This article has been accepted for publication in a future issue of a journal/proceedings, but has not been fully edited. Content may change prior to final publication. Citation information: F. Belli, Regular Expressions for Fault Handling in Sequential Circuits, Proc. ARCS 2015 - 28th International Conference on Architecture of Computing Systems, 11th Workshop on Dependability and Fault Tolerance (VERFE'15)

Regular Expressions for Fault Handling in Sequential Circuits

Fevzi Belli
Department of Computer Engineering, Izmir Institute of Technology, Urla, Izmir, 35430, Turkey
and
Faculty of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, Paderborn, Germany,
belli@adt.upb.de

***Abstract*— Based on regular expressions (RE), this paper proposes an approach to modeling sequential circuits and related faults, and introduces a strategy for self-detecting, self-localizing, and self-correcting modeled faults. A case study evaluates the approach and analyzes its characteristics.**

***Index Terms*— regular expressions, fault tolerance, redundancy, logic circuits**

I. INTRODUCTION

VERY early on, regular expressions (RE) became popular for modeling and testing logic circuits [3,4,5,8]. This paper proposes notions and techniques to model both sequential circuits and related faults based on automata theory and formal languages.

Faults can be handled in three consecutive stages: Detection, localization, and correction. If a fault in a circuit under consideration (CUC) can be detected, localized, and corrected unambiguously, then the system can perform the correction itself without needing control from outside. It is then said to be *self-correcting* in response to a fault of this type. The system is *fault-tolerant* (FT) if it can, in spite of a failure caused by a fault of the considered type, continue delivering the specified services, perhaps at a reduced level but still to the satisfaction of the user.

If a system is not immanently FT, it can be extended to behave FT, which necessitates additional resources *(redundancy)* that are not required to perform the specified services but needed to realize the FT behavior. Note that the acronym FT can read either as "fault-tolerant" or "fault tolerance" or "fault tolerating," etc., depending on its context.

This paper proposes models and methods to handle faults in sequential circuits with emphasis on (i) fault modeling to operationalize FT, (ii) analysis of a given CUC to determine whether it has the required properties to handle modeled faults, and (iii) if not, extension of the given CUC to make it possess the required, desirable feature.

This work has been supported by *Institute for Dependability and Reuse*, Paderborn.

The next section summarizes notions and techniques used in this paper. Section 3 introduces the approach that will be exemplified, and validated in Section 4. Conclusions and open problems are discussed in Section 5.

II. REGULAR EXPRESSIONS, TRANSFORMING THEM TO SEQUENTIAL CIRCUITS AND V.V.

Regular expressions (RE) play a key role in this paper. The basic assumption for the use of REs to model CUC is that the system is controlled by a sequence of input signals, and the set of legal sequences build a regular language that can be represented by REs. Apart from their relevance for automata theory and formal languages, REs provide an efficient tool for modeling and analysis of a wide range of problems in practice. Specifically, all non-recursive, sequential sequences of events can be described by REs [6]. The notation of REs is brief and precise and can easily be transformed into directed, state-transition graphs of finite-state automata (FSA) accepting the symbol strings generated by the corresponding expressions. Efficient methods have been developed for the analysis and verification of system properties (completeness, consistency, complexity) and the synthesis of composite REs to model complex systems.

A RE consists of symbols connected by the following basic operations (detailed description and use of REs is available in text books, e.g., [6]).

- *Sequence (concatenation)* of symbols–not notated by an explicit notation.
- *Selection* between symbols–notated by "+."
- *Iteration*–notated by "(…)*", meaning an arbitrary i-fold ($0 \leq i < \infty$) concatenation of the contents of the brackets.

So, *ab, a+b, and (b)** are simple REs.

Well-known techniques are suggested [3,4,6,7,8,9] for transforming a CUC into a RE, and transforming the extended RE back to a corresponding circuit. A state transition table, which is a tabular representation of CUC`s output equations, is used for obtaining an FSA from a given CUC. This table contains input-output values for current-next states. The

FSA is then created by using the state transition table and represents a Mealy machine, whose outputs are determined using current state and inputs. Other conversion from the FSA to RE is made by using JFLAP software [9].

