The Role of Architectural Model Checking in Conducting Preliminary Safety Assessment

Omar T. Jaradat; Mälardalen University; Västerås, Sweden

Patrick Graydon; Mälardalen University; Västerås, Sweden

Iain J. Bate; The University of York; York, UK, and Mälardalen University; Västerås, Sweden

Keywords: Architecture, Model Checking, Argument, Standard, Conformance, Certification

Abstract

Preliminary safety assessment is an important activity in safety systems development since it provides insight into the proposed system's ability to meet its safety requirements. Because preliminary safety assessment is conducted before the system is implemented, developers rely on high-level designs of the system to assess safety in order to reduce the risk of finding issues later in the process. Since system architecture is the first design artefact developers produce, developers invest considerable time in assessing the architecture's impact on system safety. Typical safety standards require developers to show that a plan of safety activities, chosen from recommended options or alternatives, meets a set of objectives. More specifically, the automotive safety standard ISO 26262 recommends formally verifying the software architecture to show that it "complies" with safety requirements.

In this paper, we apply an architecture-based verification technique for Architecture Analysis and Design Language (AADL) specifications to an architectural design for a fuel level estimation system to validate certain architectural properties. Subsequently, we build part of the conformance argument to show how the model checking can satisfy some ISO 26262 obligations. Furthermore, we show how the method could be used as a part of preliminary safety assessments and how it can be upheld by the later implementations beside of the other recommend methods.

Introduction

Preliminary safety assessment is one of the major activities performed during the system development. Typically, the assessment is conducted during or after the architectural design phase. Throughout this activity, system design decisions are assessed to establish whether the design implements (or at least does not preclude satisfying) the safety requirements (ref. 1). Getting the architecture right is crucial for two reasons. First, eliminating design faults at this development stage is much cheaper than in later stages (ref. 2). Second, some mistakes might be missed by later verification and validation activities, potentially leading to an accident. To address these development and operational risks, preliminary safety assessment should provide evidence showing that the system architecture is complete, consistent and correct with respect to the safety requirements.

In some domains developers must demonstrate conformance to a safety standard. This typically requires showing that the software architecture has certain required properties. In particular, the automotive safety standard ISO 26262 (refs. 3, 4) defines several verification objectives for the software architecture design. Broadly speaking, these objectives cover the three Cs (ref. 5): completeness, correctness and consistency. ISO 26262 requires or recommends various means of satisfying these objectives depending upon the criticality of the software. Following hazard and risk assessment, developers assign an automotive-specific Automotive Safety Integrity Levels (ASILs) for the software (ref. 3). The standard's requirements and recommendations for verifying the software architecture depend upon the software's ASIL (ref. 3).

ISO 26262 recommends formally verifying software architectural designs at the highest ASIL levels C and D (ref. 3). Model checking is one means of such verification. Model checking is particularly desirable because model checkers can produce counterexamples illustrating how and where verification checks fail. Model checkers can detect deadlocks, incorrect interactions, especially within the concurrency, of the model. The techniques can also identify correctness, completeness, or coherency issues with the model. All these issues can lead to the final system failing and possibly causing harm (ref. 6). However, the standard states that the verification typically applies a combination of different methods (e.g., review, walk-through, inspection, model-checking, simulation, engineering analyses, demonstration, testing, etc.) (ref. 3). This is because no single verification method provides a complete means of assessing the completeness and correctness of the architecture specification. Consequently, the model checking evidence can partially satisfy an objective but it should be complemented with other forms of evidence recommended by the standard.

The contributions of this paper are to use a specimen system to (1) demonstrate how evidence from an existing architectural model checking technique contributes to a preliminary safety assessment activity and an ISO 26262 conformance argument, (2) makes a critical analysis of the strength of that argument, and (3) recommends how to complement model checking evidence to achieve complete coverage of ISO 26262 objectives.

Fuel-Level Estimation System (FLES)

We illustrate the concepts discussed in this paper with examples drawn from a real safety-critical system. This system estimates the volume of fuel in a heavy road vehicle's tank and presents this information to the driver through a dashboard mounted fuel gauge. Additionally, the system must warn the driver when this volume falls below a predefined threshold. This system is considered safety critical because its failure could lead to loss of control of the vehicle. For example, if there is less fuel remaining than the driver thinks, the vehicle might run out, bringing it to an unexpected halt. In the wrong context, this can be hazardous. As well as bringing the vehicle to a halt, the power steering and braking mechanisms could also fail. These failures would compromise vehicle controllability and could also lead to a crash.

