Automated Dysfunctional Model Extraction for Model Based Safety Assessment of Digital Systems

Tiziano Fiorucci, J.M. Daveau, Giorgio Di Natale, Philippe Roche

Automated Dysfunctional Model Extraction for Model Based Safety Assessment of Digital Systems

Tiziano Fiorucci[1,2], Jean-Marc Daveau[1], Giorgio Di Natale[2], Philippe Roche[1]

[1]STMicroelectronics, 850 Rue Jean Monnet 38926 Crolles Cedex, France
[2]Univ. Grenoble Alpes, CNRS, Grenoble INP*, TIMA, 38000 Grenoble, France

***Abstract*—In the field of automatic quality or safety assurance level evaluation, this paper proposes the first approach towards the automation of the extraction processes of both the valid and faulty state machines within a System-on-a-Chip. The data automatically extracted by this method is a relevant input for behavioural modelization and FMEA analysis. The method is based on a semi-automated approach for the systematic extraction of failure modes of a digital design in the hypothesis of a single-event upset (SEU) or stuck-at in flip-flops. This procedure aims to enhance human driven failure analysis and provide inputs for RAMS frameworks in the process of quality assurance of complex devices. The main objective is to transport and apply RAMS methods and tools in the area of SoCs design. Experimental results have been conducted on an I2C - AHB system, laying the base for a complete and more complex analysis on an entire SoC.**

I. Introduction

The classical and well established quality or safety assurance process relies on complex and systematic methods, needing as input the complete description of every block of the studied system. In particular, the set of failure modes and consequences is required for each block, in order to compose them the system's behaviour in case of faults. This process is usually man driven, such as in the case of System on Chip or performed using proven Reliability, Availability, Maintainability, and Safety (RAMS) methods in the area of mechatronics or electromechanics systems [15]. Model Based Safety Assessments (MBSA) ensure the safety assurance process using individual block failure models [10] and a composition framework based on model checking methods. Such approaches are widely used in critical software development with tools such as [18] to prove software correctness or build correct software [11] but not in the domain of digital System-On-a-Chip (SoC), although attempts have been made [9].

The Failure Mode and Effects Analysis (FMEA) of a SoC or a single Intellectual Property (IP) is still mostly human based, prone to a series of errors and omissions inherent to the human reasoning, inference and encompassing process. The main hard point of applying MBSA methods to SoCs is building the failure models for each IP composing it. Because safety models are far from the specifications of digital IPs behavioral models, a first step towards the automation of the construction of such a failure model is the automated exploration of the faulty state-space and extraction of its safety relevant behavior.

Attempts have already been put in practice on rather simple systems, relying on a human based library of components describing possible malfunction [9] [27]. However, such models are far from the complete dysfunctional model specifications and only macroscopic failures are considered, thus not leaving space to unexplored combination values in registers that may lead to unexpected faulty states and behaviours. For example, a 0 to 4 counter, which needs a 3-bit state register, may go out of it functional range in case of a bit flip. The state "$1-0-1$", in Fig.1, is a non-explored and not explicitly declared state. It may occur due to a faulty combination of values in the flip-flops storing the state.

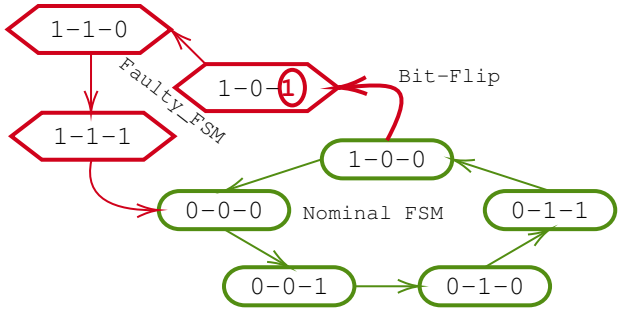

Fig. 1. Non Explicit State Exploration Example

Exploring non-functional states in a digital system is far from human encompassing capabilities, even for systems with a small number of flip-flops. The approach proposed in this paper aims at filling the gap between digital IP behavioral model and generation of a dysfunctional model for FMEA evaluation, providing a model that can be used in a MBSA framework, thus, targeting safety assessment of the full system from its individual IPs. In this work we target tools based on the Altarica safety modeling language [25].