III. APPROACH

This paper applies the notions, algorithms, and strategies, developed in [2] for regular languages, to the analysis and extension of sequential digital circuits. The author chose an intuitive, semi-formal style to enable an easy-to-understand and space-saving explanation of the concept.

A. Basic Idea, Modeled Faults

As a first step, the given CUC will be converted to an FSA that will then be converted to an RE that is an algebraic term and forms a string of symbols, which we interpret as events that represent combinations of input signals. This term will be used to model and generate a set of elementary faults that can be corrected by inserting (I-hypothesis), deleting (D-hypothesis), or replacing (R-hypothesis) events.

	Insertion	Deletion	Replacement
Stuck-at Faults			X
Bridging Faults	X		X
Delay Faults			X
Transient Faults	X	X	X

Fig. 1. Applying IDR-Hypotheses to faults in circuits

Fig. 1 summarizes typical hardware faults that can be modeled by the corresponding correction hypothesis. Most of the faults encountered in the practice can be modeled by these IDR-hypotheses, or a repetition/combination of them. They define a collection of fault prototypes that will be used to analyze the RE of a given CUC to determine whether it is fault-detecting or fault-correcting, i.e., if all faults of this type can be, respectively, detected, localized, or corrected unambiguously.

The present paper suggests to transform a CUC into a corresponding RE and to analyze this RE to find out whether it lacks a desirable fault handling property. If necessary, the redundancy for adopting this property will be determined and the given RE will be extended to a RE*. As a final step, RE* will be converted back to a circuit CUC* that represents an extension of the original CUC, and is expected to possess the desirable property; i.e., it is now fault-detecting, or correcting, etc.

Circuit hardening is used to improve the robustness against external influences to circuits like EM-fields and EM-radiation by including circuit redundancy [1,12]. Transient faults include these effects, which can be modeled by our approach.

B. Deriving Characteristic Features of Regular Expressions

Using a well-known algorithm [4,5], this section demonstrates on an example the transformation of an RE to a specific form that enables to derive the characteristic relations that are necessary for analyzing its FT features. An RE can be given thusly:

$$T=[(ba(b+c)^*)^*(a+b)^*] \quad (1)$$

As a first step, each instance of a symbol in the string T will be denoted by its order of occurrence. This leads to an indexed expression:

$$T=[^1(b^1a^1(b^2+c^1)^*)^*(a^2+b^3)^*]^1$$

Here, b^1 means "first occurrence of b, etc. In this way, symbols occurring more than once are uniquely denoted. Next, based on the well-known methods [4,6], the corresponding deterministic finite-state automaton E^{forw} will be constructed that accepts all strings forwardly generated by the expression T. The states of E^{forw} are expressed by numbers. A transition from state *i* to state *j* that is caused by the input of an *a* is denoted by *(a)i=:j*.

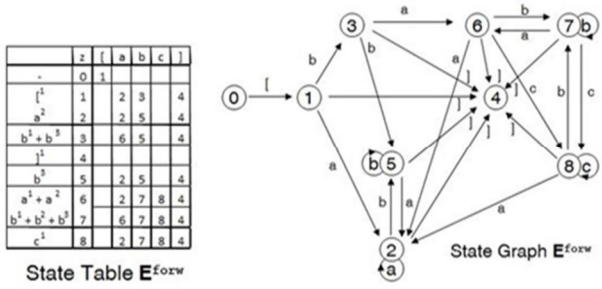


Fig. 2: Constructing E^{forw}

The operation "=:" means that "the state *j* is defined as a follow-on state *i* transduced by the input *a*." The acceptor states may be seen to be equivalent to the symbol that leads to them.

initial state = 0 — ([)0 = ([¹)0 =: 1 ≡ [¹
(a)1 = (a²)1 =: 2 ≡ a² — (b)1 = (b¹ + b³)1 =: 3 ≡ b¹ + b³
(])1 = (]¹)1 =: 4 ≡]¹ — (a)2 = (a²)2 = a² ≡ 2
(b)2 = (b³)2 =: 5 ≡ b³ — ([)2 = (]¹)2 = (]¹) = 4 ≡]¹
(a)3 = ... etc. — Final State = 4

As an example, if E^{forw} is in initial state *0* and reads the symbol *[*, which can correctly be only *[¹*, it then transfers to state *1*.

