Reliable Execution of Statechart-Generated Correct Embedded Software under Soft Errors

Ronaldo R. Ferreira[1], Thomas Klotz[2], Thilo Vörtler[2], Jean da Rolt[1], Gabriel L. Nazar[1]
Álvaro F. Moreira[1], Luigi Carro[1], Karsten Einwich[2]

[1]*Instituto de Informática – Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil*
{rrferreira, jrjoaquim, glnazar, afmoreira, carro}@inf.ufrgs.br

[2]*Fraunhofer Institute for Integrated Circuits, Dresden, Germany*
{thomas.klotz, thilo.voertler, karsten.einwich}@eas.iis.fraunhofer.de

***Abstract*—This paper proposes a design methodology for fault-tolerant embedded systems development that starts from software specification and goes down to hardware execution. The proposed design methodology uses formally verified and correct-by-construction software created from high-level UML statechart models for software specification and implementation. On the hardware reliability side, this paper uses the MoMa architecture for reliable embedded computing which we deploy as a soft-core onto an off-the-shelf FPGA. MoMa introduces architectural innovations that support the semantics of the UML statechart execution in a reliable fashion. The proposed design methodology is evaluated with a real automotive case study based on an exhaustive FPGA-implemented fault injection campaign.**

***Keywords*-Model-driven engineering; fault-tolerance; reliability, soft-errors; statecharts**

I. Introduction

Reliability has become a major design constraint in current hardware designs [1]. Since the sizes of transistors get close to a technical minimum, the logical state of a transistor can be disrupted by small electric interferences. Soft-errors, which were restricted to avionics and space systems a few transistor generations ago due to radiation induced interference, will increase reliability concerns of new system designs [2]. Radiation induced soft errors are caused by highly energized particles that strike a circuit disrupting its logical state [3]. With small transistor sizes, even particles attenuated by the atmosphere may hit circuits in operation at sea level such as the ones used in cars, not to mention highly exposed circuits used in aircrafts. Soon, automotive systems will have to handle radiation induced electromagnetic interference disrupting transistor states [4]. The solution adopted in space systems is the use of radiation hardened microprocessors, with unitary prices of $200,000 for a 25 MHz processor [5]. Hardened circuits are not feasible in domains such as automotive and aviation, which require the use of low-price off-the-shelf solutions.

'Checkpointing' is used to create a consistent architectural state where the architecture can rollback in case an error is detected. The problem with checkpointing is that its efficiency is severely reduced depending on how many instructions are allowed to execute before the architectural state is stored [6]. In the best case scenario, the total overhead to recover the architecture to a consistent state is at least 25% of the total execution time. In critical embedded systems, which usually have some sort of real-time behavior, this high overhead jeopardizes its functional behavior. In case of ASIC designs where lock-step execution is feasible, a simpler form of checkpointing is the re-dispatch of the erroneous instruction followed by a pipeline flush of the in-flight instructions, which is known as 'instruction replay'. However, this scheme works in ASIC because of the duration of the transient pulse, which usually disappears after one execution cycle. In case of long transient pulses and when the chip is deployed onto FPGAs, the failure model changes drastically, making instruction replay not feasible.

For safety critical systems, hardware reliability has to go hand in hand with software correction, where software malfunctions due to implementation errors have to be avoided at a reasonable cost. Software correction is also a mandatory requirement for system certification by governmental authorities [7]. In an ideal scenario, software specification has to start at a high level of abstraction based on formal models from which verified and correct code is automatically generated. The life cycle for critical systems requires tool support with high scalability backed up by a well-defined design methodology. That way, the traceability of high level software models down to generated correct source code and its deployment onto the reliable hardware becomes feasible.

This paper proposes a model-driven system design methodology where correct-by-construction software is automatically generated from formally verified UML statechart models and later deployed onto the *Matrix Operation Microprocessor Architecture* (MoMa) reliable hardware architecture. The execution semantics based on basic block transactions guarantees that a state either finishes correctly or it is re-executed without corrupting the architectural state. This design methodology is supported by the COSIDE® [8] design environment for system specification, making it suitable for industrial application. The proposed hardware/software design methodology is thoroughly evaluated using a real automotive application designed from software specification down to hardware execution in MoMa.