Fuel volume is estimated using a float sensor in the fuel tank. As the position of the float is affected by vehicle motion (negotiating steep hills, sharp bends, or rough terrain), the system has some challenging issues to be tackled within its design. The system must process this signal so that at all times the gauge displays an accurate measurement of the total volume of fuel remaining.

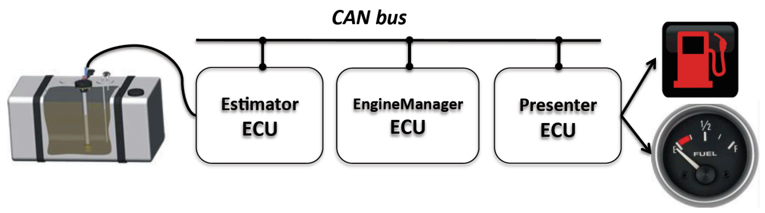

Figure 1 — The Fuel Level Estimation System

An Architecture Analysis & Design Language (AADL) Model of the System

AADL is standardized in SAE AS5506B (ref. 7) and is intended to support model-based analysis and specification of complex real-time embedded systems. AADL models system architectures using a component-and-connector structure which represents both the component(s) and their interactions. Models can be expressed in either a text notation or graphically (ref. 8). AADL supports description of the mechanisms of exchange and control of data such as message passing, event passing, synchronized access to shared components, thread scheduling protocols, timing requirements, and remote procedure calls. Moreover, the language facilitates descriptions of elements' characteristics by associating these characteristics with specific property annotations such as, timing, runtime, and synchronization properties (ref. 9).

An important issue to be considered is that the fuel level estimation system is one of many interconnected vehicle subsystems and is reliant on these systems for data and control flow. For the sake of simplicity, however, we limit the work in this paper to the functions located in the Estimator ECU. Figure 2 illustrates a graphical AADL architectural model for the software running on the Estimator ECU. The software comprises a number of processes, each with one or more threads. Table 1 enumerates these threads and their timing properties. The *BasicSoftware* process contains the *SoftwareIN* and *SoftwareOUT* threads. The *FuelEstimationCalculation* process contains the *FuelEstimation* thread. The *FuelLevelWarningCalculation* process contains the *FuelLevelWarning* thread. The Estimator ECU implements several unrelated functions; we represent these as *Other_Functions* in the figure and table.

Table 1 — Timing Properties for the FLES Threads in the Estimator ECU

Thread Name	Dispatch Protocol	Period (ms)	Deadline (ms)	Execution Time (ms)	Priority
SoftwareIN	Periodic	10	1	1	1
FuelEstimation	Periodic	10	2	1	2
FuelLevelWarning	Periodic	10	3	1	3
Other_Functions	Periodic	10	6	3	4
SoftwareOUT	Periodic	10	7	1	5

The *SoftwareIN* thread reads the sensed fuel float position from the *ADC* and stores this in the real-time database *RTDB*. *FuelEstimation* reads this sensor value and computes an estimate of the current fuel volume in litres. When the vehicle might be moving (i.e., its parking brake is not set), the *FuelEstimation* thread uses a Kalman filter algorithm to reduce the noise introduced by vehicle motion. This algorithm requires the recent history of fuel volume estimates to be stored. *FuelEstimation* outputs a smoothed fuel volume estimate to the *RTDB*. *FuelLevelWarning* then reads this estimate, compares it to the low-fuel warning threshold, and writes the low-fuel warning status to the *RTDB*. *SoftwareOUT* reads the fuel volume and low-fuel warning status from the *RTDB* and sends these over the Controller Area Network (CAN) bus to the *Presenter*. These threads are dispatched periodically every 10 milliseconds. It is important that the events are performed in the order described above; otherwise the end-to-end latencies will increase.

