Automatic Generation of Failure Scenarios for SoC

Marco Marazza, Orlando Ferrante, Alberto Ferrari

Automatic Generation of Failure Scenarios for SoC

Marco Marazza
Dipartimento DIET, Sapienza - Università di Roma,
Via Eudossiana 18, 00184 Roma, Italy
e-mail: surname@diet.uniroma1.it

Orlando Ferrante, Alberto Ferrari
ALES S.r.l.,
Via Barberini 50, 00187 Roma, Italy
e-mail: name.surname@ales.eu.com

***Abstract*—As process technology downscales, testing difficulties and susceptibility of circuits to random hardware faults arise. This trend, combined with increasing complexity of functions to be performed by Systems-on-Chip, poses crucial concerns when system engineers have to quantify the dependability achieved by their SoC design. In this paper we propose an extension of the existing approaches to the fault analysis of SoCs describing (1) an algorithm for the automatic generation of failure scenarios based on Bounded Model Checking (BMC) (2) a methodology and Simulink-based tool for the automatic execution of SoC safety analysis and (3) an application of the proposed analysis flow to a concrete SoC use case.**

Keywords—Safety-critical Systems, Systems-on-Chip, Safety Analysis, Formal Methods

I. Introduction

System-on-Chip (SoC) design and testing complexity seems to fit very well a monotonic non-decreasing function of time. On the one hand, technology downscaling is causing considerable drawbacks, e.g. higher susceptibility of circuits to radiation-induced soft errors [1]. On the other hand, SoC functions are getting more and more sophisticated as SoCs spread over safety-critical domains such as automotive, aerospace, building automation, railway and medical. The trend of pervasively using SoCs to control physical systems directly interacting with human beings creates the need for new methodologies and tools to quantify the dependability – and in particular the functional safety– achieved by the system. This requires improving the capabilities of testing techniques and failure scenario identification for SoCs and represents one of the most crucial concerns for engineers and international certification authorities. The service delivered by a system is its behaviour as perceived by another system (human or physical) interacting with it. The deviation of the service delivered by a SoC from the specified service is known as a service failure. Random hardware faults introduce unpredictability to the services provided by the SoC, and can generate service failures. The typical design approach to control service failures, and consequently the system's functional safety, is to introduce redundancies to the system design in the space, time and information domains. However, even if controlled, still radiation-induced soft errors can cause residual service failures for a number of reasons: the techniques to control failures (1) might be based on models, which by nature suppress some information, (2) could cover a subset of the total set of occurring faults, (3) might only cover faults assumed to occur in specific use cases, (4) could have some limitations due to implementation costs. In summary, regardless of the efforts spent for techniques aiming at improving the dependability of delivered services, SoC services will be inevitably affected by residual failures due to soft errors. Standards like IEC 61508 [2] and ISO 26262 [3] introduce metrics and recommend methodologies and tools to quantify the Safety Integrity Level (SIL) of a SoC; for each SIL level the target value for metrics to be achieved by the design is defined and must be guaranteed. In this context, automatic generation of failure scenarios assumes fundamental importance to test the SoC design against fault models and to explore design solutions, including evaluation of the effectiveness of fault-tolerant mechanisms and cost trade-off. Several techniques and tools have been described in literature proposing model-based approaches to safety analysis. In [4] the application of a model based flow for the automatic generation of fault trees based on generation of minimal cut sets is shown. In [5] the SCADE formalism is used to perform Deductive Cause-Consequence Analysis (DCCA) using Design Verifier as formal engine. In [6] Joshi et al. describe some results on exploiting Simulink and SCADE for the safety analysis based on Simulink models. In [7] Bozzano et al. describe the SLIM language for the formalization of hybrid systems in presence of faults and an automatic fault analysis tool-flow using NuSMV. All the previously described approaches use high level models of the system under analysis for the automatic generation of fault-trees and binary decision diagrams (BDDs) as formal analysis techniques.