This case study is evaluated for reliability with fault injection experiments using an off-the-shelf Xilinx® FPGA.

This paper is organized as follows: Section II presents related work. Section III introduces the design methodology and explains how UML statecharts are executed in a reliable fashion by MoMa. Section IV presents the case study and discusses the experimental evaluation of the approach. Finally, Section V concludes the paper.

II. Related Work

Brisolara et al. [9] propose the synthesis of heterogeneous systems from UML models, which are further transformed into low-level Simulink models. From Simulink, hardware task partitioning and design space exploration is performed. Leveraging the synthesis of hardware/software systems, Mischkalla et al. [10] utilize UML and SysML models for hardware and software modeling, and functional verification by transforming the UML models into equivalent SystemC ones. From the generated SystemC models, the hardware platform is synthesized onto an FPGA. UML models were considered in [11] as a model to describe hardware platforms with fine-grained control of how to meet real-time requirements specified in this UML model. Although UML has been studied for many tasks of hardware/software system specification and synthesis, radiation induced errors and their impact on reliability have not been considered by design methodologies yet.

The challenge of generating reliable hardware is that several software and hardware aspects influence reliability in a non-linear way. Parizi et al. [12] show that even the compiler has an influence on the reliability. The focus on the hardware side in the reliability for industrial practice is usually based on reliable processors. Most efforts of the European community are centered on the development of the LEON-FT platform [13], with current research being led by European Space Agency [14]. The problem with most of the radiation tolerant microprocessors, besides their performance gap and high unit prices, is that they triplicate most of the hardware elements, jeopardizing power consumption [2].

Differently from previous work in the literature, we use UML for software specification, formal verification and code generation to a reliable hardware platform that was designed to guarantee the correct execution of the generated software. Correct execution is attained with a transactional data-path, which ensures that the computation performed in a state of the UML statechart is only persisted if it is correct. The transitions of the UML statechart are ensured to be correct by the branch-free state machine unit. This paper puts the existing expertise and best-practices for UML modeling and software verification together with a novel reliable hardware substrate ready to be deployed onto an FPGA.

III. Design Methodology for Reliable Systems

This section introduces the proposed model-driven methodology for reliable system design based on the fault-tolerant execution of statecharts. Section III-A introduces the workflow of the proposed design methodology. Section III-B presents the

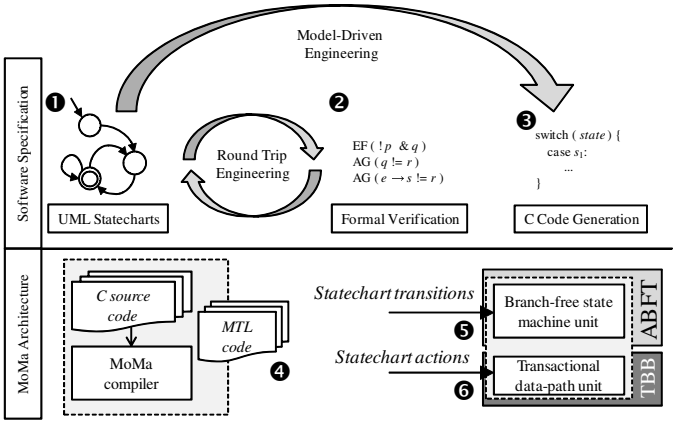


Fig. 1. Model-driven design methodology workflow. Upper swim lane encloses SW specification and the bottom one the MoMa core.

UML statecharts and how they can be used as an executable artifact. The execution of UML statecharts is composed of two parts, with each one being connected to a dedicated MoMa architectural unit: i) the data-flow execution that takes place inside a state; and ii) the control-flow resolution, which computes the next state transition function based on existing Boolean guard conditions. Section III-C presents how the code of a state is executed and protected against errors that compromise the behavior of instructions. Section III-D discusses how the state transition function is resolved in order to decide the next state to be executed.