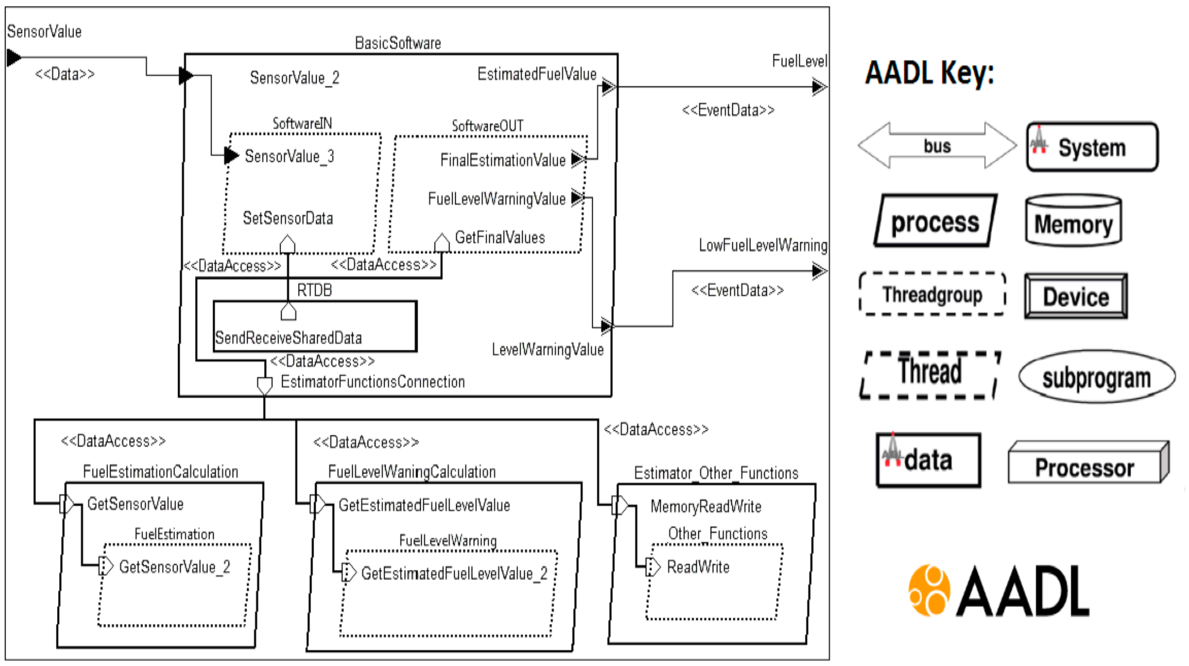

Figure 2 — Architecture of Estimator Software

<u>Software Architecture Verification Obligations From ISO 26262</u>

ISO 26262 requires developers to produce a *safety case* (ref. 3), which it defines as *an argument that the safety requirements for an item are complete and satisfied by evidence compiled from work products of the safety activities during development* (ref. 3). This definition is subtly different from more typical definitions of safety case (e.g., refs. 4, 10) in that its argument traces safety evidence to requirements rather than to an operational definition of adequate safety.

In this part of the paper, we present the software architecture verification obligations that ISO 26262 requires developers to address in their safety cases. In later parts, we will show how evidence derived from model checking a software architecture expressed in AADL can contribute to satisfying these obligations. To the extent that these obligations are similar to obligations imposed by other standards, or describe details that developers would need to present in a traditional safety case, our discussion will be useful to developers who are not conforming to ISO 26262.

ISO 26262 defines the objective of software architecture verification as (ref. 3):

- **Ensure that the architecture "complies" to software safety requirements** (i.e., that the architecture supports the meeting of the requirements and does not inhibit them being met).
- **Demonstrate that the architecture compatible with the target hardware.**
- **Demonstrate that the architecture adheres to design guidelines.**

ISO 26262 imposes two obligations to meet these objectives. The first obligation is developers shall verify that the software architecture was designed according to the standard's general requirements and recommendations regarding software architecture design. This is a generic obligation, i.e. not limited to the software architecture verification, throughout the lifecycle. To conform to this obligation, developers must (ref. 3):

- **Create a verification plan.** Developers shall carry out a verification planning for each phase and sub-phase of the safety lifecycle.
- **Consider the verification specification**. Developers shall select and specify the methods to be used for the verification (e.g., test cases, simulation scenarios, analysis, etc.).
- **Execute and evaluate the verification**. Developers shall execute the verification as planned.

The second obligation is the recommendation of software architectural design verification methods, as shown in Table 2. It is related to the first obligation, however it is specific to software architecture verification. The standard recommends using these methods based on the determined ASIL for the system in question.