In the proposed approach, digital fault injection, in the form of bit-flip, stuck-at and transient faults, is used to extract the non-functional behavior of the studied digital block from its functional model. The extraction of those data allows building a failure model that includes the propagation of errors to and from inputs and outputs, thus enabling structural composition. We show that it is possible to extract a failure model in the form of a state machine describing the faulty behavior with scalable level of details and ensuring the fault propagation to (resp. from) outputs (resp. inputs).

The paper is organized as follow: In Section II we provide some background, including a brief description of the Altarica language. In section III we describe the general methodology to extract faulty behaviour. In section V we apply faulty state extraction to two relevant examples and present the extracted faulty state automaton. In section IV-C we perform faulty state extraction on the complete system, proving the scalability of the model, allowing the composition of more complex systems without missing faulty states. Future work and conclusion are presented on sections V-F and VI.

II. State-of-the-Art

Several attempts towards fully automated FMEA are reported in literature. They can be divided into 3 categories based on the main idea that drove the approach to the problem: (1) Fault injection based, (2) manual-developed libraries approach, and (3) formal netlist verification methods. In the

*Institut National Polytechnique Grenoble Alpes

first category, the work in [24] aims to create primitives for a different standard [IEC61508], starting from a fault injection campaign and analyzing the results to evaluate the FMEA of a safety critical SoC, in order to evaluate the compliance to the standard. In the second category, the works in [9] and [27] have developed a framework for behavioral modeling a SoC (then being able to extract the FMEA from there) but starting from a library of elementary blocks, human written and prone to errors, which do not assure the complete FSM coverage for the blocks under test. In the last category, the works in [19] [7] [12] [26] have tackled the problem from a different point of view, trying to formally verifying the netlist of a specific circuit and then build a translator from Verilog to CLU, the language utilized to verify control and mixed (data/control) paths.

In [20], the behavior of system components are specified by UML (Unified Modelling Language) state machines determining intended/correct and undesired/faulty behaviors. The UML state machine description represents both nominal behavior of the component but also the failure modes though dedicated states (called *failure* states). The behavior of the component in each state is defined using the Object Constraint Language (OCL). The user then specify top nodes of the fault tree (state combinations at the system boundaries) and sequences of events composing the fault tree are computed and expanded. In [8] a reverse approach is followed were fault trees are converted and integrated into the *statechart* behavioural model of the system under evaluation.

In this work we focus on tools based on the Altarica language [25], which is a high level formal modeling language dedicated to safety analysis. It can be seen as a generalization of Petri nets for the behavioral part, and block diagrams for the structural part. It borrows to Petri nets the notion of states, events and guarded transitions and to block diagrams, the notion of hierarchical descriptions and flows circulating through a network. Starting from such dysfunctional models, fault scenarios leading to a specified set of unexpected states can be computed and quantified to determine the probability of such behaviours. Automatic generation of reliability models such as fault tree, event tree, markov chain or monte-carlo simulation models for use in reliability assessment tools can be performed. Framework such as SimfiaNeo [14], [22], based on the Altarica language, belong to that category.

III. Methodology

On top of any explicit finite state machine or control code encoding the user specified behaviour, it is possible to build an *extended* state including the totality of the signals belonging to the control path of a design. These signals compose a more complete and larger state machine exposing new states and transitions that are not explicitly specified. Combinations of these signals in these states can lead to a subtle set of fault states, difficult to identify from the HDL description as the encoding in this state machine is sparse due to correlations. Such argument can be strengthened by the fact that even for a small ($>\approx$50) number of flip-flops, the complete state space (2^{50}) cannot be traversed in a reasonable time. Therefore a non-negligible proportion of these states are what we call *faulty states*, potentially leading to undesired or unspecified and faulty behavior.

In order to build a failure model from a nominal behavioral Register Transfer Model (RTL) in Verilog or VHDL, behaviour of the system under faults must be analyzed and faulty behavior as well as failure modes must be extracted. We proceed using the following steps:

1) **Identification and Extraction of Control Signals** - Starting from the functional description, the entire set of possible values in the flip-flops of the control path, composing what we denominate as the *state*, has to be identified and extracted. In digital systems, the state is easily determined and is composed by all the flip-flops composing the control path and possibly the datapath in some cases. This set correspond to both the *control* (and possibly *data*) state of the system and possible fault injection sites, as described in Fig.2.