In this paper we propose an extension of existing approaches to perform automatic failure scenario generation during the design of complex System-on-Chip architectures. The main contributions of the paper can be summarized as follows: (1) an algorithm for the generation of Minimal Critical Failure Set (MCFS) based on Bounded Model Checking (BMC) is described (2) a methodology and a Simulink-based tool for the automatic generation of failure scenarios for SoC is described and (3) the application of the proposed analysis flow to a concrete SoC use case is presented and discussed. Our approach differs from the previous described ones in several ways. The algorithm is not founded on the construction of BDDs but relies on Bounded Model Checking (BMC) technique taking full advantage of the last advances in parallel SAT solving [8]. The use of BDD limits the applicability of existing techniques to models containing hundreds of Boolean memory elements whereas the use of Parallel SAT solvers in conjunction with BMC allows for successful algorithm application to industry-sized level problems tackling models containing thousands of Boolean variables. The analysis tool automatically explores all the minimal critical failure sets starting from a functional model of the SoC and the specification of components' failure modes relying on an internal model transformation engine [9] that automatically builds the error model starting from a functional formalization of the SoC and a list of failure

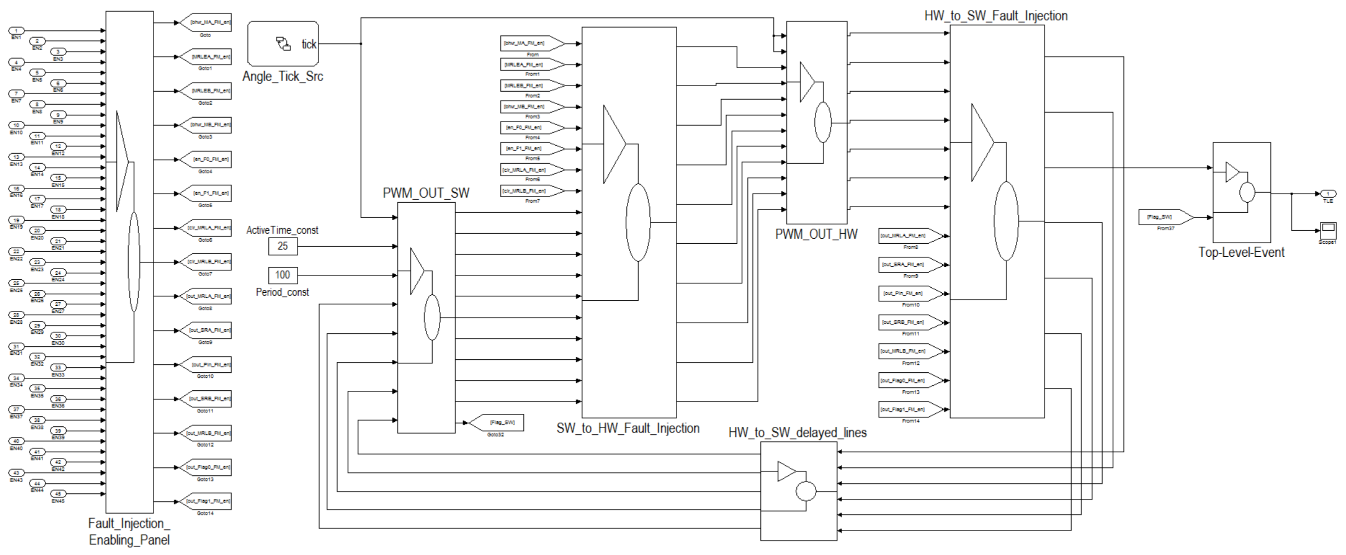


Fig. 1. Top-level view of the HW/SW composition.

mode functions allowing the designer focusing on the SoC architecture and fault-tolerance mechanisms design aspects. The minimal nature of injected faults intuitively ensures that only the failures significant to the analysis will be considered in the generated scenarios as will be described in detail in the following sections. Finally, our analysis process supports the MATLAB® Simulink/Stateflow [10] framework as modelling and analysis front-end that represents the standard de-facto for system design.