Table 2 — Methods for the Verification of the Software Architectural Design (ref. 3)

Methods		**ASIL**			
		A	**B**	**C**	**D**
1a	Walk-through of the design[a]	++	+	o	o
1b	Inspection of the design[a]	+	++	++	++
1c	Simulation of dynamic parts of the design[b]	+	+	+	++
1d	Prototype generation	o	o	+	++
1e	Formal verification	o	o	+	+
1f	Control flow analysis[c]	+	+	++	++
1g	Data flow analysis[c]	+	+	++	++

Where:

a	In the case of model-based development these methods can be applied to the model.
b	Method 1c requires the usage of executable models for the dynamic aspects of the software architecture.
c	Control and data flow analysis may be limited to safety-related components and their interfaces.
++	Highly recommended for the identified ASIL.
+	Recommended for the identified ASIL.
o	No recommendation for or against its usage for the identified ASIL.

An Architecture-Based Verification Technique for AADL Specifications

Johnsen et al. have proposed a technique for verifying architectural specifications (ref. 2). The technique verifies completeness and consistency properties using model checking and facilitates testing the implementation through model-based testing. In this paper, we explain how this technique can be used in an ISO 26262 safety case to address verification obligations related to the software architecture.

AADL facilitates specification of real-time and schedulability properties of the architecture such as dispatch protocol, deadline, execution time, scheduling policy, and period. That portion of a specification might conflict with data flow implied by connections between components: thread priorities might dictate that a component be executed before its input data is available. The verification technique can detect such inconsistencies, check control and data reachability, and verify concurrency properties (ref. 11). The analysis assumes fixed-priority preemptive scheduling, using non-preemptive scheduling.

The basis for the technique is the control and data flows between the interfaces of AADL components. Formally, the verification technique specifies five types of relations. The relations define the possible binding of control and data, and they are used to generate the control and data flows diagram. From these relations, three verification sequences were derived. These sequences specify the possible paths in the control- and data-flow diagrams, and generate the verification sequences. (ref. 2)

The semantics of AADL are not defined formally. AADL models describe the dispatching and scheduling semantics for their components using natural language text. Without normal semantics it is difficult to formally verify AADL specifications (ref. 12). However, we can work around this problem using a methodology called semantic anchoring, in which AADL model is transformed into timed automata constructs through well-defined transformation (mapping) rules (ref. 11).

In this work, we use the UPPAAL model checker named for (UPPsala University in Sweden and AALborg University in Denmark) (ref. 13). To verify the architecture, developers should follow the following steps (ref. 2):

- **Use the transformation rules to transform the AADL specification to an UPPAAL model**.

- **Apply the architecture-based verification criteria to AADL model to extract control and data flows sequences**. Verification criteria determine which samples of the specification to evaluate, how they are extracted, and how many samples to evaluate. The samples generated from the criteria are sequences of control flows and data flows among interconnected components.

- **Determine the AADL sequences from the previous step in the corresponding automata paths**. Developers should determine the paths between UPPAAL automata according to the extracted AADL control flows and data flows sequences. This determination is done through a structural mapping between AADL and UPPAAL models.

- **Use UPPAAL temporal logic to check the properties for the sets of automata paths**. After determining the automata paths in the previous step, developers are able now to simulate and verify the properties of these paths. The verdicts from the simulations reveal the consistency and completeness of the AADL specification, where a correction of the specification should be made if it is shown inconsistent or incomplete.

Developers interested in determining whether the analysed properties hold for the delivered system can use the temporal logic queries (that have been used during the model checking) to generate test cases to test the conformance of the implementation with respect to the architecture specification (i.e., model-based testing).

Applying the Verification Technique To the Fuel Level Estimation System

To check the example fuel level estimation system using this technique, we transformed the five threads listed in Table 1 into UPPAAL automata. To these we added a sixth automaton, the *Scheduler*, to simulate the scheduling function of the Estimator ECU operating system. We applied the verification criteria to the AADL model (see Figure 2) to extract the following verification sequences:

- SoftwareIN.SetSensorData →FuelEstimation.GetSensorValue_2 → FuelLevelWarning.GetEstimatedFuelLevelValue_2 → SoftwareOUT.GetFinalValues → SoftwareOUT.FuelLevelWarningValue