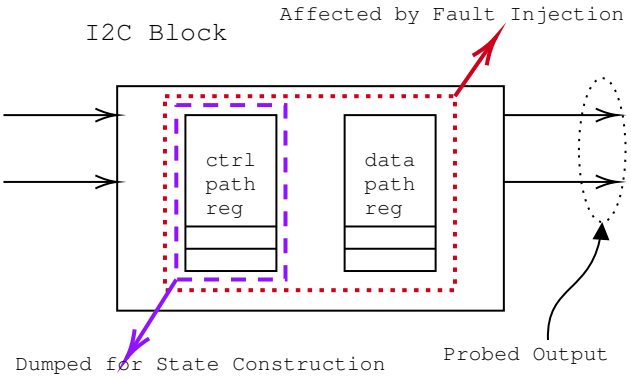

Fig. 2. scheme of probes placement and affected/dumped registers

2) **Testbench Setup** - A standalone testbench is set up and special care is given on evaluation of the coverage and testbench representativity as the states traversed during the golden execution will serve as non faulty behavior reference. Tools like *Incisive Metrics Coverage* (IMC) [1] or *Certitude* [5] can be used to assess testbench coverage. A first golden run is performed as reference run to allow extraction of functional states that will be used later in the process to be differentiated from non-functional ones.
3) **Fault Injection Campaign** - Fault injection is the mean by which the misbehavior and faulty execution is provoked on purposes. Probes (i.e., observation points) are defined during the setup of the fault injection campaign. They are set on the outputs of all blocks in order to identify failures that propagates to other blocks. Probes monitor and compare the probed signal value at each clock cycle with the golden reference and report any difference. They have been set to stop simulation when a fault reaches an output of the design. This step is the core of our analysis aimed at extracting faulty behaviour, modes and effects though exploration of the faulty states by fault injection.
4) **Extraction of Faulty Behavior** - Once the faulty runs have completed, non-functional (i.e. *faulty*) states and behavior are extracted by subtracting functional states from the state dictionary taken from the golden run to the faulty run states, leaving only newly discovered faulty states and transitions.
5) **Construction of the Faulty Model** - The newly discovered states and transitions are used to augment the functional models with faulty behavior. Transitions from a functional to a non-functional state are labeled with the responsible fault such are states responsible for an incorrect output. This model serves as a base for the translation into the Altarica language.

Currently, the method is limited in the *effect* analysis of the FMEA. Effect such as *loss of power* cannot be attached automatically to a faulty state as it would requires an inference and abstraction process out the reach of the tool currently. Thus, such labelling is performed manually by attaching the *Effect* (of FM***E***A) to the output and then back-propagating it to

the states and faults responsible for the given output corruption.

IV. Test Case

In order to detail the methodology presented in Section III, we use a test case composed of 2 blocks: an *I2C* slave [23] connected to an *AHB* [16] bus master interface. Commands (*read* or *write*) along with parameters (*address* and *data*) are received on the serial line and transformed into a series of AHB read and write transactions. Such a system, composed of two interconnected blocks, is humanely understandable so are its dysfunctional modes, while being complex enough to detail thoroughly the methodology.

The I2C slave, taken from [4], receives *read* or *write* commands followed by an address byte and an optional data byte. On an I2C read, the byte returned from the AHB read transaction is returned. Chronograms for the read and write sequences are represented on figure 4. The system is represented on figure 3. At both end of the system (I2C and AHB buses), verification IPs (VIP) are attached to generate and verify I2C and AHB transactions. An I2C Master and AHB slave VIP are used.

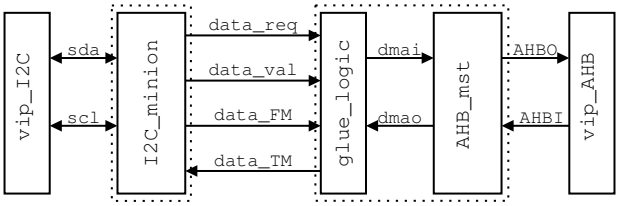

Fig. 3. Block Diagram of the System under Evaluation

A. I2C Test Case

The testbench is composed of a series of read and write requests. The coverage evaluation of the design has been carried out using both Modelsim FSM coverage evaluation tool and IMC. The results are presented in Table II and I. Having considered the results of the two coverage evaluations sufficient, the attention has been moved onto the application of the method presented in section III. The list of all the injection sites, reported by Cadence *Functional Safety Verification* tool are considered for state including ones containing data as the serial nature of the I2C protocol mixes control and data on the same signals. All outputs are probed so that any mismatch with the reference run will stop the simulation and report the fault as *detected*. State (flip-flop value, i.e. '0' or '1') is simply extracted at each clock cycle and printed in the simulation logs.

