Formal Verification of AUTOSAR FlexRay State Manager

Ghada Bahig
Mentor Graphics
Cairo, Egypt

Amr El-Kadi
American University in Cairo
Cairo, Egypt

Ashraf Salem
Mentor Graphics
Cairo, Egypt

***Abstract*— Automotive software systems have continuously faced challenges in managing complexity associated with functional growth, flexibility of systems so that they can be easily modified, scalability of solutions across several product lines, quality and reliability of systems, and finally the ability to detect errors early in design phases. AUTOSAR was established to develop open standards to address these challenges. Formal method is one way to address the ability to detect errors and ensure compliance to requirements in early design stages. In this paper, AUTOSAR's FlexRay State Manager basic software module is formally represented in finite state machine augmented with complex data types. Specification requirements are mapped into formal model theorems and assertions. SMT solvers are utilized to validate design compliance to specification to show the possibility of detecting errors early in the design phase via mapping AUTOSAR's specification into formal design notation.**

I. INTRODUCTION

Failure of safety critical software could cause hazardous consequences on human life. One major factor that contributes to unsafe systems is incompliance to specification. It is crucial in automotive systems to ensure design correctness from compliance to specification perspective as early as possible. Safety standards put strict processes that involve manual reviews and requirements traceability in all software life cycle to ensure specification compliance. Industry still heavily relies on manual reviews and processes which is impractical since specification is still captured in informal and semi-formal notations which opens the door for requirement specification ambiguity. Attention to safety software engineering started when failures in embedded critical systems resulted in hazardous consequences. A good number of such failures are attributed to incompliance to specification,

Existing approaches that target Automotive software/system safety via ensuring specification compliance include manual dependency on standards and processes, such as, ISO-26262 in automotive domain, enforced by regulatory committees to ensure software safety, extensive testing at different levels including white box testing, black box testing, system and integration testing based on a variety of algorithms, such as, random test generation, path oriented, goal oriented, expert based adhoc test designs, model driven approaches which depend on modeling an abstraction of the system and simulating these abstractions manually based on designed test cases and finally formal methods but on a very small scale [1][2].

In this paper, we described AUTOSAR's automotive basic software module, FlexRay State Manager, in SAL [3] formal notations. AUTOSAR's specification requirements are directly mapped to theorems to assert counterexamples that exist that could violate these theorems. We verified the formally represented system by SMT solver to ensure the system upholds specification requirements mapped to theorems. Any violation is asserted by SMT solver as a counterexample allowing the system designer to address such compliance violations. This case study allows detection of early design errors, including ambiguous requirements in the specification since generated counterexamples from theorem requirements identified by SMT solver are analyzed to identify if they are actual violations against the specification or incorrect mapping in the design.

The paper is organized as follows: section two gives brief overviews on SAL formal transition language [3] and checkers, on road vehicles – Functional Safety ISO 26262 standard and its safety mandates in relation to our proposed implementation and finally on AUTOSAR (AUTomotive Open System ARchitecture) and FlexRay State Manager software module in the AUTOSAR modules stack. Section 3 shows a case study of the framework on AUTOSAR's FlexRay State Manager.

II. BACKGROUND

A. AUTOSAR

AUTOSAR is a worldwide development cooperation of car manufacturers, suppliers and other companies from the electronics, semiconductor and software industry. Since 2003 they have been working on the development and introduction

978-1-4799-8200-4/14/$31.00 ©2014 IEEE

of open, standardized software architecture for the automotive industry [7][8][9].

AUTOSAR specification relies on informal and semi-formal representation, namely UML, of systems which opens the door for design errors due to ambiguity and clarity challenges of semi-formal notations. Indeed, several iterations and releases of the specification were required to address a good number of such ambiguities. In the AUTOSAR Layered Software Architecture, the FlexRay State Manager belongs to the Services Layer, or more precisely, to the Communication Services as shown in Figure 1 FlexRay Communication Stack Architecture [10].