The symbol a correctly read by state *1* is always a^2 and causes a transition into state *2*, etc. Each symbol string not generated by *T* is not accepted by E^{forw}, i.e., the automaton is transduced into a *reject state e*. The graph and state table of E^{forw} are shown in Fig. 2. For reasons of more compact presentation, the reject state *e* is generally omitted.

An additional relationship between the symbols s^i of a *T* and the states of E^{forw} is established if the index *i* of the symbol s^i is replaced by the set of states containing s^i. Thus we transform *T* into a new expression.

$$T^{forw}=[^1(b^{3+7}a^6(b^7+c^8)^*)^*(a^{2+6}+b^{3+5+7})^*]^4$$

This operation is called *forward indexing* of *T*. Addi-tionally, we form the mirror image T^{mirr} of *T* and index it in the same way as *T*.

$$T^{mirr}=]^1(b^3+a^2)^*((c^1+b^2)^*(a^1b^1)^*[^1$$

The finite-state automaton E^{back} corresponding to T^{mirr} is constructed in the same way as E^{forw}.

Now, we again form the mirror image of T^{mirr} and analogously derive a relationship between the symbols of T^{mirr} and states of E^{back}.

$$T^{mirr\_forw} =]^1 (b^2 + a^2)^* ((c^4 + b^{3+7}) * (a^{2+6} b^{3+8})^* [^5$$

This operation is called *backward indexing*. Here, to distinguish the indices $T^{mirr\text{-}orw}$ from those in T_{forw}, they are written in a subscript position. After a second mirror operation, we obtain the expression:

$$T_{back} := T^{mirr\_forw\_mirr} = [_5 (b_{3+8} a_{2+6} (b_{3+7} + c_4)^*)^* (a_2 + b_2)^*]_1$$

Performing forward and backward indexing simultaneously, we obtain:

$$T_{back}^{forw} = [^1{}_5 (b_{3+8}^{3+7} a_{2+6}^{6} (b_{3+7}^{7} + c_4^{8})^*)^* (a_2^{2+6} + b_3^{3+5+7})^*]_1^4$$

This double indexing is called a *coding* of the expression *T*. The coding is the basis of both tools needed for our error treatment. One important tool is the *compatibility relation C* that consists of all pairs (*i*,*j*) of a forward index *i* and a backward index *j* existing in a coded symbol s_j^i of T_{back}^{forw}. This is described by the notations *iCj* of *isj*. The latter means that states *i* and *j* are compatible via the symbol *s*. Fig. 3(a) gives the relation *C* for *T*.

i [j	i a j	i b j	i c j	i] j
i [5	2 a 2	3 b 3	8 c 4	4] i
	6 a 2	3 b 8		
	6 a 6	5 b 3		
		7 b 3		
		7 b 7		
		7 b 8		

l^{forw}	s^{forw}	r^{forw}	l_{back}	s_{back}	r_{back}
---	$[^1$	$a^2+b^5+]^4$	---	$[_5$	$a_2+b_3+b_8+]_1$
$[^1+a^2+a^5+b^5+c^5$	a^2	$a^2+b^5+]^4$	$[_5+a_2+b_3+c_4$	a_2	$a_2+b_3+]_1$
b^5+b^7	a^5	$a^2+b^7+c^5+]^4$	b_8	a_6	$b_7+b_8+c_4$
$[^1$	b^3	$a^5+b^5+]^4$	$[_5+a_2+b_3+c_4$	b_3	$a_2+b_3+]_1$
$a^2+b^3+b^5$	b^5	$a^2+b^5+]^4$	$a_6+b_7+c_4$	b_7	$b_7+b_8+c_4$
$a^5+b^7+c^5$	b^7	$a^5+b^7+c^5+]^4$	$[_5+a_6+b_7+c_4$	b_8	a_6
$a^5+b^7+c^5$	c^5	$a^2+b^7+c^5+]^4$	$a_6+b_7+c_4$	c_4	$a_2+b_3+b_7+b_8+c_4+]_1$
$[^1+a^2+a^5+b^5+b^5+b^7+c^5$	$]^4$	---	$[_5+a_2+b_3+c_4$	$]_1$	---