- SoftwareIN.SetSensorData → FuelEstimation.GetSensorValue_2 → SoftwareOUT.GetFinalValues → SoftwareOUT.FinalEstimationValue

- SoftwareIN.SetSensorData → OtherFunctions.ReadWrite → SoftwareOUT.GetFinalValues

According to the fourth step in the previous section, we determined the paths between UPPAAL automata according to the extracted verification sequences. To relate the UPPAAL models to the AADL model shown in Figure 2, we used the names of AADL threads and their interfaces as identifiers in the UPPAAL automata. The automata model the task timing characteristics encoded in the AADL model and presented in Table 1. To automatically simulate the model, we used the UPPAAL verifier to write temporal logic queries to check if whether the paths' properties are satisfied or not. More specifically, we checked:

- Whether the scheduler automaton dispatch the automata (i.e., transformed AADL threads) according to their specified priorities.
- Whether the automata meet their specified deadlines.
- The data-flow reachability for each automaton in the paths.
- The control-flow reachability for each automaton in the paths.

The fuel level estimation is a single core system with non-preemptive scheduling and little computation done in interrupt handler. Therefore, there is no concurrency that exists during thread execution.

Conformance Argument

ISO 26262 states that "*The safety case should progressively compile the work products that are generated during the safety lifecycle*" (ref. 3). An option for satisfying this requirement is a conformance argument that contains the results of all associated requirements from the standard. For the software architecture verification of the fuel level estimation system, compliance to ISO 26262 means that the conformance argument shall contain results (i.e., evidence) from study, analysis or test of the associated verification methods from the standard. The determined ASIL for the fuel level estimation system is "C" and according to the data in Table 2, formal verification (model checking) is recommended method by ISO 26262.

In this part of the paper, we build a conformance argument in which we show the contribution of the model checking, in the last two sections, to verify the essential software architectural properties outlined in the Introduction and Fuel-Level Estimation sections. We use the Goal Structuring Notation (GSN) (ref. 14) to record the argument's claims (known in GSN as goals and represented by rectangles), argument strategies (represented by parallelograms) and evidence (known in GSN as solutions and represented by circles). The conformance argument demonstrates that the architecture "complies" with software safety requirements, which is the standard's first objective for software architecture verification. We do not address how to demonstrate that the architecture is compatible with the target hardware and adheres to design guidelines (i.e., the second and third objectives); that is left to future work.

Figure 3 presents a fragment of the conformance argument. The fragment's top-level goal, SWArchCompliance, represents a claim that the software architecture complies with software safety requirements. Two context elements (rounded rectangles) provide information that helps the reader to understand this claim: Complies defines what 'complies' means and ImplConfAndDeliver clarifies that the argument is about the architecture *as specified*, not necessarily as the system implements it. The latter is a key point: checking the architecture is not sufficient to show that the future system implementation will satisfy the software safety requirement. What we can do instead is to check the architecture to determine whether it precludes the ability to implement these requirements or support it. Consequently, in the argument we claim that the architecture is free from the errors that can prevent the implementation of the safety requirements.

To satisfy the top goal SWArchCompliance, we argue over the software safety requirements. However, we focus only on the fuel estimation (SR1) in this fragment, leaving SR2 and SR3 undeveloped (represented by a goal with a rhombus underneath it). To satisfy SR1, we focus on the architectural properties that allow the fuel estimation. The argument claims that a bad estimation can be caused by both fuel moving about in the tank (i.e., sloshing) and the timeliness of the fuel level calculation. Goal FilterMaskSlosh addresses the former while ArchSpecCalc addresses the latter. The architecture includes a Kalman filter to mask the effect of sloshing fuel. The related goal, KalmanApp is left undeveloped. For the time properties, the architecture specifies appropriate completion time properties. Completion time relies on the execution time for each thread (ExeTimBugExcComp) in addition to release time (ArchSpecExeCompRe). We satisfy the claim about the execution time by showing that the architecture specifies a deadline for each thread (ThrDeadlineSpec) and a release condition for each thread (ThrReleaseSpec). The latter two goals are away goals in the conformance argument primary fragment. This means that they are located in another module of the argument called Arch. MC, as shown in Figures 3 and 4. The module Arch. MC contains most of the argument that the architecture achieves the necessary temporal characteristics.