A. Methodology Overview and Design Artifacts

Fig. 1 depicts the proposed model-driven design methodology split up into two swim lanes. The upper swim lane encloses the tasks and artifacts of software specification, while the bottom one shows how the generated C code is deployed onto the MoMa architecture. In the following, we refer to each artifact by its number shown inside the black circle in Fig. 1.

Software specification starts with the creation of UML statecharts model (#1). When the designer is satisfied with a design, it is formally verified using model checking for correctness (#2). For this, a formal specification in terms of properties is mandatory, which is derived from the requirements. The designer performs round-trip engineering until the modeled UML statecharts are asserted as correct. From the verified statechart, a correct-by-construction C program (#3) is generated.

This correct C code is compiled with the MoMa compiler, producing its corresponding machine code in so-called *Matrix Transfer Language* (MTL) (#4). The state machine in MoMa works by executing the control-flow graph of the program as a matrix multiplication which is protected with *Algorithm-Based Fault Tolerance* (ABFT) [15]. The MoMa compiler generates the matrix structure for the state machine of the program to be executed by the branch-free state machine execution unit (#5), which implements ABFT in MoMa. The data-path is protected by the architectural innovation *transactional data-path*, which decodes and executes the basic blocks of the

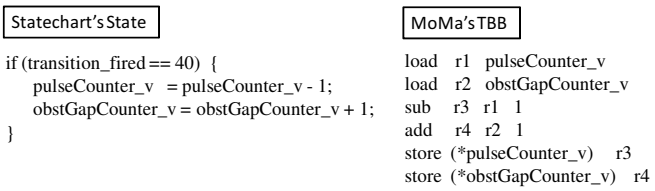


Fig. 2. A state in C with its respective TBB.

program as atomic *transactions* (#6).

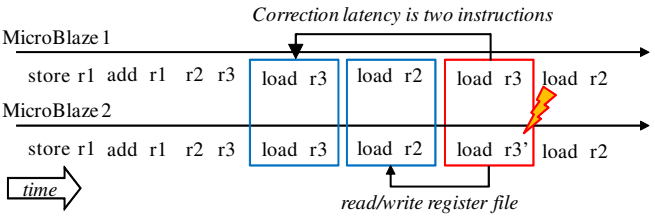
B. Modeling and Verification Using UML Statecharts

The *Unified Modeling Language* (UML) [16] is the *de-facto* standard for system modeling at distinct abstraction levels. UML state diagrams, often also called statecharts, describe the dynamic behavior of a system in terms of inputs, an internal state, outputs, and the transition between states.

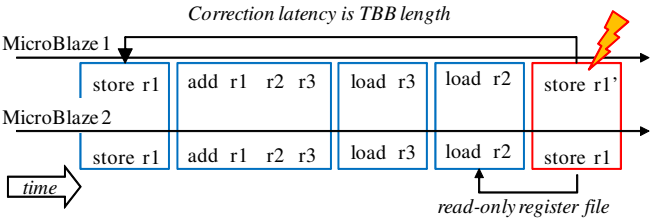
The formal technique *model checking* [17] allows to automatically prove whether a set of properties representing the specification holds on a given system model or not. For specifying this model some type of finite state representation is considered, and the system requirements need to be formalized in terms of temporal logic properties. If a property is violated by the model, a counterexample in terms of a simulation trace is provided. Applying model checking to formally verify statecharts has the major advantage that design errors are found early in the design process. However, UML statecharts according to [16] are not suitable for modeling embedded software, because the UML does not provide a formal execution semantics, and, hence, it does not exactly define how events are handled. UML statecharts may also contain non-determinism if several transitions can be fired at the same point in time. Moreover, the application of model checking requires additional constraints to be feasible in practice, e.g., restricting the supported data types.