II. MOTIVATING EXAMPLE

As a motivating example we refer to a simple function generating a Pulse Width Modulated (PWM) waveform with period and active time specified by an external module. The example models the implementation of the function on the eTPU IP [11]. The eTPU is a programmable CPU co-processor designed for timing control (real-time input capture/analysis and output generation) provided with configurable hardware channels operating in parallel. A previous work [12] successfully applied formal verification to the eTPU based on System Verilog Assertions (SVA) language and Cadence^{TM}IFV tool for static formal verification. As shown in Fig. 1, at top level our model takes three main inputs: a reference source of ticks –representing the reference in the time, angle or similar domains– modelled as a counter register and a couple of values, representing the desired period and active time of the PWM waveform to generate. We can consider them as constants, with no loss of generality. The function is modelled as a composition of two main FSMs: the the hardware FSM and the software FSM. The hardware FSM (PWM_OUT_HW in Fig. 1) represents an eTPU channel configured in "either-match, non-blocking" mode [11]; it is composed of two "action units" operating in parallel: each action unit is provided with a comparator and three latches. Fig. 3 shows the finite state machine representation of one action unit in Simulink/Stateflow. The comparator seeks for a match in value between the ticks reference (TCRA) and a software-defined value (MatchA). When these two match, the comparator triggers the channel pin-action logic, which drives the output pin. The modelled actions on the output pin are: set pin-high (PinA=1 in Fig. 3) and set pin-low (PinA=0 in Fig. 3) [11]. The effective output pin behaviour is modelled by another FSM (not shown here) that keeps into account the events coming from both the action units. The three latches operate as follows: one latch notifies that a match event occurred, the service request latch notifies the eTPU scheduler that the channel has required service

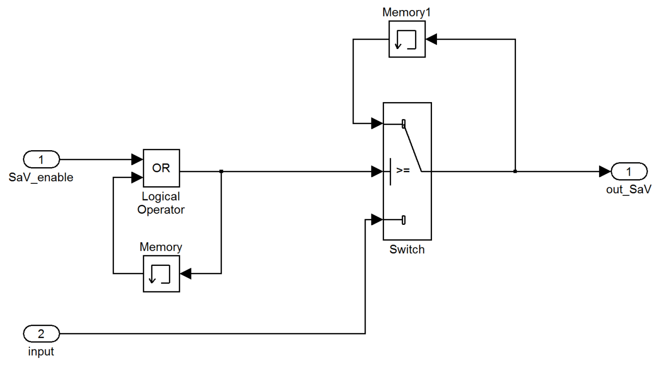


Fig. 2. Functional model of a stuck-at-value.

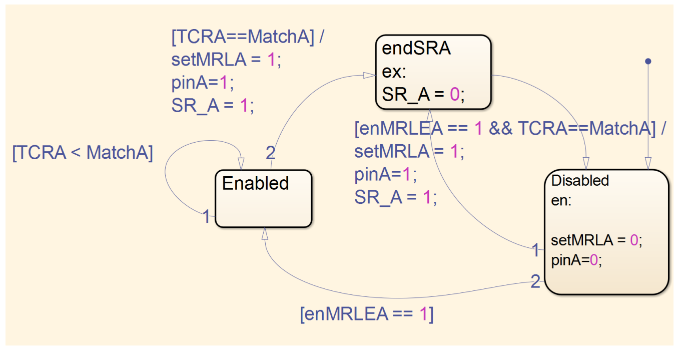


Fig. 3. Representation of one of the two action units in PWM_OUT_HW FSM as a Statechart in Simulink/Stateflow.

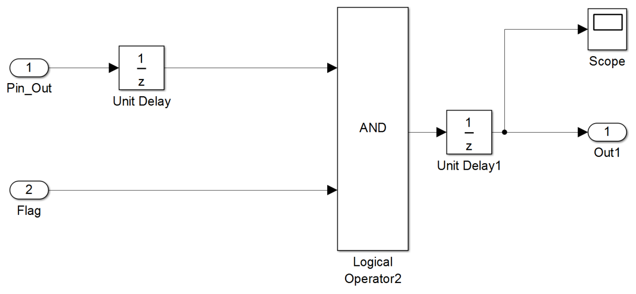
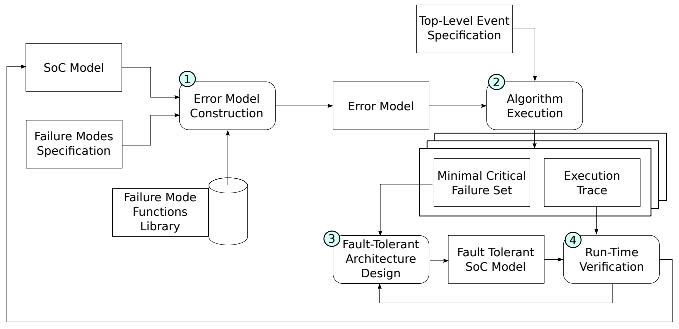


Fig. 4. Functional specification of the top-level event (TLE).