B. AHB Interface Test Case

The AHB bus interface is taken from the GRLIB [2] library with added custom logic to be connected to the master parallel interface of the I2C. The added logic comprise an interpreter for the command received by the I2C and the glue interface to the GRLIB AHB master block. A verification IP is connected to the AHB interface side to respond to transactions and check protocol. Figure 4 represents the translation of the I2C signals in the appropriate AHB transaction request signals. Coverage for AHB block is low and can be explained as only a limited use of the AHB protocol is made:

1) only byte access are performed.
2) only single (SINGLE) non-sequential (NONSEQ) transfer are performed.
3) the VIP has not been programmed to insert HREADY wait states in the transaction.

	I2C			AHB		
	cov.	tot.	overall	cov.	tot.	overall
Overall	352	410	93.8%	333	1057	61.3%
block	164	180	95.4%	51	68	85.55%
Expression	44	44	100%	7	17	41.18%
Toggle	112	148	75.68%	266	963	30.17%
FSM	32	38	83.36%	9	9	100%

TABLE I
Coverage Figures reported by IMC (%)

	Total	Covered	%
State Coverage	8	8	100
Transition Coverage	25	13	53

TABLE II
Coverage Figures for the I2C block reported by Modelsim (%)

4) the VIP has not be programmed to generate HRESP transaction response error.

The low coverage obtained here doesn't restrict the generality of the methodology but may prevents failure mode to be identified in this specific case.

C. Complete System Test Case

The complete system is composed of both the I2C slave and AHB master along with both VIPs at the end. As previously mentioned, probes are set on the output of the complete system, leaving this time, faults freely propagating internally between the I2C and the AHB without being reported by FSV or the simulation to be stopped. The main difference of this testbench regarding the two standalone previous ones is that faults injected in one block will be able to propagate to the other one (I2C $\rightarrow$ AHB, for example) and back-propagate to the first block (AHB $\rightarrow$ I2C) as simulation will not be stopped when the fault will output from the first (i.e. I2C), and later second (i.e. AHB), block. Such "fault loop" (I2C $\circlearrowleft$ AHB or AHB $\circlearrowleft$ I2C) are expected to be the main possible source of faulty states differences between the standalone and full system faulty states extraction. However, as faults are injected on the inputs in both approach (standalone and full system), we expect to capture, at least a part of theses "faults loop" induced faulty states in the standalone extractions.

V. Application

In this section, we apply the methodology presented in section III to the two blocks of our test case one after the other, constructing a failure model for each one.

A. Identification and Extraction of Faulty States

As stated in section I, all digital systems can be represented as a finite state automaton, where the state is composed by all the flips-flops of the system, whether they maintain a control or data state. In this approach we consider, in a first approach, only control states with the following justification: faults (bit-flip) in datapath may not propagate in control states nor even create a faulty control state. Therefore faults in data states (that is flip-flops) will not lead to faulty behavior unless some data states are transformed directly into control states and encoding is sparse (some data states do not correspond to any control state and will therefore result in faulty state unless handled explicitly handled by a *default* case in the design).

On our example, analysis is performed at the RTL level, and the (signals composing the) state have been identified