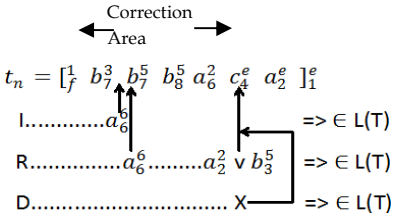
Fig. 3: (a) Context relation and (b) compatibility relation

As a second, more powerful tool, the relations for *the right context* and *left contextr*^{forw}, r_{back}, and l^{forw}, l_{back} , respectively, are used. They determine the symbols for every s^i or s^j that may appear, respectively, as its immediate successor or predecessor. Fig. 3(b) gives the context relation, derived for the symbols of T^{forw} and T^{back}.

C. Modeling, Detection, and Correction of Faults

As mentioned at the beginning of Section 3.1, the system has to apply a certain hypothesis about the structure of a detected error for its correction. Most errors in the input string can be corrected if one of the following hypotheses is used (Fig. 4(a)).

1) An additional symbol has to be *inserted* between two adjacent symbols (*I*-correction).
2) A symbol has to be *deleted* from the string (*D*-correction).
3) A symbol has to be *replaced* by another one (*R*-correction).



(2)

Fig. 4. (a) Correction area (b) Multiple R-corrections in an area

These three hypotheses can correct errors caused by a missing, false, or superfluous symbol in the string. The hypotheses may be generalized to the cases where, instead of a single symbol, a substring of *n* symbols is considered ($n \geq 1$).

1^n) An additional substring of *n* symbols has to be *inserted* between two adjacent symbols of the string (I^n-correction).

2^n) A substring of *n* symbols of the string has to be *deleted* from the string (D^n-correction).

3^n) A substring of *n* symbols of the string has to be *replaced* by another one (R^n-correction).

The location in the string where a hypothesis might be applied is called a *correction position*, and the symbol used by the correction is called a *correcting symbol*.

If one of the hypotheses applies exactly to one position in a given erroneous string, then the place of the error can be localized unambiguously. Such errors are called Q^n-diagnosing if all errors are Q^n-diagnosable. If two hypotheses, P^n and Q^n out of $\{^{In.}D^n, R^n\}$, apply to the same, erroneous string, they are P^nQ^n-dependent; otherwise they are P^nQ^n-independent. Erroneous strings that can be corrected by use of one hypothesis Q^n and by exactly one symbol are said to be Q^n-*correctable*. The system is Q^n-*correcting* if all errors are Q^n-*correctable*.

If an error is Q^n-diagnosable, Q^n–correctable, and all hypotheses P^n, Q^n are pairwise independent, it can be automatically corrected by the system itself; i.e., this error is *self-correctable*. If all errors are self-correctable, the system is said to be P^nQ^n-*self-correcting* or P^nQ^n-*fault-tolerant*.

The treatment of erroneous symbol sequences at run-time consists of steps (a) *detecting* the sequence is erroneous, (b) *localization* of a substring where a Q-correction may apply, and (c) *correction*. These three consecutive steps are explained in the following.

(a) *Detection of Erroneous Inputs*. For this purpose, the input string is successively indexed forward and backward (read

from left to right by E^{forw} and vice versa by the corresponding automaton E_{back}). Thus, if the string is correct, it can be transformed into a coding, as demonstrated by the example of T^{forw}_{back}. Otherwise, at a certain position, the coding is no longer possible without violating the compatibility relation. Consider the example of an erroneous sequence *w* (Fig. 4(b).

$$w = [bbbacabba]$$

The corresponding coding is (that will be used in Section 4 as a test input t_n)

$$t_n := w^{forw}_{back} = [^1_f b^3_7 b^5_7 b^5_8 a^2_6 c^e_4 a^e_2 b^e_3 b^e_3 a^e_2]^e_1$$

where*e*, *f* represent the reject states of E_{forw} or E_{back}, respectively.