A subset of UML statecharts has been derived to overcome these ambiguities, which is sketched next. This subset contains simple and composite states, as well as initial, junction, and shallow history states. Transitions are not just annotated with trigger, guard, and activity, but also with a user-defined *priority* in terms of a natural number to exclude non-determinism. The smaller this number, the higher the priority of the corresponding transition. The given input and internal variables may trigger events to fire transitions; only the event firing the transition with the highest priority is taken into account, and all others are rejected, i.e., there is no event queue. For a detailed presentation of this subset readers should refer to [18].

Considering the defined subset of statecharts, formal verification can be applied to verify their correctness. The verified statechart is then automatically transformed into an semantically equivalent C code to be deployed in simulation or to be executed on an embedded system. The next sections discuss how the MoMa architecture attains the reliable execution of the C code generated from the statechart.



(a) Error detected in a load instruction. Register file is in read/write mode.



(b) Error detected in a store instruction. Register file is in read-only mode.

Fig. 3. TBB error scenarios.

C. Reliable Execution of the Actions of the UML Statechart

The architectural unit of MoMa that executes the instructions of the actions of the statechart is the *transactional datapath* (TDP). The TDP contains two modified MicroBlaze® cores in lockstep, i.e., each core executes exactly the same instruction as the other one during the complete microprocessor life cycle. The control-flow machinery of these cores has been removed and further relational instructions have been added.

The software construct that executes in the TDP is the *transactional basic block* (TBB). A TBB is a basic block, i.e., a sequence of assembly instructions that ends with some non-branching terminator instruction. In addition, if a TBB produces a value that will be used by another block, this TBB must guarantee that this value is written to main memory before the TBB ends. Therefore, a TBB is a block that does not share any register with another TBB.

In a TBB, a sequence of store instructions is the block terminator. No load instructions are allowed within this set of store instructions, i.e., all load instructions are placed before the stores. To illustrate the TBB construction, Fig. 2 shows a state written in C with its respective TBB. Notice that the TBB loads the program variables *pulseCounter_v* and *obstGapCounter_v* without assuming that these values were active in the register file.

The TDP guarantees the correct execution of the TBB through its instruction checking mechanism that compares for all instructions of the TBB whether the two executions of each MicroBlaze® core of the instruction operands and the destination registers are the same. If this comparison fails, an error is detected and the same TBB is re-executed from the start. Recalling the definition of the TBB, all memory loads precede the stores. If the TBB reaches a store instruction, it is guaranteed that all the preceding instructions are correct, and thus the register file as well. When an error is detected in any store instruction, the TBB is re-executed without permission

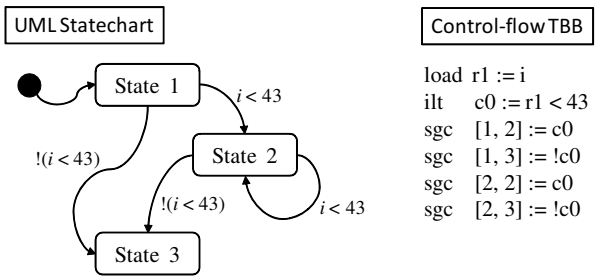


Fig. 4. A UML statechart with its control-flow TBB.

to modify the register file. Fig. 3 illustrates these two error scenarios. This figure also shows the error correction latency depending on the point of the TBB exection where the error was detected.

Fig. 3a shows a scenario where the second TBB instruction contains an error. In this case, only two instructions are re-executed, thus the correction latency is small. On the other hand, when the error is detected in the last TBB instruction, as shown in Fig. 3b, the latency is the TBB length. Therefore, the upper limit on the correction latency is the TBB length.

Concluding, when the code implementing a state of the statechart is mapped to a TBB it will be executed by TDP. The TDP guarantees that if the TBB and thus the state of the statechart has finished to execute, its execution is accredited as correct. The remaining statechart construct that needs to be protected by MoMa for its reliable execution is the computation of the next state, which is presented in the next section.