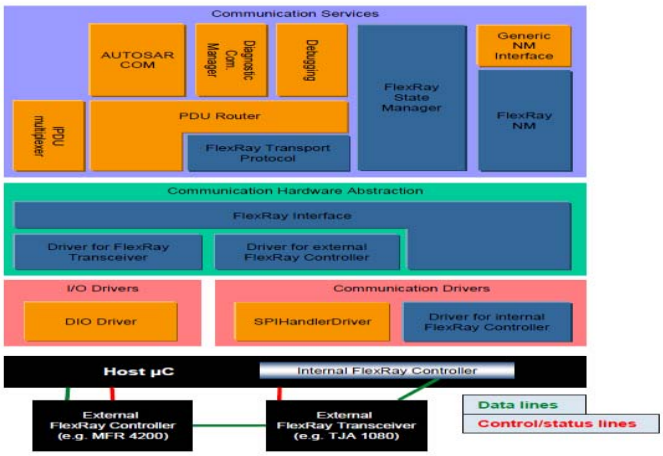

Figure 1 FlexRay Communication Stack Architecture

Testing AUTOSAR software modules as well as automotive software functions is quite a challenge. Systems are still being tested using adhoc based test designs. Industry tends to focus on HiL (Hardware in the loop) testing after all software and hardware components are integrated [11][12]. There is also dependency on modeling the system and depending on manually designed simulations use-cases to verify a system functionality and specification compliance [13][14]. Existing approaches are not enough and industry is at a loss between the reluctance to adopt new verification measures and change their legacy flow and coping with new requirements from regulatory bodies to ensure that safety is adhered to in all possible measures. Automotive industry, for one, is faced with a new safety standard, ISO 26262, that aims to ensure system safety through several guidelines and mandates in every stage of system development. In general, there are a lot of challenges facing embedded safety critical system development and it is crucial to identify solutions that integrate ISO 26262 guidelines in a non-disruptive approach to the industry [15] and fulfill the driving forces to establish AUTOSAR standardization.

B. Functional Safety – ISO 26262

ISO 26262 is a functional safety standard that publishes its objectives as: providing an automotive safety lifecycle (management, development, production, operation, service, decommissioning) and supports tailoring the necessary activities during these lifecycle phases, providing an automotive specific risk-based approach for determining risk classes (Automotive Safety Integrity Levels, ASILs). It is divided into 10 parts namely, vocabulary, management of functional safety, concept phase, product development at the system level, product development at the hardware level, product development at the software level, production and operation, supporting processes, ASIL oriented and safety oriented analysis and finally guidelines on ISO 26262 [16]. ISO 26262 architecture design guidelines aim to ensure that the software architectural design captures the information necessary to allow the subsequent development activities to be performed correctly and effectively and that it shall be described with appropriate levels of abstraction by using formal/semi-formal or informal notations for software architectural design. Table 1 Software Architectural Design Methods enumerates the methods that ISO 26262 recommends an architectural designer uses to capture system design. (‘++’ indicates that the method is highly recommended for the identified ASIL, ‘+’ indicates that the method is recommended for identified ASIL, ‘o’ means no recommendation) [16].

Table 1 Software Architectural Design Methods

Methods		**ASIL**			
		A	**B**	**C**	**D**
1a	Informal notations	++	++	+	+
1b	Semi-formal notations	+	++	++	++
1c	Formal notations	+	+	+	+

ISO 26262 highly recommends the usage of semi-formal notations to capture design elements with a primary aim of being as unambiguous as possible.