During the forward indexing, string *w* cannot be read properly after the symbol $a^2$by E^{forw}. For the backward indexing, the same happens at symbol $b7$ by E_{back}. In these cases, the automata are transduced into a reject state *e* or *f*.

(b) *Localization.* The error causing the automaton to stop during the forward and/or backward indexing is always situated in the substring between the leftmost symbol that can be accepted at backward indexing and the rightmost symbol that can be accepted at forward indexing [2]. This is the only part of the string where, if possible, correction hypotheses may apply. Therefore, this part is called a *correction area.*

(c) *Correction.* A correcting symbol placed between the other ones in the correction area must belong to the left context relation of its successor, and to the right contest relation of its processor. This is demonstrated by the example in Fig. 4(a) of the correction area k^{forw}_{back} of w^{forw}_{back} (compare the context and compatibility tables in Fig. 3). Fig. 4(b) already shows that the given system is not R-diagnosing and that the hypotheses are pairwise-dependent.

For a complete analysis of the system, all correction areas of different length are to systematically be constructed. This requires that all combinations of symbols be constructed that "do not fit together," i.e., which cannot be neighbors in a correct string. Correction areas of length *l*=1 are obtained when all symbols are placed between neighbors that cannot be their left or right context. For the term *T* from example 2, e.g., in the strings a^2ca_6, b^3ca_6, and b^5ca_6, the symbol c is not in correct context with its neighbors. Correction areas of length *l* = 2 are formed by the combination of all symbol pairs (s^i,t_n) that are mutually not in context, i.e.,

$$s_j \notin l_{back}(t_n), t^m \notin r^{forw}(si), \text{ for all } j,m.$$

In this way, the Q-diagnosis and correction ($Q \in \{G, H, J\}$) capabilities of the system can be analyzed by means of its corresponding RE. The procedure can be generalized to the case of arbitrary length of correction areas and arbitrary *n*for Q^n. This enables to canonically analyze and compose the fault tolerance properties of the system in the design phase, providing the basis for improving its fault tolerance by use of additional redundancy.

D. Extending the Regular Expression for Automatic Fault Handling

Errors cannot be unambiguously diagnosed and corrected if the analysis of an RE reveals that the system is not Q-diagnosing, or Q-correcting, or that hypotheses are mutually dependent. This is illustrated by the example in Fig. 4(b). Since there are two positions where the hypothesis H might apply, a unique R-diagnosis and an R-correction are not possible.

The general idea how to avoid this problem is to embed in the regular expression at least one of the four symbols (*s*, *u*, *t*, or *z*; or *v*, *x*, *y*, or *k*) into a different, appropriate left of right context, e.g.,

$$s^i_j \rightarrow as^i_jb$$

so that the R-correction can only apply to one position. This process is called the *extension of the regular expression.* The extension yields fault-tolerance system behavior if the introduced redundancy is sufficient.
The symbol embedding can be systematically performed to remove the considered ambiguities from the correction areas. It does not imply the necessity of additional activities.

E. Generalization to Arbitrary Fault Condition

Using the above explained principles, all corrections areas are now to systematically be constructed and analyzed where a Q-correction is not possible because of ambiguities in detectability and/or correctability, or pairwise independence of hypotheses. As a next step, all of the symbols s^i_j, u^k_l, t^m_n and z^i_l (Fig. 4(a)) in those correction areas are to be collected into a set *S*. The objective is then to systematically reduce the number of the correction positions and the correcting symbols in each of the correction areas to exactly one by appropriate extensions to exclude detecting/correcting ambiguities and hypotheses. It is evident that this objective defines an optimization process where the number of extending symbols may differ with regard to the number of eliminated correction positions, correcting symbols, etc.