D. Reliable Next State Transition Resolution

The control-flow machinery implemented in MoMa, called *branch-free state machine* (BFSM), is based on *aggressive branch predication*, i.e., on the transformation of all control-flow dependencies into data-flow ones [19]. In statecharts terminology, a sequence of triggers and guard conditions, i.e., the expressions that must hold in order to the statechart branch from one state to another, are transformed into a set of TBBs. The difference is that a TBB implementing the control-flow in MoMa does not modify the register file and the memory, because triggers and guard conditions do not contain store instructions.

Fig. 4 shows an illustrative example of a statechart with its respective control-flow TBB. This TBB makes use of two instructions introduced in MoMa. The *ilt* (is less than) compares two operands and computes their less-than relation, and *sgc* (set guard condition) sets the truth value of the guard condition for a given transition of the statechart. In this example, we have the Boolean expression $i < 43$ guarding the transitions from state 1 to 2 and the self-transition of state 2. The negation of this Boolean expression guards the transitions from state 1 to 3, and from state 2 to 3.

To protect the guard conditions of the control-flow TBBs, MoMa stores the evaluated guard conditions in a *state transition matrix* M. The values of this matrix are set using the *sgc* instruction as shown in Fig. 4. Using the example of the figure, the positions where the guard condition *c0* holds will be set to 1, thus $M[1,2] = M[2,2] = 1$. In the cases where *c0* does not hold, the respective M positions will be set to 0, thus $M[1,3] = M[2,3] = 0$.

MoMa encodes each state of the statechart with a *state vector*. Assuming that the statechart has n states and that these states are labeled with an integer $1 \leq k \leq n$, a state k is encoded by a vector s_k, where $s_k[k] = 1$ and 0 for its remaining $n-1$ positions. Using again the statechart depicted in Fig. 4, we have the following three state vectors: $s_1 = [1\ 0\ 0]$, $s_2 = [0\ 1\ 0]$, and $s_3 = [0\ 0\ 1]$.

Let us assume that the active state of the statechart is encoded with the state vector s_k. With the state transition matrix and the state vectors, the next state s_{k+1} can be computed as

$$s_{k+1} := s_k \times M \tag{1}$$

The BFSM unit implements Eq. 1, which is a matrix multiplication, protecting it with the technique Algorithm-Based Fault Tolerance (ABFT) [15]. ABFT computes very efficiently the row and column checksum of matrices produced from matrix operations. If the checksum before the matrix operation differs from the one computed after it completes, ABFT corrects the erroneous value using residue arithmetic. ABFT guarantees that all single errors are corrected. The product described in Eq. 1 is in fact a bit-matrix multiplication. The BFSM bit-matrix multiplication implementation only uses combinational logic ('or' and 'and' gates), eliminating the need for memory elements to store multiplication elements.

IV. Case Study: Automotive Window Lifter

This section evaluates the proposed design methodology using a real-world example from the automotive domain, namely the control of a window lifter. Section IV-A describes the functionality and the modeling of the control using statecharts, and Section IV-B shows how the correctness of the statechart is verified using model checking. Finally, Section IV-C presents the error coverage evaluation of the window lifter executing in MoMa based on exhaustive fault injection.

A. Description

The window lifter controls the window with two buttons located in the door, controlling the motor that puts the window in motion. A HAL sensor is utilized to detect and signal the motion of the window.

Fig. 5 shows the statechart modeling the window lifter control. Input, output, and internal variables are identified with the suffixes '*_i*', '*_o*', and '*_v*', respectively. First, the internal and output variables are initialized and the statechart is set to the state *STOP*. If the down key is pressed (*downKey_i*), the statechart transitions to *MANDOWN*, where the motor is enabled (*motor_en_o*) to move the window down (*motor_dir_o=true*). There, the variable *hallimpulse_i* is an input from the hall sensor signalizing the movement of the window; the window position is tracked by *pulseCounter_v*. If either the window is completely open (*pulseCounter_v==0*)