SoC Model
Failure Modes Specification
Failure Mode Functions Library
1
Error Model Construction
Error Model
Top-Level Event Specification
2
Algorithm Execution
Minimal Critical Failure Set
Execution Trace
3
Fault-Tolerant Architecture Design
Fault Tolerant SoC Model
4
Run-Time Verification

Fig. 5. Methodology.

and a Flag latch reflects a Boolean variable set by software; in our example the Flag latch is set by eTPU-software to reflect its state in hardware. At the match event both the service request latch and the action unit's latch are raised. With reference to Fig. 3, the variables used to represent these two latches are respectively SR_A and setMRLA. Depending on the current state of hardware latches a specific eTPU-software routine is selected by the eTPU scheduler and executed by the eTPU engine. Such a routine updates the proper match-register of the hardware channel with the next-match time value and (re)sets the related action unit latches. The eTPU-software FSM (PWM_OUT_SW in Fig. 1) consists of three states: (1) an initialization state that sets the initial hardware configuration, (2) a state handling the actions related to the end of the active time and (3) a state handling the end of the PWM period. For sake of simplicity, the interaction between hardware and software has been modelled as a finite positive delay, with the intention not to explicitly model the whole channels operating in parallel and the scheduler managing their concurrent requests. We use this example to analyse unwanted functional behaviours (top-level events) through our algorithm in order to automatically generate failure scenarios. Unwanted functional behaviours are typically identified through hazard analysis techniques, exploiting pre-defined sets of keywords. An example of possible unwanted top level events is: (1) wrong duration of the generated period and (2) wrong duration of the generated active time. We modelled functional faults as stuck-at-value occurring to hardware signals and variables (Fault injection blocks in Fig. 1). The functional specification for the stuck-at-value fault is depicted in Fig 2. Until the Boolean SaV_Enable signal equals zero, the Switch block lets the input value flow directly to the output (as an ideal wire). Once the SaV_Enable signal goes to one (no matter how many time it stays at one) the control of the Switch makes it isolate the input: the output is stuck at its last assumed value and cannot change any more (represented by Memory1 block in Fig. 2).

III. Analysis Methodology

We propose a model-based methodology for the automatic generation of failure scenario. The main steps are summarized in Fig. 5 and in this section we briefly review each of the activities. It is important to stress that the developed methodology and algorithm have also been applied to different application contexts, in particular as part of the effort in exploiting synergies between formal analyses and testing for the verification of complex embedded systems in the MBAT project [13] and as part of the effort in defining methodology and tools for the design of Systems-of-Systems in the DANSE project [14].

A. Error Model Construction

As a first step the Error Model of the SoC architecture is formalized (Activity 1 in Fig. 5) by modelling its functional architecture. For each component the nominal behaviour is described in terms of Simulink/Statechart blocks and the specification of the failure modes that can affect it is provided. This step should be guided by a previously performed hazard analysis or by the expertise of the designers. In the general case a failure mode is represented as a state machine (directly modelled by the designer or selected from a specific library) and formalizes how the nominal behaviour of the component is modified in presence of a failure. After an automatic model transformation step an error model is constructed. In Section V the formal aspects of the transformation step will be described in detail.

B. Top Level Event Specification

The top-level event (TLE) is a hazardous behaviour that must not occur. It is an invariant property that must hold when the model behaves correctly (as expected). The TLE can be either addressed by deductive safety analysis such as fault tree analysis (FTA), or it can be identified directly from system requirements specification. The top-level event is a function involving the outputs and possibly the inputs of the subsystem. The evaluation of the function modelling the TLE should always be FALSE except in case of property violation, for which it switches to TRUE and the TLE is said to be *activated*. Faults occurring across the system have the potential of activating the TLE. For example, the TLE modelled in Fig. 4 formalizes the following property: in absence of faults the value of Pin_Out (an output of PWM_OUT_HW in Fig. 1) delayed by one step and the value of Flag (an output of (PWM_OUT_SW in Fig. 1) must not be equal to one simultaneously.

C. Algorithm Execution

The Error Model is processed by the fault analysis (FA) algorithm that takes a top-level event specification as an additional input. The objective of the fault analysis is to exercise the error model by exhaustively exploring the inputs space and injecting failures in order to reach this unsafe behaviour. The outcome of this activity is two-fold: on the one hand a list of minimal sets of failures is produced (the minimal critical failure set). Each set collects the failures to be injected in order to assert the TLE and its minimal nature guarantees that if any of the collected (minimal set of) failures is not injected, the hazardous behaviour is not reached. On the other hand a set of execution traces is produced, each exposing a sequence of failure injections and input values that leads to the hazard for each minimal critical failure set. The traces can be used to effectively exercise the Error Model to show how the SoC evolves to a hazardous state and to help reasoning on its expected and unexpected unsafe behaviours.