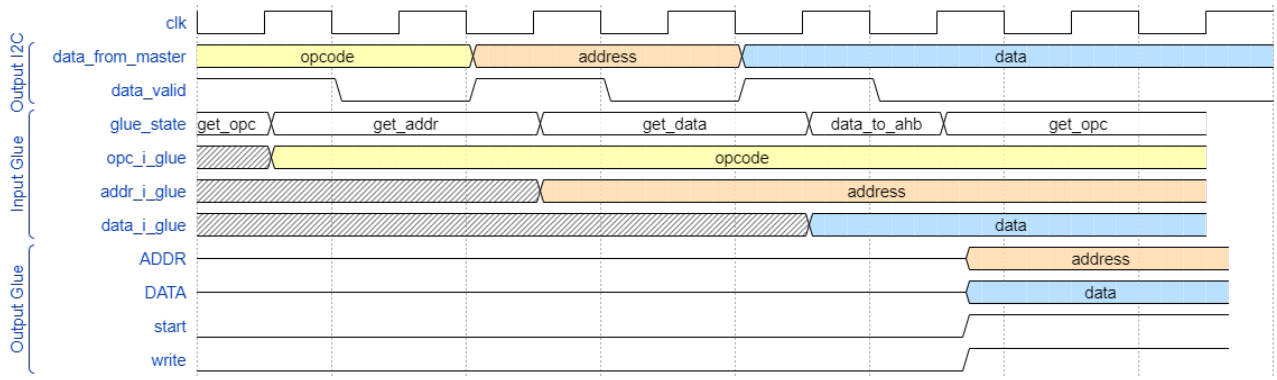

Fig. 4. Chronogram of a communication between I2C and AHB

easily considering the small size of the design. As the I2C protocol makes use of a serial line, data states and control states are merged when they can't be completely differentiated. On the AHB interface block, data states are excluded as no path exists from a data signal to a control signal (the reverse being obviously not true).

From a general perspective, analysis should be performed at the gate netlist level to ensure correct extraction of the control and data states with the drawback of slower fault simulation. Also data states can be pruned from the whole state using graph netlist forward (from data input) and backward (from data output) propagation algorithms. However, fault injection tools operating at the RTL level such as FSV [1] and ZOIX [6] do perform a pre-synthesis step to identify potential injection sites, that is flip-flops.

Once signals composing the state of a block have been identified, using a standalone testbench which can be derived from verification ones, a golden run is performed and golden states and transitions are recorded at each clock cycle. Such states and transitions are referred as *legal* composing the *non-faulty* or *golden* behaviour. Results for I2C and AHB blocks are reported in table III. The golden state automaton for the I2C is represented on figure 5.

B. Fault Injection Campaign Setup

The next step in the methodology is fault injection. It is performed using Cadence fault injection tool *FSV* [1]. Once fault injection sites are automatically identified from the RTL description, fault injection is performed and 400 faults are injected per identified site using a custom pre-generated fault dictionary. An in-house tool build on top of the *GSL* [3] has been developed for this purpose. Such number is statistically significant enough [21] without compromising fault injection campaign running time. Faults are also injected on inputs to take into effect of faults propagated from other blocks during composition. Also, fault probes are set on the outputs to record injected faults that will propagate to other blocks. For each fault injection run, states are recorded to identify new states and transitions that appear as a consequence of the injected faults. The new discovered states and transitions are referred as *illegal* or *faulty*.

In our modeling approach, we are interested only in states and transitions and we omit executions paths that are the ordered list of transitions traversed during a golden or fault run, even though it is available from the extracted data. The reason is that the faulty behaviour modelling strategy targeted, based on the Altarica language, doesn't requires such information. Thus only new faulty discovered states and transitions are extracted from execution runs and faults leading to an illegal execution path (list of traversed transitions) containing only *legal* states and transitions will not be reported as a faulty behavior unless an incorrect output is reported during simulation.

In order to achieve this status during the nominal execution of the testbench, the IP has been modified, adding dedicated code to write, into a log file, the state vector that includes all the selected control signals. This allows to have a real time, event driven, transition set in between the different combinations of the observed signals. The created log file will be crucial to the rest of the procedure, that will take it as input.

C. Faulty Behavior Extraction

Once the raw data have been dumped by the simulator in the log files (for a total of 12000 files equal to 400 faults $\times$ 30 flip-flops for the I2C and 42000 files equal to 400 faults $\times$ 105 flip-flops for the AHB), it is necessary to extract the values of the signals composing the state, records them and keep trace of all transitions. In order to complete this task, a python script using the *NetworkX* library [17] has been written to create a dictionary containing all the different states and their occurrence count, as well as another dictionary with all the transitions, to perform statistics and extract the faulty behavior model.

An example of raw log is given on listing 1. States and transitions extraction from the logs is straightforward and requires only one pass per log file. An example of extracted fault behavior is represented on figure 5 for one flip-flop. Extracted automaton characteristics are reported on table III.