To avoid unnecessary redundancy, the number of the extended symbols in the sequence should be as small as possible. Therefore, the set *S* is to be minimized to achieve a specified fault-tolerance property.
For theexample here, it can be shown that R-diagnosability is achieved only by extending the symbols a^2_2, a^6_6,and b^5_3. To realize this, we can use, on the one hand, embedding symbols already existing in the string. In our case, the extension then could be

$$a^2_2 \rightarrow a^2_2a, a^6_6 \rightarrow ba^6_6b, b^5_3 \rightarrow ab^5_3.$$

On the other hand, to mark the positions where redundancy has been included into the string, it is often useful to introduce new embedding symbols. Choosing "$," "_," and "&" as additional symbols, then the extension could be performed as follows.

Using these, we obtain the extended expressions:
T1=(bbab(b+c))*(aa+ab)*or T2=(b#a#(b+c)*=*(a$+&b)*.*
which are both R-diagnosing.

The concepts shown here for the use of Q-corrections can be generalized to the cases of Q^n-hypotheses (i.e., *n* errors have to be simultaneously corrected). The basic idea thereby is to transform the simultaneous *n*-fold corrections into n sequential steps.

The addition of the redundancy to the system may be performed stepwise so that the degree of redundancy can be adapted to the required error detection/correction properties. As a further advantage of the method, the detection and correction of errors at run time can be performed by a straightforward algorithm without any backtracking procedures.

A description of the tools that have been developed to apply the method in a comfortable environment can be found in [9].

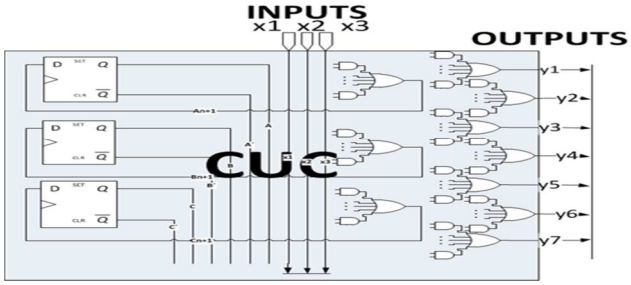


Fig. 5. Sequence loader as CUC

IV. A CASE STUDY AND EVALUATION

Fig. 5 depicts the block diagram of the circuit C as a CUC, which is a sequence loader borrowed from a real-life project. Using the well-known algorithms [3,4,7,8], this circuit will be transformed into an FSA and, subsequently, to following RE.

$$T = [(ba(b + c)^*)^*(a + b)^*] \qquad (3)$$

Symbols in *T* represent well-determined triples of inputs, x_1 to x_3 of *C*, i.e., *[*:=000; *a:=001*; *b*:=010; *c*:=100; *]*:=111(see[4,7,8]).

T represented in (3) is exactly the same RE used as an example (1) in Section 3.2 to explain the approach. For R-detecting and R-correcting,*T* has been extended to

$$T^* = (bbab(b+c)^*)^*(aa+ab)^*$$

Thus, the extended RE, that is, *T**, is now to be converted into a circuit via a corresponding FSA using the tool JFLAP [9]. Finally, this FSA is converted into an extended circuit*C** by means of the State Transition Table and K-maps (Fig. 6).

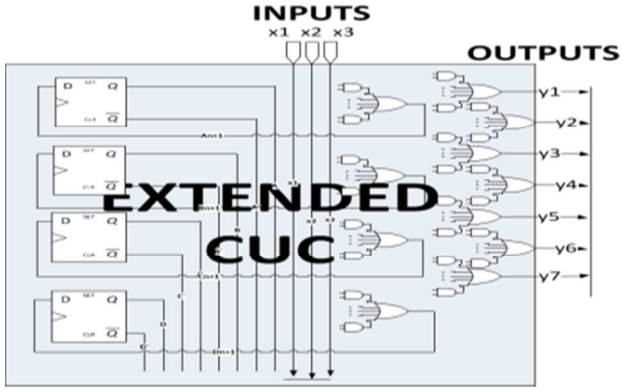


Fig. 6. Extension of the given CUC

Outputs of *C* and *C** are evaluated for equivalency of input sequences. In addition, the behavior of *C** is also observed for the test sequence$t_n$in (3) in Section 3.3. It is assessed that both circuits produce equivalent outputs. Consequently, *C* and *C** are considered as equivalent for operational purposes. Finally, because *C** satisfies the desired properties, it is stated that *C** is both R-detecting and R-correcting.