C. Symbolic Analysis Laboratory: SAL

SAL is a framework for combining different tools for abstraction, program analysis, theorem proving, and model checking toward the calculation of properties (symbolic analysis) of transition systems [3]. A key part of the SAL framework is an intermediate language for describing transition systems. This language is intended to serve as the target for translators that extract the transition system description for other modeling and programming languages, and as a common source for driving different analysis tools [3]. SAL intermediate language is a basic transition system language. SAL describes transition systems in terms of initialization and transition commands [3]. The current generation of SAL tools contains a group of state of the art LTL based model checkers and auxiliary tools based on them. In the future, it is expected that the tools will be expanded to

include static analysis, invariant generation, abstraction, and a tool bus to connect these tools together . SAL framework comes with a variety of existing tools that verifies transition based systems. The checkers include sal-smc which is a BDD-based model checker for finite state systems. The checker confirms whether a specific theorem holds or not and asserts counterexamples showing how a theorem can be invalidated on a specific state machine . The model checker can do forward and backward search and prioritized traversal as well. The checker is for finite systems. Sal-deadlock-checker is a SAL auxiliary tool based on the SAL symbolic model checker to detect deadlock states. Sal-bmc is a bounded model checker based on SAT solver for finite state systems. It generates counterexamples/assertions and detects bugs via refutation. It also can perform verification by k-induction. SAL can use several SAT solvers but defaults to Yices. Sal-inf-bmc is an infinite bounded model checker for infinite state systems based on SMT solvers

III. FLEXRAY STATE MANAGER

We will use FlexRay State Manager module to demonstrate our proposed implementation and specification compliance flow [10]. Figure 2 FlexRay State Manager State Machine illustrates FlexRay State manager UML state diagram as presented in AUTOSAR's specification. The State machine model combined with state machine to SAL transformation rules shown below shall be applied in order to generate the SAL based system model.

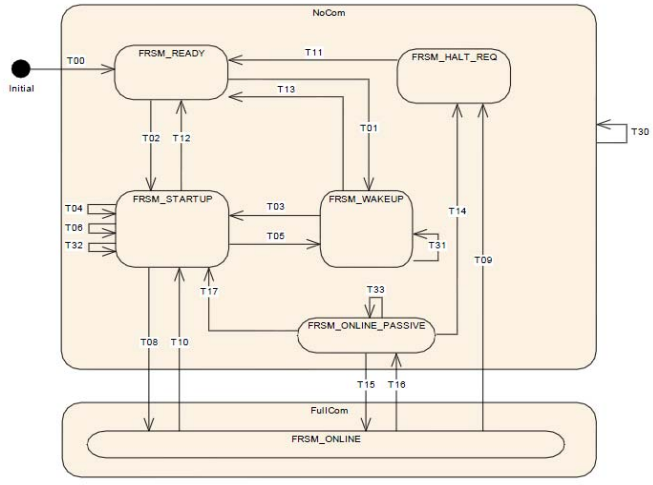

Figure 2 FlexRay State Manager State Machine

Rule 1 – Create a SAL context for FlexRay UML Class element

```
FlexRay_SM : CONTEXT =
BEGIN
```

Rule 2 – Create a SAL state type for FlexRay states

```
state : TYPE = {FRSM_INIT, FRSM_READY, FRSM_HALT_REQ,
FRSM_STARTUP, FRSM_WAKEUP, FRSM_ONLINE_PASSIVE,
FRSM_ONLINE};
```

Rule 3 – Create a SAL state machine module and a current state variable of type state – created in previous step

```
main : MODULE =
        BEGIN
                OUTPUT current_state : state
```

Rule 4 – Create input SAL variables for each transition precondition.

```
        INPUT isWakeupECU : BOOLEAN
        INPUT isColdStartECU: BOOLEAN
        INPUT StartupRepetitionsWithWakeup: [0..255]
        INPUT startupRepetitions : [0..255]
        INPUT pocFreeze : BOOLEAN
```

Rule 5 – Create output SAL variables for each transition postcondition/action.

```
        OUTPUT fe_config : BOOLEAN
        OUTPUT t3_IsActive : BOOLEAN
        OUTPUT t3_fired : BOOLEAN
        OUTPUT t2_fired : BOOLEAN
        OUTPUT t1_fired : BOOLEAN
```

Rule 6 – Create SAL initialization clause that maps initial state machine into SAL notation.

```
INITIALIZATION
        current_state = FRSM_INIT;
```