	I2C Write	I2C Read	AHB	I2C + AHB §
State width (FF in state)	19	19	38	19+38
Fault site (i.e. FF)	30	30	105	133
#Faults per FF	400			
Total injected faults	12000	12000	42000	53200
Nominal states	36	36	10	55 (44+11)
Nominal transitions	71	71	18	108 (90+18)
Faulty states	20	53	15	156 (95+51)
Faulty transitions	39	144	45	302 (213+89)
Faults to outputs	34	147	19934	23496
Flow through faults†	3	0	11900	14900
Simulation timeouts	11967	231	0	0

†: *Flow through faults* are faults injected on inputs propagating to outputs

§ : *The extra states found in this column are not belonging to the control state but are the result of mixed control and data path on the same registers*

TABLE III
AUTOMATONS REPORTS

Once the golden and faulty states and transitions have been extracted from the logs, the failure model can be built by collapsing states and transitions into the desired ones for the

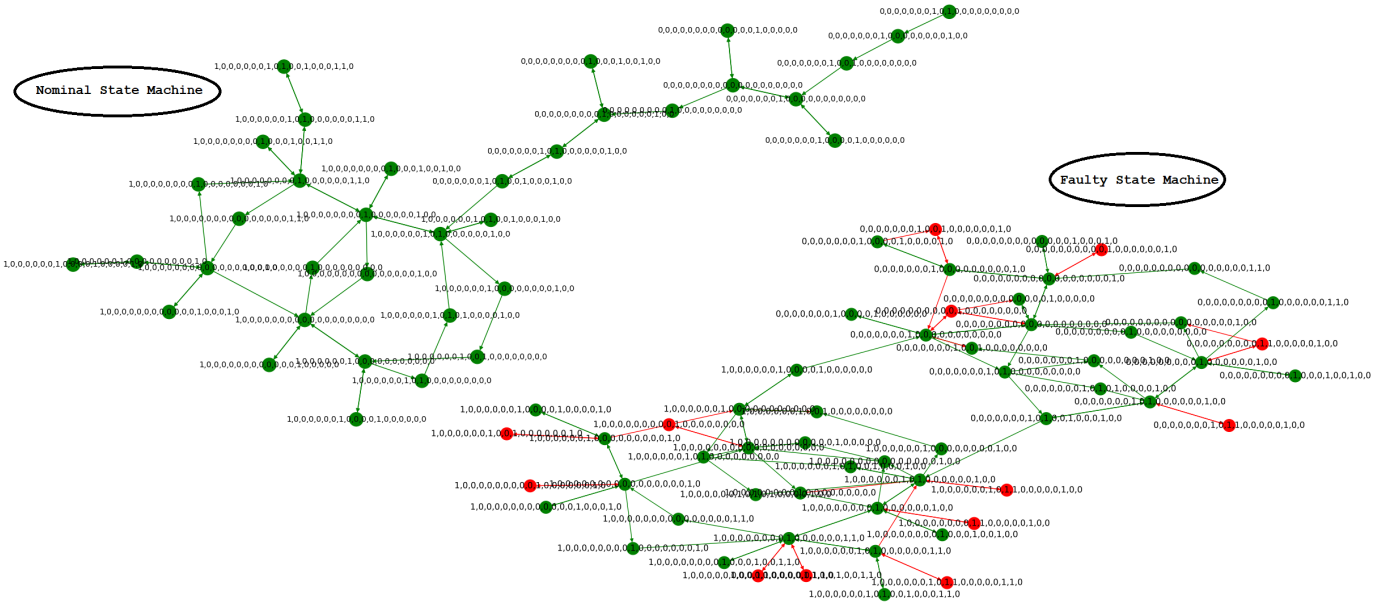

Fig. 5. Golden and Faulty Automaton I2C

Listing 1. Raw log from I2C FSM extraction

```
 --- Testing repeated reads ---
out:0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0
out:0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0
```

model. The raw faulty automaton (before node collapsing) for the I2C block is partially represented on figure 5 where legal states and transitions are represented in green and illegal ones in red.

D. Faulty Behavior Model Construction

Once the faulty behavior has been extracted from faulty runs, the faulty model can be constructed using graph analysis algorithms. The first step in the model construction is collapsing states that are not meaningful for the dysfunctional model. We proceed currently with the following rules:

- any component (connected subgraph) comprising only legal states and legal transitions are collapsed into one single *functional* state.
- legal states with illegal transitions or incorrect output (output value do differs from reference in these states) are kept and illegal transition probabilities are attached.
- any component comprising only nodes not propagating any fault to output are collapsed into one single *faulty* state. Probabilities to enter this state can be extracted from transitions leading to the collapsed states.
- faulty nodes propagating faults to outputs are kept and transition probabilities are attached to allow computing incorrect output probabilities.
- Effect (*E* of *FMEA*) attached to output pins are back propagated in the state graph faulty states where output corruption occurs.

