O,OldO,@tc) =
   (Ex Cep: Ex Cet: Ex Tc:
      (All p: fourth(p) =>
                count4bit(@pen,p in Cep,p in Cet,@srn,p in Tc,
                          p -o 3 in I, p -o 2 in I, p -o 1 in I, p in I,
                          p -o 3 in O, p -o 2 in O, p -o 1 in O, p in O,
                          p -o 3 in OldO, p -o 2 in OldO,
                          p -o 1 in OldO, p in OldO) &
                (~(p=$) => (andrel(p in Tc,@cep,p+4 in Cep) &
                           (p+4 in Cet <=> p in Tc))) &
                (0+3 in Cep <=> @cep) &
                (0+3 in Cet <=> @cet) &
                (@tc <=> ($ in Tc))));
```

Every time `fourth(p)` holds, a `count4bit` cell is instantiated. It is connected to the preceding 4 input and output ports and it computes the counter relation over these values. The rippling of the terminal count `@tc` to the enabling control lines `@cep` and `@cet` of the next module follows the solution indicated in Figure 5. Finally, the internal control signals CEP and CET are connected to the global ones and the global terminal count is defined to be the last position of TC.