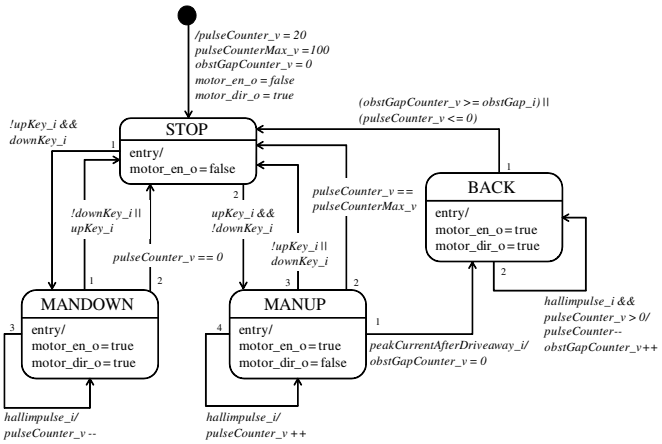


Fig. 5. UML statechart modeling the window lifter control.

or the up key is pressed (*upKey_i*), the state *STOP* is activated again.

The same procedure has been modeled for closing the window (*MANUP*), except that *pulseCounter_v* may reach the closed position (*pulseCounterMax_v*). To avoid accidents, an additional clamping protection has been implemented. The presence of an obstacle is detected in terms of a peak in the current driving the motor (*peakCurrentAfterDriveaway_i*). If the latter occurs while closing the window, the statechart branches to the state *BACK*. In this state, the window is opened again until either the window reaches the predefined position *obstGap_i* or the window is completely open (cf. *MANDOWN*); *obstGapCounter_v* tracks how much the window has been lowered.

Note: The statechart in Fig. 5 is a simplified model for explaining our methodology. A more detailed version of the statechart, providing additional functionality such as an automatic calibration of the window lifter, can also be handled by our design methodology.

B. Verification

For the modeling and the formal verification of the statechart, the proposed methodology adopts the state-of-the-practice design environment COSIDE®, which automatically generates a finite state model in terms of SMV code for the symbolic model checker NuSMV [20] from the given statechart. Besides the statechart, a formal specification asserting the expected behavior using properties expressed in temporal logics is needed as input (in this paper we use CTL). The verification is executed in background and, found counter examples are visualized in the statechart. Next, a number of properties that were proven correct are discussed[1].

In the window lifter control all states of the statechart have to be reachable at any time. Formalized in CTL this requirement can be expressed, e.g., for the state *STOP*, as:

$$\mathbf{AG\ EF}\ (state = STOP) \tag{2}$$

[1]To successfully verify the control, also a simplified model of its environment is necessary: In our example, enabling *motor_en_o* results in a corresponding *hallimpulse_i* modeling the movement of the motor.

stating that always (**AG**) there exists a path such that eventually (**EF**) *STOP* will be active.

Moreover, it is required to ensure that the motor may only close the window in the state *MANUP*, formalized as

$$\mathbf{AG}\ ((\neg motor\_dir\_o\ \wedge\ motor\_en\_o) \rightarrow (state = MANUP)) \tag{3}$$

The correct implementation of the clamping protection is crucial to avoid serious accidents. The property

$$\begin{aligned}\mathbf{AG}(\ &(\neg motor\_dir\_o\ \wedge\ motor\_en\_o\ \wedge\\ &peakCurrentAfterDriveaway\_i)\ \rightarrow\\ &\mathbf{AX}\ \mathbf{A}[\ (motor\_dir\_o\ \wedge\ motor\_en\_o)\\ &\mathbf{U}\ (pulseCounter\_v \leq 0\ \vee\\ &obstGapCounter\_v \geq obstGap\_i)\ ]\ )\end{aligned} \tag{4}$$

specifies that always if the motor closes the window ($\neg motor\_dir\_o\ \wedge\ motor\_en\_o$) and an obstacle is detected ($peakCurrentAfterDriveaway\_i$), then in the next time step (**AX**) the window is opened again ($motor\_dir\_o\ \wedge\ motor\_en\_o$), until (**U**) the window is completely open ($pulseCounter\_v\ \leq\ 0$) or it reaches the predefined open position ($obstGapCounter\_v \geq obstGap\_i$).