However additional rules may be added like to remove faulty nodes and transitions from masked faults for example, especially those not leading to any latent faults (execution is correct with no faults propagated to outputs and internal state doesn't differs from reference one at some point, i.e. fault has *vanished*). We ultimately target discrete-time Markov chain [13] for our dysfunctional behaviour modelling.

E. Completeness of the Extraction

We verify that standalone extraction of the faulty model is complete, that is no new faulty behavior appear when the DUT is integrated and studied in the complete system. Thus the same stages as depicted in section III are performed on the complete system composed of the I2C+AHB system.

F. Result and Comparison

Data gathered by the application of the methodology is then elaborated starting from data shown in table III and the dictionaries collecting all states and transitions, created at faulty behavior extraction phase.

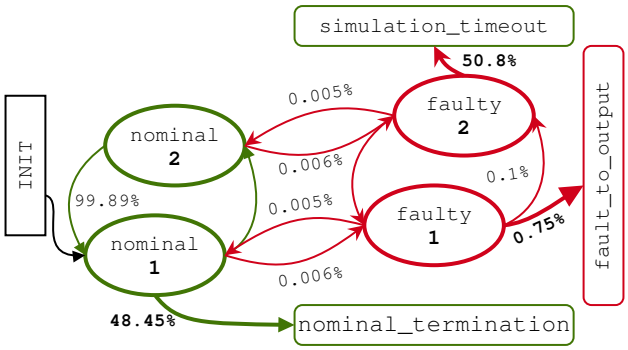

Fig. 6. Extracted FSM for the I2C Block

The result is the model shown in Fig 6, belonging to the I2C block, follow the rules listed in section V-D showing the actual probabilities of transition between the connected subgraphs, being:

- `Nominal 1` - Subgraph made of legal states only, part of the nominal execution.
- `Nominal 2` - Subgraph made of legal states only, part of the nominal execution.
- `Faulty 1` - Illegal state Subgraph, leading to a propagation of the fault to the output.
- `Faulty 2` - Illegal state Subgraph, leading to a simulation timeout.

This kind of dataset is prone to be described in languages and tool like Altarica, as described in section II and object of future work.

VI. Discussion and Future Work

In this paper we have set-up an experimental method for automatic extraction of failure behavior for digital IPs. We have shown experimentally, using fault injection, that it can be used to build a dysfunctional model suitable for composition though fault propagation. This step represent the first step toward building a complete automatic FMEA analysis of a digital block for composition in an MBSA framework. The next steps comprehend the generation of the Altarica dysfunctional models based on these results and their composition in the SimfiaNeo framework.

Currently, the approach is mostly manual in it implementation (identification and instrumentation of extracted *state* flip-flops, extraction of state through simulation logs) thus limiting the size and complexity of the design that can be handled. Also it suffer from the inherent limitation of *behavior* extraction based on test (and statistical fault injection) execution which rely on testbench representativity to exercise correctly all states of the design. The first limitation can be removed by automation of control state identification as mentioned in section V-A and splitting large design into smaller ones to be re-composed. The state size, which could be foreseen as problematic in large IP is actually expected to stay of reasonable size as most flip-flops in a design are usually *data* flip-flops not part of the state. Systems like a processor that can re-inject a data state into the control state (through a *jmp register*, for example) will require special care to avoid state explosion through proper cut of the data$\longrightarrow$control paths. The second limitation, inherent to verification, can be lessened by ensuring proper coverage of the stimulus used to exercise the design.

VII. Acknowledgment

This research has been possible thanks to the help provided in trouble shooting by Pietro Inglese from TIMA Laboratory/Univ. Grenoble Alpes and Daniel Thirion from STMicroelectronics/ESISAR-Valence. The authors wish to thank Emmanuel Arbaretier, Thomas Jacquet and Julien Niol from APSYS/AIRBUS for their collaboration in this project.