Example: t_n*, the extended form of the example test sequence t_n, (see (2) in Section 3.3) allows only one R-detection and correction.

$$t_n^* = [{}^1_f a^2_f b^6_f a^2_f b^6_f a^2_f b^6_f a^2_f a^2_f c^e_4 a^e_2]^e_1$$
$$\text{R}.....a^2_6 \Rightarrow \in L(T^*)$$

CUC and CUC* were implemented in XC3S100E from Spartan-3 FPGA family. Xilinx ISE Project Navigator was used for the design, implementation, and synthesis of the circuits. An overhead comparison of the circuits CUC and CUC* with TMR and structural BIST is under work.

V. CONCLUSION AND FUTURE WORK

Based on finite-state automata and regular expressions (RE), an approach is introduced to analyze fault handling features of a given sequential circuit. If required, the redundancy necessary to adopt a desirable feature for fault detectability, correctability, etc., can be determined and added to the given circuit. Novelties of the approach stem from the analysis and extension of circuits for automatic fault handling.

Problems we have presently been working on are (i) generalization of fault models to consider a broad class of faults encountered in the practice, and (ii) optimization of the redundancy necessary for extension, considering also events that are not included in the original RE that models the CUC.

Comparisons with similar approaches using estimated values are under work.

Last but not least, the present approach is to be applied to large scale, complex circuits in order to check to what extent it can cope with the state space explosion.

ACKNOWLEDGMENT

The author would like to thank Onur Kilincceker and Uras Tos, who implemented the circuits and worked out the experiments.

REFERENCES

[1] M. Abd-El-Barr, Design and analysis of reliable and fault-tolerant computer systems. Imperial College Press, 2007.

[2] F. Belli, Extension of Regular Languages for Self-Detecting and Self-Correction of Syntactic Errors (in German), R. Oldenburg Verlag, München, 1978.

[3] J.A. Brzozowski, "Regular Expressions from Sequential Circuits," *IEEE Trans. Electronic Comp.*, vol. 13, 741 - 744, 1964.

[4] J. A. Brzozowski, and McCluskey Jr., E. J., "Signal flow graph techniques for sequential circuit state diagrams," IEEE Trans. on Electronic Comp., vol. 12, 67 -76, 1963.

[5] R. W. Floyd, and Ullman, J. D., "The Compilation of Regular Expressions into Integrated Circuits," J. ACM, vol. 29, 603-622, 1984.

[6] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Languages and Computation, 3rd Ed., Addison-Wesley, 2006.

[7] M. Morris Mano, and Charles R. Kime, Logic and computer design fundamentals, Upper Saddle River: Pearson Prentice Hall, 2008.

[8] Ott, G., and Feinstein, N.H. "Design of Sequential Machines from Their Regular Expressions," *J. ACM*, vol. 8, 585-600, 1961.

[9] S.H Rodger, Th. W. Finley, Jflap-an Interactive Formal Languages and Automata Package, Sudbury, MA: JonesBartlett, 2006.

[10] A. Strano, D. Bertozzi, A. Grasset, and S. Yehia, "Exploiting structural redundancy of SIMD accelerators for their built-in self-testing/diagnosis and reconfiguration," Proc. IEEE ASAP, 141-148, 2011.

[11] A. Strano, C. Gomez, D. Ludovici, M. Favalli, M. Gomez, and D. Bertozzi, "Exploiting Network-on-Chip Structural Redundancy for A Cooperative and Scalable Built-In Self-Test Architecture," Proc. DATE, 2011.

[12] Y. Zhang, and J. Jiang, "Bibliographical review on reconfigurable fault-tolerant control systems," Annual Reviews in Control, vol. 32, 229–252, 2008