The formal verification of the statechart describing ensures that it meets its specification. From the verified statechart, COSIDE® automatically generates correct-by-construction C code which runs on an embedded control unit controlling the mechanics and the motor of the window lifter system using digital output ports also described in the statechart.

C. Error Coverage Evaluation

The fault injection was performed using the MoMa VHDL code deployed on a Xilinx® Virtex-5 FPGA board similar as in [21]. The fault model we assume is the single error, where only one bit is changed when the fault is injected. This fault model accounts for *Single Event Transients* (SET) and *Single Event Upsets* (SEU) errors. The fault injection campaign was exhaustive, i.e., at each program cycle we have injected one fault on each signal of the entire netlist.

The simulation of SET errors was done with a 'saboteur' module, which after one signal of the circuit under test has been randomly chosen, flips one of the bits of the signal that injects the error in the circuit logic. The SEU simulation is similar, but the bit-flip is inserted in flip-flops, i.e., in memory elements. The fault injection campaign is composed of 254,499 injected faults with a random walk execution on the window lifter statechart presented in Fig. 5.

Table I presents the fault injection results for each MoMa architectural unit divided by SETs, i.e., errors in logic, and SEUs, i.e., errors in memory elements. The error detection coverage found was 100%, with a measured error correction coverage of the entire MoMa executing the statechart is 99.84203%.

The error correction coverage obtained for the BFSM unit can be increase with a small cost in area. The BFSM has a priority encoder, which accounts for approximately 10% of the

TABLE I
EXHAUSTIVE FAULT INJECTION RESULTS WITH ABSOLUTE NUMBER OF INJECTED FAULTS, SET+SEU TOTAL ERROR COVERAGE, AND RATE OF INJECTED FAULTS NOT LEADING TO AN ERROR (MASKED).

MoMa Unit	SET Correction	SET Detection	SEU Correction	SEU Detection	Total Coverage	No. of Faults	Masked Faults
TDP	100%	100%	100%	100%	100%	190,511	48.04814%
Rollback	100%	100%	100%	100%	100%	31,832	7.38251%
BFSM	92.68293%	100%	93.64243%	100%	100%	32,156	89.99876%
TOTAL ERROR CORRECTION COVERAGE				**99.84203%**			

BFSM area. The priority encoder is responsible for checking whether the ABFT algorithm implementation of the BFSM unit works correctly. If the priority encoder is affected by an error, it may compute a false positive, or even corrupt the computed address of the next TBB to be executed. However, the entire BFSM unit accounts for only 8% of the area MoMa needs on the FPGA. Therefore, the triplication of the priority encoder is feasible, so that the coverage of uncorrected errors can be increased.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we have introduced a model-based methodology for the design of reliable embedded systems that uses high-level UML statecharts as modeling formalism. These statecharts are formally verified using a model checker coupled with the state-of-the-practice COSIDE® design environment. After the statecharts are asserted as correct, correct-by-construction C code is deployed into the fault-tolerant MoMa architecture. The MoMa architecture provides architectural units that guarantee the reliable execution of the code generated from the statecharts by executing both the state actions and the resolution of control-flow transitions in an atomic fashion. This work has presented how to put together software correctness and hardware reliability supported by an industrial case study from the automotive domain.

As future work, we will address the use of formal properties and verification results from the model checker embedded in COSIDE to create *fault injection assertions*. These assertions will allow the system designer to create dependability cases based on the reuse of artifacts that have already been created for software verification. For very critical applications the system designer has to certify that his solution is dependable [7], and the formal way a system designer shows that the system is correct is writing safety cases. This idea could also be used to plug together unit tests, e.g., JUnit, and the reliability evaluation. At the end, using these verication artifacts the safety cases for reliability could be generated automatically as [22] does for software correction.

ACKNOWLEDGMENT

This work is supported by CNPq and FAPERGS, Brazil. R. Ferreira was also supported by DAAD, Germany, during his stay at the Fraunhofer Institute. The research leading to these results has partially received funding from the ARTEMIS Joint Undertaking under grant agreement Nr. 295311 and from the German Federal Ministry of Education and Research (BMBF).