D. Fault tolerant architecture design

The information collected in the previous activity is reviewed by the safety engineers to drive the design of one or more fault-tolerant versions of the SoC. It may be needed to iterate between activities 1, 2 and 3 (Fig. 5) to reach a mature description of the functional design before terminating the activity. The result of this activity is a fault-tolerant version of the SoC produced as an executable (and possibly synthesizable) model.

IV. Run-time verification

The traces generated during activity 3 (Fig. 5) are used to exercise the fault-tolerant model to check the effectiveness of the fault identification, detection and recovery (FDIR) mechanisms. During this phase flaws in the fault-tolerant SoC design may be discovered and iterations with both the safety engineering and the SoC design teams may be needed. The use of a model-based flow for the safety analysis of complex SoC has several advantages with respect to the classical manual process. At first the use of executable models helps the different teams to reason on a shared formal representation of the system, which helps minimizing the misinterpretation of requirements and communication issues between design- and safety-engineers. Then, models can be processed automatically using model-transformation engines easing the design exploration process and reducing the occurrence of design errors, costs and time-to-market. Finally, a formal model is important for supporting the certification process needed for the qualification of fault-tolerant SoCs [2], [3].

V. Formal Specs Verifier - Fault Analysis Tool

In this section the Formal Specs Verifier tool for Fault Analysis (FSV-FA) is described. At first an overview of the tool architecture will be provided and then the detailed description of the underlying exploration algorithm will follow.

A. Tool Overview

The FormalSpecs Verifier (FSV) is a framework for the verification of complex embedded systems. The core of the tool is based on a translator from a Simulink model to the NuSMV tool native language [8]. The transformation process produces a semantically equivalent NuSMV representation of the input model taking into account the non-determinism resolution that may be introduced during the transformation step. For the execution of the Fault Analysis the FSV tool has

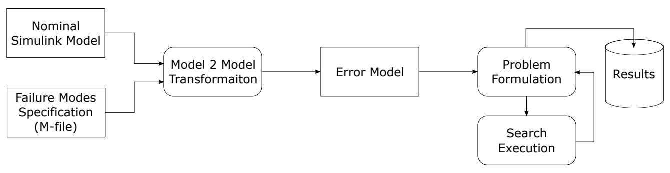


Fig. 6. Tool Overview.

been enriched with additional modules that implement part of the flow described in Section III. The enrichment covers in particular the automatic generation of the Error Model and the algorithm execution, summarized as follows: the tool translates a Simulink model to an internal format performing a model transformation step. The model is then processed by a second transformation that enriches each component with faulty behaviours taking into account the specification of the failure modes provided by the user obtaining a NuSMV model that can be processed by the internal formal engine.

B. Algorithm Description

The tool supports fixed-step discrete time Simulink models. A nominal model is formalized as a function $F[\mathbf{u}, \mathbf{x}, \mathbf{f}, k]$ that at each discrete time k maps inputs $\mathbf{u}[k]$ and current state $\mathbf{x}[\mathbf{k}]$ vectors to a vector of outputs $\mathbf{y}[\mathbf{k}]$ and next-state values $\mathbf{x}[\mathbf{k}+\mathbf{1}]$. The error model is produced by enriching the behaviour of the nominal model with the failure mode functions: for each component's failure mode an additional input variable, called the failure mode enable input, is created. Let $F = \{f_1, \ldots, f_N\}$ be defined as the set of all the failures added to the model: a failure enable input is a discrete-time Boolean function $f_j[k]$ that is TRUE if the failure f_j is active at the $k-th$ step, FALSE otherwise. The error model is then modelled as a function over the discrete time $\widetilde{F}[\mathbf{u}, \mathbf{x}, \mathbf{f}, k]$ where $\mathbf{f}$ is the vector of failure mode enable inputs. In case the latter is a constant vector of false values (identified as $\mathbf{0}$) we have that $\widetilde{F}[\mathbf{u}, \mathbf{x}, \mathbf{0}, k] = F[\mathbf{u}, \mathbf{x}, \mathbf{f}, k]$. In other terms, in case all failures are disabled, the nominal and error models are equivalent. Finally, the Top-Level Event is modelled as a Boolean function over discrete time $H[k]$ that is TRUE if the TLE occurs at time step k, FALSE otherwise. For the execution of the failure scenario generation algorithm additional variables called monitors are introduced. Each monitor is a Boolean function over the discrete time $m_j[k]$ such that $m_j[k]$ is TRUE if there exists a step $i \in [0, k]$ for which the failure enable input $f_j[i]$ has been TRUE, and it is FALSE otherwise. In other terms, each monitor is associated to a failure enable input that goes and remains to TRUE if the corresponding failure input has been TRUE in the past or in the current step. Our fault analysis algorithm (Algorithm 1) loops until a given discrete time limit bound provided by the user is reached (line 2). At step 3 the formal engine searches for a counter example that, at the given time step k, both minimizes the sum of the values of the monitors $m_1, \ldots, m_N$ and verifies the hazard predicate $H[k]$. The counter example is obtained iteratively by applying bounded model checking on the error model taking into account the dynamic behaviour of the SoC. Notice that the use of BMC in conjunction with a parallel SAT solver allows for successful application of the technique to real-life industry sized models. The minimization procedure ensures that only a minimal number of failures are injected to trigger