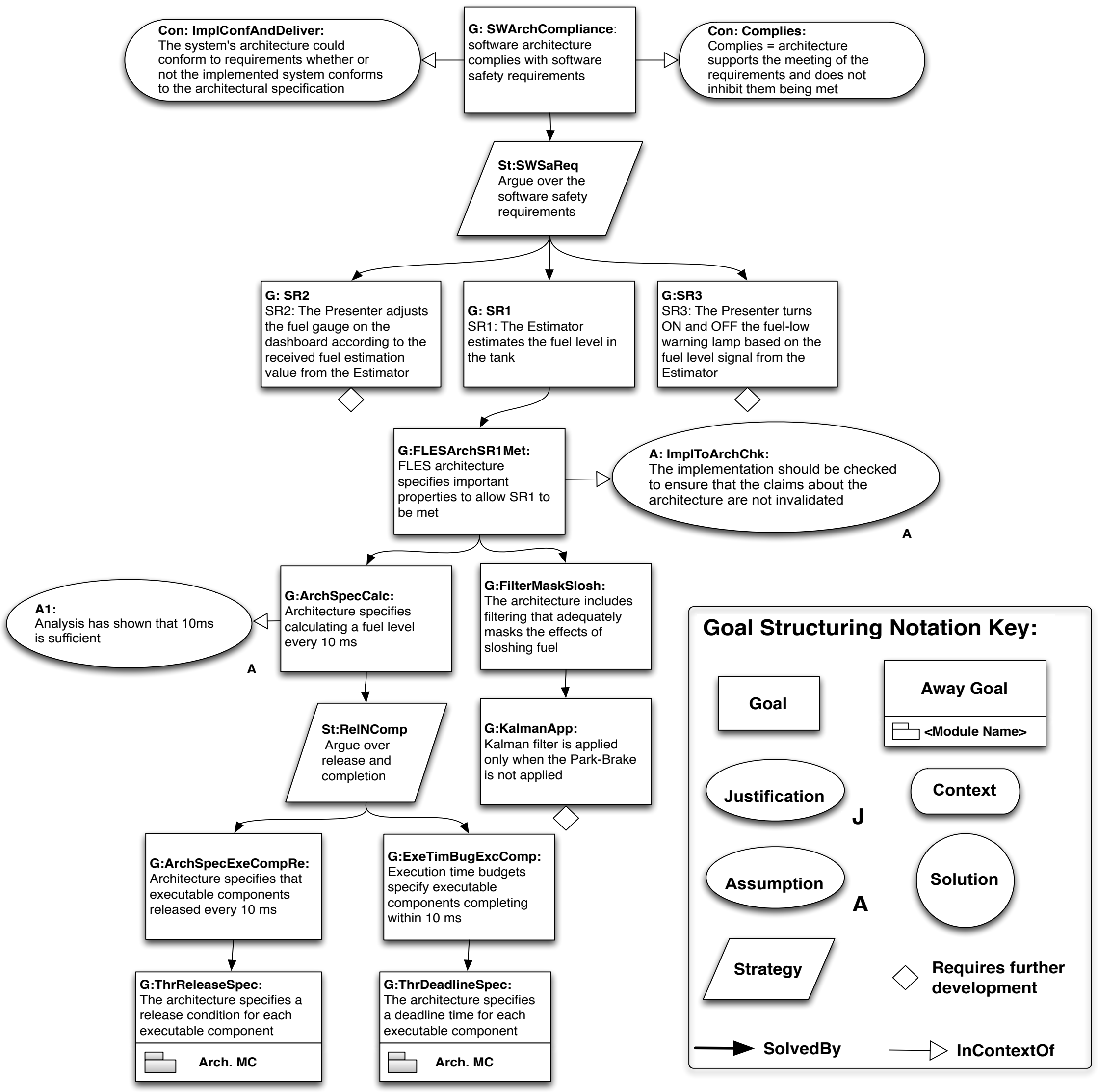
Con: ImplConfAndDeliver: The system's architecture could conform to requirements whether or not the implemented system conforms to the architectural specification
G: SWArchCompliance: software architecture complies with software safety requirements
Con: Complies: Complies = architecture supports the meeting of the requirements and does not inhibit them being met
St:SWSaReq Argue over the software safety requirements
G: SR2 SR2: The Presenter adjusts the fuel gauge on the dashboard according to the received fuel estimation value from the Estimator
G: SR1 SR1: The Estimator estimates the fuel level in the tank
G:SR3 SR3: The Presenter turns ON and OFF the fuel-low warning lamp based on the fuel level signal from the Estimator
G:FLESArchSR1Met: FLES architecture specifies important properties to allow SR1 to be met
A: ImplToArchChk: The implementation should be checked to ensure that the claims about the architecture are not invalidated
A
A1: Analysis has shown that 10ms is sufficient
A
G:ArchSpecCalc: Architecture specifies calculating a fuel level every 10 ms
G:FilterMaskSlosh: The architecture includes filtering that adequately masks the effects of sloshing fuel
St:RelNComp Argue over release and completion
G:KalmanApp: Kalman filter is applied only when the Park-Brake is not applied
G:ArchSpecExeCompRe: Architecture specifies that executable components released every 10 ms
G:ExeTimBugExcComp: Execution time budgets specify executable components completing within 10 ms
G:ThrReleaseSpec: The architecture specifies a release condition for each executable component
Arch. MC
G:ThrDeadlineSpec: The architecture specifies a deadline time for each executable component
Arch. MC
Goal Structuring Notation Key:
Goal
Away Goal
<Module Name>
Justification
J
Context
Assumption
A
Solution
Strategy
Requires further development
SolvedBy
InContextOf