Rule 7 – Create SAL transition table based on UML transitions, preconditions and post conditions.

```
TRANSITION
[
% Transition 0
current_state = FRSM_INIT AND fe_config = TRUE
--> current_state' = FRSM_READY
% Transition 1
[]
current_state = FRSM_READY AND reqComMode =
COMM_FULL_COM AND isWakeupECU = TRUE
--> current_state' = FRSM_WAKEUP;
        fe_trcv' = normal;
        startupCounter' = 1;
        fe_wakeup' = TRUE;
]
```

UML to SAL transformation is now complete and a FlexRay state manager in SAL transition based languages augmented with complex data types is constructed from a UML finite state machine input. Next stage involves constructing the theorems.

A. Requirement Specification to SAL Theorems

Compliance requirements are drawn from specifications. In this section we show several examples where we map specification requirements to SAL theorems.

FlexRay State Manager specification details the expected state transitions pre and post conditions. Figure 3 Transition 01 Preconditions and Post Actions shows one state transition pre and post conditions as documented in the specification [10].

FrSm072	T01	[reqComMode = FullCom AND IsWakeupECU]	FE_TRCV_NORMAL StartupCounter := 1 FE_WAKEUP

Figure 3 Transition 01 Preconditions and Post Actions

Our first theorem was constructed via mapping the specification transition table entry to a theorem. Transition 01 switches the state machine from FRSM_READY to FRSM_WAKEUP when reqComMode = FullCom and IsWakeupECU is true. The actions that will take place after transition happens are as shown in Figure 3. A transition validation via the model checker could be done based on generating a theorem from the transition pre-post conditions as follows:

Rule 1: Map transition precondition to theorem pre-requisite

Th<id>: THEOREM main |- G(reqComMode = COMM_FULL_COM AND isWakeupECU = TRUE => F());

Rule 2: Map current state into machine pre-requisite

Th<id>: THEOREM main |- G(reqComMode = COMM_FULL_COM AND isWakeupECU = TRUE => F());

Rule 3: Map transition actions into theorem outcome

Th<id>: THEOREM main |- G(reqComMode = COMM_FULL_COM AND isWakeupECU = TRUE AND current_state = FRSM_READY => F(fe_trcv = normal AND startupCounter = 1 AND fe_wakeup = TRUE));

Rule 4: Map resulting new state into theorem outcome

Th<id>: THEOREM main |- G(reqComMode = COMM_FULL_COM AND isWakeupECU = TRUE AND

current_state = FRSM_READY => F(fe_trcv = normal AND startupCounter = 1 AND fe_wakeup = TRUE AND current_state = FRSM_WAKEUP));

Our second theorem was constructed based on constraints in the specification. The specification defines the following requirement:

FrSm034: StartupRepetitions determines how often the ECU can repeat the startup procedure by reinitializing the FlexRay CC. This value must not be smaller than StartupRepetitionsWithWakeup.

Th<id>: THEOREM main |- G(NOT(startupRepetitions < StartupRepetitionsWithWakeup));

Our third theorem was constructed based on boundary value constraints in the specification. The specification defines the following requirement:

FrSm033: startupCounter is the number of startup attempts that have been performed. Valid values are in the range of 0-255

Th<id>: THEOREM main |- G(NOT(startupCounter < 0 AND startupCounter > 255));

Constructed theorems one, two, and three are examples for ways to construct theorems from specification. The theorems have been constructed to ensure compliance to state machine requirements, constraints satisfaction and boundary value constraints. ISO-26262 recommends using several techniques to ensure design correctness as shown in ements are mapped to theorems.

Table 2 Error Detection at Software Architectural Level [16]. It is possible to easily construct theorems to ensure design correctness as proposed in the safety standard as we have shown above. Assertions will be reported by the solvers if the SAL design of FlexRay State manager module contains any in-compliances to the requirements in the specification since requirements are mapped to theorems.