Algorithm 1 Fault Analysis Algorithm

```
1: Initialize bound: $k \leftarrow 0$
2: while $k < MAX\_BOUND$ do
3:     find a counter example s.t. $m_1[k] + m_2[k] + \ldots + m_N[k]$ is minimal and $H[k]$ is TRUE
4:     if counter example exists then
5:         define $\mathbf{m}^* = [m_1^*, m_2^*, \ldots, m_N^*]$ the found monitor configuration
6:         exclude $\mathbf{m}^*$ from the admissible solutions of the future searches
7:         extract input and failure enable values from the found counter example and store as a failure scenario
8:     else
9:         Increase time bound $k$
10:    end if
11: end while
```

the TLE. If a counter example is found, then the monitor values configuration is extracted and a new constraint is added to the model in order to exclude the found configuration from future searches (lines 5 and 6). The test case is extracted from the counter example storing inputs, outputs and internal variables of the system (lines 7 and 9). The loop is executed again to look for additional minimal possible monitor configurations until no more configurations can be found. The algorithm ensures that (1) every failure enable set produced is a minimal critical failure set for the TLE under analysis and (2) it is true by construction that there are no two traces exposing the same minimal critical failure set.

VI. Application to The Reference Example

The failure scenario generation methodology has been applied to the eTPU case study described in Section II using the FSV-FA toolset. A total number of 45 faults have been modelled in MATLAB Simulink using a predefined library. The simulation time was limited to $k = 28$ steps for a waveform with period $p = 100$ and active time $a = 25$. After 60 minutes of computation 5 failure scenarios have been found using a platform based on an Intel i7 2.4 GHz with 8 GB of RAM. One of the 5 property violations was observed at time step $k = 2$, while the other four occurred at time step $k = 28$. Fig. 7 reproduces one of the generated scenarios. The TLE has been modelled as the Simulink subsystem shown in Fig. 4; referring to the failure scenario shown in Fig. 7 such a TLE is activated when a stuck-at-one fault affects the output pin of the eTPU channel (Pin_Out). In order for the property to be violated, the fault is to be active from the beginning ($k = 0$). The lower sub-plot of Fig. 7 shows the value of the output pin (an output of PWM_OUT_HW in Fig. 1) and the value of the Flag signal (an output of PWM_OUT_SW in Fig. 1). The Flag signal is a local variable of the PWM_OUT_SW machine set to zero when the software is expected to be handling the hardware state "pin high" of PWM_OUT_HW, and set to one when the software is expected to be handling the state "pin low" of PWM_OUT_HW. At the init time ($k = 0$) the eTPU software sets the output pin value to zero (Flag=1), the "next match" to occur at time $k = 1$ and the action related to the match event to "set pin high". The pin is stuck at one all the time, in particular from the beginning of the simulation. The expected sequence of events is the following: at time $k = 1$, the pin should transition to 1 and at time $k = 2$ the value of Flag should transition to 0, representing a change in state of the eTPU software FSM (PWM_OUT_SW). Since the stuck-