Figure 3 — Conformance Argument Fragment

Figure 4 presents a module of the argument that is mainly related to the Architecture model checking Arch. MC. In this module, we claim that the software architecture of our system is free from the common errors that may occur (FLESArchFree). The strategy we adopted to satisfy this claim is to sort out the classes of common errors and argue over each one of them (ClsOfCommErr). Goals DeadlocksFree and LivelocksFree are solved by a control flow analysis (CtrlFloAn). DataAccess is solved by a data flow analysis (DataFloAn). NoFlowsContradiction is left undeveloped because (as mentioned earlier) FLES is a single core system with non-preemptive scheduling. Goal FLESTimeErrFree expresses a claim that the architecture is free from release time errors and execution time errors. To support it, we demonstrate that the architecture supports appropriate control of threads' execution (Correct-TemporalBehaviour). The architecture supports appropriate control if it specifies a release time condition for threads (ThrReleaseSpec), specifies deadlines (ThrDeadlineSpec), allows the release of the tasks within threads in a correct order (TaskOrderSpec) and specifies the correct priority for each executable component (ArchPrioSpec). These four goals are solved by syntactic analysis and preliminary timing analysis (SyntAn and PreTimAn).

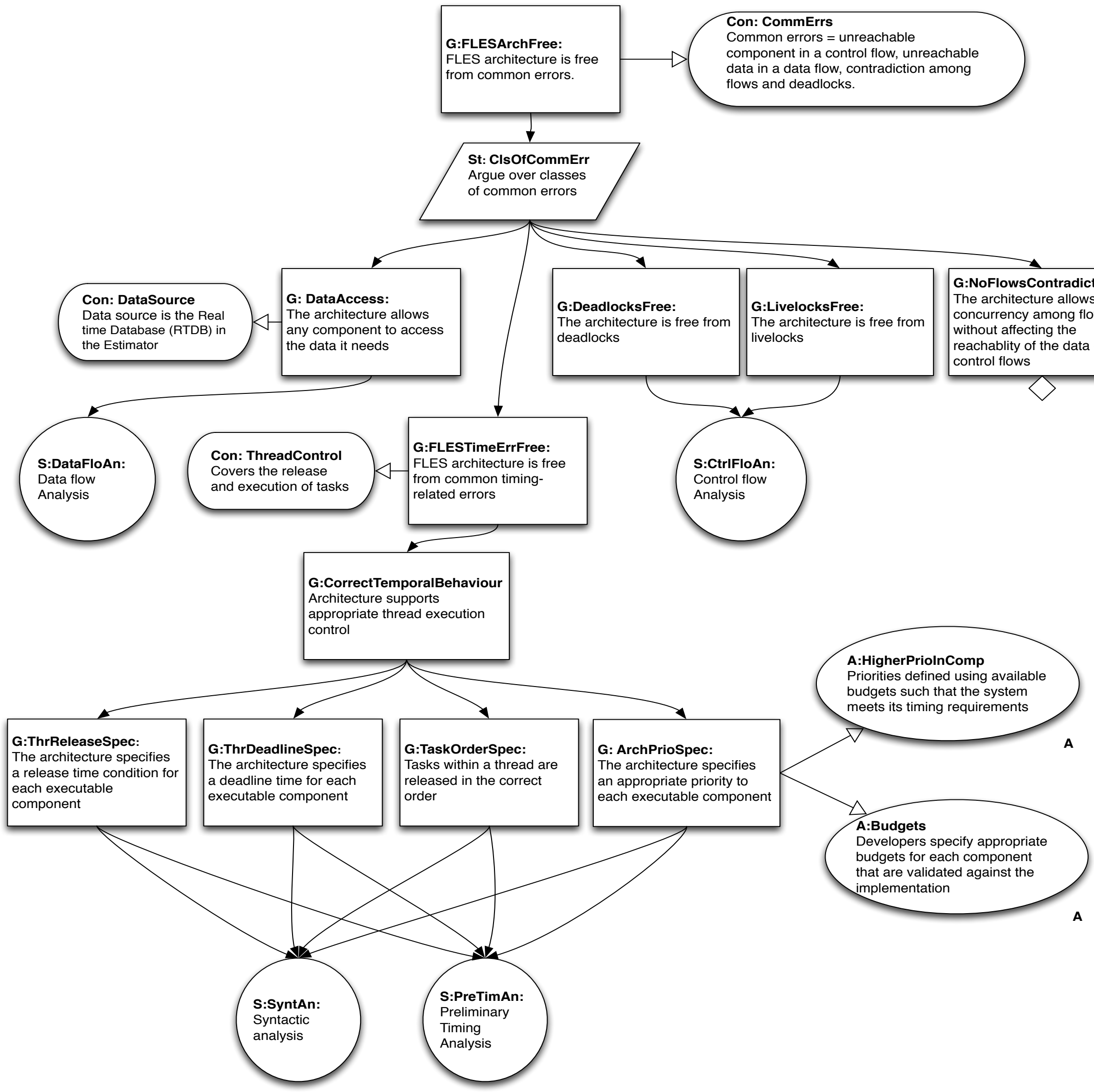
G:FLESArchFree: FLES architecture is free from common errors.
Con: CommErrs Common errors = unreachable component in a control flow, unreachable data in a data flow, contradiction among flows and deadlocks.
St: ClsOfCommErr Argue over classes of common errors
Con: DataSource Data source is the Real time Database (RTDB) in the Estimator
G: DataAccess: The architecture allows any component to access the data it needs
G:DeadlocksFree: The architecture is free from deadlocks
G:LivelocksFree: The architecture is free from livelocks