Table 2 Error Detection at Software Architectural Level

Methods		**ASIL**			
		A	**B**	**C**	**D**
1a	Plausibility check-a	++	++	++	++
1b	Detection of data errors-b	+	+	+	+
1c	External monitoring facility	O	+	+	++
1d	Control Flow monitoring	O	O	+	++

a- Plausibility checks include assertion checks. Complex plausibility checks can be realized by using a reference model of the desired behavior.

b- Types of methods that may be used to detect data errors include error detecting codes and multiple data storage.

B. Results

SAL SMT solver was run against the mapped SAL model and the constructed theorems. First theorem run showed that the theorem holds and that there are no violations against the specification. We started introducing errors or incompliances against the transition table specification in the UML model. The checker was successfully able to assert counterexamples showing how the theorem got violated. The requirement inFigure 4 theorem was reported as 'proved' by the SMT Solver.

FrSm124	T33	[NOT t3_IsActive]	FE_STARTUP_ ERROR_IND

Figure 4 Transition 33 Preconditions and Post Actions

The requirement inFigure 4 dictates that transition 33 should only take place when t3_IsActive is false. If so, the state machine should make an internal transition from FRSM_ONLINE_PASSIVE to itself. The constructed theorem for the above specification based on our mapping algorithm is:

th1: THEOREM main |- G((current_state = FRSM_ONLINE_PASSIVE AND t3_IsActive = FALSE) => F(current_state = FRSM_ONLINE_PASSIVE));

When the checker is first run against the SAL model constructed from the UML Model. It reports that the theorem was proved. We then changed the theorem such that an error in the SAL model would be detected where we indicate that if T3_IsActive is true, the state machine should stay in FRSM_ONLINE_PASSIVE state which is now a specification violation and is expected to be detected by the checker. We run the checker which asserts a 5 step counterexample that demonstrates the theorem violation in the model. Counterexample steps asserted by the model checker were:

Step1:

reqComMode = COMM_NO_COM , isWakeupECU = false, isColdStartECU = true, StartupRepetitionsWithWakeup = 101, pocState = wakeup, startupRepetitions = 233, pocFreeze = true

Current State: INIT

New State: READY

Step2:

reqComMode = COMM_NO_COM, isWakeupECU = false, isColdStartECU = true, StartupRepetitionsWithWakeup = 57, pocState = normalActive, startupRepetitions = 85, pocFreeze = false

Current State: READY

New State:STARTUP

Step3:

reqComMode = COMM_SILENT_COM, isWakeupECU = false, isColdStartECU = true, StartupRepetitionsWithWakeup = 71,

pocState = normalActive, startupRepetitions = 58, pocFreeze = false

Current State: STARTUP

New State: ONLINE

Step4:

reqComMode = COMM_SILENT_COM, isWakeupECU = false, isColdStartECU = false, StartupRepetitionsWithWakeup = 33,

pocState = normalPassive, startupRepetitions = 112, pocFreeze = false

Current State: ONLINE

New State: ONLINE_PASSIVE

Step5:

reqComMode = COMM_SILENT_COM, isWakeupECU = false

isColdStartECU = false, StartupRepetitionsWithWakeup, = 5, pocState = wakeup, startupRepetitions = 7, pocFreeze = false

Current State: ONLINE PASSIVE

New State: ONLINE Passive

Similarly, for theorems 2 and three, SAL SMT checker was run and reported that there was no violation in SAL model for theorem 3 but yet reported a violation for theorem 2 and listed counterexample to showcase the violation as shown below

$ sal-inf-bmc FlexRay_SM th7

Counterexample:

==========

Path

==========

Step 0:

--- Input Variables (assignments) ---

reqComMode = COMM_NO_COM

isWakeupECU = false

StartupRepetitionsWithWakeup = 255

pocState = halt

startupRepetitions = 0

pocFreeze = false

--- System Variables (assignments) ---

fe_config = false

t3_IsActive = false

t3_fired = false, t2_fired = false, t1_fired = true

current_state = FRSM_INIT

fe_trcv = standby

startupCounter = 255

fe_wakeup = false, fe_start = false, fe_allow_coldstart = false

fe_start_com_rx = false, fe_start_frif = false, fe_start_com_tx = false, fe_dem_status = failed, fe_full_com_ind = false

fe_stop_com_tx = false, fe_stop_frif = false

fe_stop_com_rx = false, fe_halt = false

fe_no_com_ind = false, fe_startup_error_ind = false

s0 = false, s1 = false, s2 = false, s3 = false, t0 = false

Several other theorems following the proposed algorithm have been proven by the checker. The proposed method proves to be able to utilize mathematical proofs based on constructing formal model based on semi-formal /informal AUTOSAR specification of FlexRay State Manager module.

IV. CONCLUSION

We showed how a UML finite state machine model can be transformed to a formal transition model augmented with complex data types in SAL notation. We have constructed theorems that represent specification requirements. SAL SMT solver was utilized so that a formal verification can be accomplished on the software model. It has been shown that specification incompliances could be detected by the SMT solver through the asserted counterexamples. The application model engineer could fix the model violations at a very early stage based on the formal verification of the model using the proposed approach. The proposed approach makes emerging ISO 26262 standard advisory guidelines possible in an automated fashion and addresses one of the major drives behind AUTOSAR standardization which is early design errors detection. We have showed that SAL is a simple transition based notation with extreme resemblance to UML state machine diagram. Problems such as the complexity of the Formal notations, theorem construction and checkers execution and analysis have been masked away via the use-case yet the benefits of using formal validation are retained.

REFERENCES

[1] Gwangmin Park ; Daehyun Ku ; Seonghun Lee ; Woong-Jae Won, Test methods of the AUTOSAR application software components, ICCAS-SICE, 2009.

[2] Mews, M. ; Svacina, J. ; Weissleder, S. ,From AUTOSAR Models to Co-simulation for MiL-Testing in the Automotive Domain, Software Testing, Verification and Validation (ICST), 2012 IEEE.

[3] http://sal.csl.sri.com/introduction.shtml.

[4] Taguchi, K. ; Nishihara, H. ; Aoki, T. ; Kumeno, F. , Building a Body of Knowledge on Model Checking for Software Development, Computer Software and Applications Conference (COMPSAC), 2013 IEEE.

[5] Whittle, J. , Hutchinson, J. , Rouncefield, M., "The State of Practice in Model-Driven Engineering", Software, IEEE (Volume:PP , Issue: 99), 23 April 2013.

[6] Galvao, I. , Enschede ; Goknil, A., "Survey of Traceability Approaches in Model-Driven Engineering",Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International, Oct. 2007.

[7] http://www.autosar.org/

[8] http://www.autosar.org/download/R4.0/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf.

[9] http://www.autosar.org/download/AUTOSAR_TechnicalOverview.pdf

[10] http://www.autosar.org/download/R3.1/AUTOSAR_SWS_FlexRay_StateManager.pdf.

[11] Hermans, T. ; Ramaekers, P. ; Denil, J. ; Meulenaere, P.D. ,Incorporation of AUTOSAR in an Embedded Systems Development Process: A Case Study, Software Engineering and Advanced Applications (SEAA), 2011.

[12] Huang Bo ; Dong Hui ; Wang Dafang ; Zhao Guifan,Basic Concepts on AUTOSAR Development, Intelligent Computation Technology and Automation (ICICTA), 2010.

[13] Huichoun Moon ; Gwanghun Kim ; Yeongyun Kim ;Seokkyoo Shin Automation Test Method for Automotive Embedded Software Based on AUTOSAR, Software Engineering Advances, 2009. ICSEA '09.

[14] Wainer, G , Applying modelling and simulation for development embedded systems, Embedded Computing (MECO), 2013.

[15] Knight, J.C. , "Safety critical systems: challenges and directions", Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on Software Engineering, May 2002.

[16] http://www.iso.org/iso/catalogue_detail?csnumber=43464 .