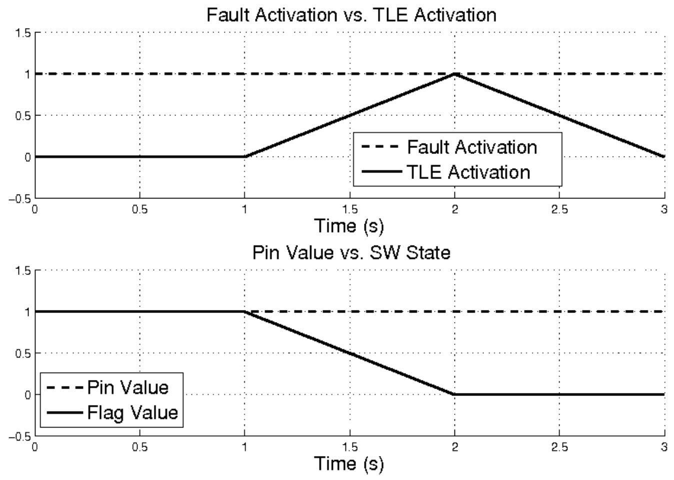


Fig. 7. Analysis Results.

at-one fault prevents the pin from changing, the property is violated and the TLE activtes.

VII. Conclusion

In this paper we presented a model-based methodology for the automatic generation of failure scenarios. The methodology has been applied to the analysis of a SoC architecture using a Simulink-based tool that implements a novel algorithm based on bounded model checking exploiting the recent advances of parallel SAT solving. A concrete SoC use case has been proposed to show the effectiveness of the described approach. In future work we will apply the presented methodology to more complex Systems-on-Chip models. In particular, we are interested in evaluating the effectiveness of cross-layer dependability mechanisms employed in safety-critical SoCs. Cross-layer dependability mechanisms allow distributing the fault/error detection phase and the diagnosis and handling phases across the whole system stack, thus reducing production costs. To this purpose hierarchical models of the various hardware and software abstractions (e.g. gate, circuit, hardware architecture, firmware, operating system, middle-ware and application) will be developed, exercised and analysed with the proposed methodology.

Acknowledgment

The research leading to these results was partially funded by the EU ARTEMIS Joint Undertaking under grant agreement n. 269335 MBAT and the Italian Ministry of Education, University and Research (MIUR) and by European Commission for funding the Large-scale integrating project (IP) proposal under the ICT Call 7 (FP7-ICT-2011-7) DANSE (Contract Number: INFSO-ICT-287716).

References

[1] R. C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE TRANSACTIONS ON DEVICE AND MATERIALS RELIABILITY*, 2005.

[2] IEC. *IEC 61508*, 04 2010.

[3] ISO. *Road vehicles – Functional safety*, 11 2011.

[4] T. Peikenkamp, A. Cavallo, L. Valacca, E. Bode, M. Pretzer, and E.M. Hahn. Towards a unified model-based safety assessment. *SAFECOMP*, 2006.

[5] M. Gdemann, F. Ortmeier, and W. Reif. Using deductive cause-consequence analysis (DCCA) with scade. *SAFECOMP*, 2007.

[6] A. Joshi and Mats P.E. Heimdahl. Model-based safety analysis of simulink models using scade design verifier. *SAFECOMP*, 2005.

[7] M. Bozzano, A. Cimatti, J-P. Katoen, V.Y. Nguyen, T. Noll, and M. Roveri. Safety, dependability and performance analysis of extended aadl models. *Comput. J.*, 2011.

[8] O. Ferrante, L. Benvenuti, L. Mangeruca, C. Sofronis, and A. Ferrari. Parallel nusmv: a nusmv extension for the verification of complex embedded systems. *Proc. of the International Conference on Computer Safety, Reliability, and Security*, 2012.

[9] A. Ferrari, L. Mangeruca, O. Ferrante, and A. Mignogna. DesyreML: a sysml profile for heterogeneous embedded systems. *ERTS, Embedded Real Time Software and Systems*, 2012.

[10] Mathworks. Matlab. http://www.mathworks.it/products/matlab/.

[11] Freescale. *ETPURM, Enhanced Time Processing Unit (eTPU) Reference Manual*, 05 2004.

[12] C. Rodrigues. A case study for formal verification of a timing co-processor. *IEEE, LATW*, 2009.

[13] MBAT. https://www.mbat-artemis.eu/home/.

[14] DANSE. https://www.danse-ip.eu/home/.