References

[1] Cadence. https://www.cadence.com/en_US/home.html.
[2] Grlib. https://www.gaisler.com/products/grlib/grlib-gpl-2020.4-b4261.tar.gz.
[3] Gsl, gnu scientific library. https://www.gnu.org/software/gsl/.
[4] I2c minion repository. https://github.com/oetr/FPGA-I2C-Minion.
[5] Synopsys certitude. https://www.synopsys.com/verification/simulation/certitude.html.
[6] Synopsys zo1x. https://www.synopsys.com/verification/simulation/z01x-functional-safety.html.
[7] Z. S. Andraus and K. A. Sakallah. Automatic abstraction and verification of verilog models. In Proceedings. 41st Design Automation Conference, 2004., pages 218–223, 2004.
[8] O. Ariss, D. Xu, and W. E. Wong. Integrating safety analysis with functional modeling. IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans, 41:610–624, 2011.
[9] R. A. Austin, N. Mahadevan, A. F. Witulski, G. Karsai, B. D. Sierawski, R. D. Schrimpf, and R. A. Reed. Automatic fault tree generation from radiation-induced fault models. In 2020 Annual Reliability and Maintainability Symposium (RAMS), pages 1–7, 2020.
[10] M. Batteux, T. Prosvirnova, A. Rauzy, and L. Kloul. The altarica 3.0 project for model-based safety assessment. In 2013 11th IEEE International Conference on Industrial Informatics (INDIN), pages 741–746, 2013.
[11] F. Boussinot and R. de Simone. The esterel language. Proceedings of the IEEE, 79(9):1293–1304, 1991.
[12] B. A. Brady, R. E. Bryant, S. A. Seshia, and J. W. O'Leary. Atlas: Automatic term-level abstraction of rtl designs. In Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010), pages 31–40, 2010.
[13] Yen-Chi Chen. Discrete-time markov chain, 2018.
[14] Xavier de Bossoreille, Mathilde Machin, and Laurent Sagaspe. Un Nouvel Outil de Safety pour Maitriser la Complexité des Systèmes. In "Maîtrise des risques et transformation numérique: opportunités et menaces, Reims, France, October 2018.
[15] J. Farquharson, R. Gallman, and B. King. How to assess ram for a system-of-systems (s-o-s) in military applications. In Annual Reliability and Maintainability Symposium. 2001 Proceedings. International Symposium on Product Quality and Integrity (Cat. No.01CH37179), pages 278–284, 2001.
[16] P. Giridhar and P. Choudhury. Design and verification of amba ahb. In 2019 1st International Conference on Advanced Technologies in Intelligent Control, Environment, Computing Communication Engineering (ICATIECE), pages 310–315, 2019.
[17] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, Proceedings of the 7th Python in Science Conference, pages 11 – 15, Pasadena, CA USA, 2008.
[18] M. Hendriks, Wang Yi, P. Petterson, J. Hakansson, K. G. Larsen, A. David, G. Behrmann, M. Hendriks, Wang Yi, P. Petterson, J. Hakansson, K. G. Larsen, A. David, and G. Behrmann. Uppaal 4.0. In Third International Conference on the Quantitative Evaluation of Systems - (QEST'06), pages 125–126, 2006.
[19] Y. Ho, A. Mishchenko, and R. Brayton. Property directed reachability with word-level abstraction. In 2017 Formal Methods in Computer Aided Design (FMCAD), pages 132–139, 2017.
[20] Christof Kaukewitsch, Henrik Papist, M. Zeller, and M. Rothfelder. Automatic generation of rams analyses from model-based functional descriptions using uml state machines. 2020 Annual Reliability and Maintainability Symposium (RAMS), pages 1–6, 2020.
[21] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: Quantified error and confidence. In 2009 Design, Automation Test in Europe Conference Exhibition, pages 502–506, 2009.
[22] Mathilde Machin, Laurent Sagaspe, and Xavier de Bossoreille. Simfia-neo, complex systems, yet simple safety, 2018.
[23] Jayant Mankar, Chaitali Darode, Komal Trivedi, Madhura Kanoje, and Prachi Shahare. Review of i2c protocol. International Journal of Research in Advent Technology, 2(1), 2014.
[24] R. Mariani, G. Boschi, and F. Colucci. Using an innovative soc-level fmea methodology to design in compliance with iec61508. In 2007 Design, Automation Test in Europe Conference Exhibition, pages 1–6, 2007.
[25] T. Prosvirnova. AltaRica 3.0: Model-Based approach for Safety Analyses. Theses, Ecole Polytechnique, November 2014.
[26] S. A. Seshia and P. Subramanyan. Uclid5: Integrating modeling, verification, synthesis and learning. In 2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE), pages 1–10, 2018.
[27] A. Witulski, G. Karsai, N. Mahadevan, R. Austin, R. Schrimpf, B. Sierawski, R. Reed, J. Pellish, M. Campola, J. Evans, and P. Majewicz. Development of a flight-program-ready radiation model-based assurance platform. In 2020 IEEE Aerospace Conference, pages 1–8, 2020.