REFERENCES

[1] R. Baumann, "Soft errors in advanced computer systems," *IEEE Des. Test Comput.*, vol. 22, no. 3, pp. 258–266, 2005.
[2] S. Hamdioui, D. Gizopoulos, G. Guido, M. Nicolaidis, A. Grasset, and P. Bonnot, "Reliability challenges of real-time systems in forthcoming technology nodes," in *DATE*. IEEE, 2013, pp. 129–134.
[3] E. Normand, "Single event upset at ground level," *IEEE Trans. Nucl. Sci.*, vol. 43, no. 6, pp. 2742–2750, 1996.
[4] T. Konefal, A. Marvin, J. Dawson, and M. Robinson, "A statistical model to estimate an upper bound on the probability of failure of a system installed on an irradiated vehicle," *IEEE Trans. Electromag. Compat.*, vol. 49, no. 4, pp. 840–848, 2007.
[5] J. Penix and P. C. Mehlitz, "Expecting the unexpected: Radiation hardened software," NASA Ames Research Center, Tech. Rep., 2005.
[6] H. Chen and C. Yang, "Fault detection and recovery efficiency co-optimization through compile-time analysis and runtime adaptation," in *CASES '13: int. conf. on Compilers, Arch., and Synthesis for Embedded Systems*. ACM, 2013, p. 10.
[7] Y. Moy, E. Ledinot, H. Delseny, V. Wiels, and B. Monate, "Testing or formal verification: DO-178C alternatives and industrial experience," *IEEE Softw.*, vol. 30, no. 3, pp. 50–57, 2013.
[8] "COSIDE web page," http://www.coside.de, 2013.
[9] L. Brisolara, M. Oliveira, R. Redin, L. Lamb, and F. Wagner, "Using UML as front-end for heterogeneous software code generation strategies," in *DATE*. IEEE, 2008, pp. 504–509.
[10] F. Mischkalla, D. He, and W. Mueller, "Closing the gap between UML-based modeling, simulation and synthesis of combined HW/SW systems," in *DATE*. IEEE, 2010, pp. 1201–1206.
[11] I. Gray, N. Matragkas, N. Audsley, L. Indrusiak, D. Kolovos, and R. Paige, "Model-based hardware generation and programming - the MADES approach," in *ISORC*. IEEE, 2011, pp. 88–96.
[12] R. B. Parizi, R. R. Ferreira, L. Carro, and A. F. Moreira, "Compiler optimizations do impact the reliability of control-flow radiation hardened embedded software," in *IESS*. Springer, 2013, pp. 49–60.
[13] J. Gaisler, "A portable and fault-tolerant microprocessor based on the SPARC v8 architecture," in *DSN*. IEEE, 2002, pp. 409–415.
[14] A. Pouponnot, "Strategic use of SEE mitigation techniques for the development of the ESA microprocessors: past, present, and future," in *IOLTS*. IEEE, 2005, pp. 319–323.
[15] K.-H. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comp.*, vol. 33, no. 6, pp. 518–528, 1984.
[16] Object Management Group, "Unified Modeling Language Specification version 2.4.1," 2011.
[17] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. The MIT Press, 2000.
[18] T. Klotz, E. Fordran, B. Straube, and J. Haufe, "Formal verification of UML-modeled machine controls," in *ETFA*. IEEE, 2009, pp. 1–7.
[19] J. R. e. a. Allen, "Conversion of control dependence to data dependence," in *POPL*. ACM, 1983, pp. 177–189.
[20] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV2: An opensource tool for symbolic model checking," in *CAV*. IEEE, 2002, pp. 359–364.
[21] M. Aguirre, V. Baena, J. Tombs, and M. Violante, "A new approach to estimate the effect of single event transients in complex circuits," *IEEE Trans. Nucl. Sci.*, vol. 54, no. 4, pp. 1018–1024, 2007.
[22] N. Basir, E. Denney, and B. Fischer, "Deriving safety cases for hierarchical structure in model-based development," in *SAFECOMP*. Springer, 2010, pp. 68–81.