G:NoFlowsContradiction: The architecture allows concurrency among flows without affecting the reachablity of the data or control flows
S:DataFloAn: Data flow Analysis
Con: ThreadControl Covers the release and execution of tasks
G:FLESTimeErrFree: FLES architecture is free from common timing-related errors
S:CtrlFloAn: Control flow Analysis
G:CorrectTemporalBehaviour Architecture supports appropriate thread execution control
A:HigherPrioInComp Priorities defined using available budgets such that the system meets its timing requirements
A
G:ThrReleaseSpec: The architecture specifies a release time condition for each executable component
G:ThrDeadlineSpec: The architecture specifies a deadline time for each executable component
G:TaskOrderSpec: Tasks within a thread are released in the correct order
G: ArchPrioSpec: The architecture specifies an appropriate priority to each executable component
A:Budgets Developers specify appropriate budgets for each component that are validated against the implementation
A
S:SyntAn: Syntactic analysis
S:PreTimAn: Preliminary Timing Analysis

Figure 4 — Conformance Argument (Arch. MC Module)

Conformance Argument Analysis

A conformance argument, drafted early in the development lifecycle and updated as designs are changed and analyses completed, can help to contextualise and explain review, analysis, and test evidence. In this section we illustrate this benefit by providing a critical analysis of the conformance argument fragment presented in the previous section. Our analysis reveals how model checking contributes to meeting the standard's objectives. Moreover, the analysis identifies where additional information must be provided later (i.e., when the system implementation is available).

Provided Evidence: The conformance argument provides four solutions (i.e., evidence) as shown in Figure 4.

Data Flow Analysis (DataFloAn): The objective of data flow analysis, according to the verification technique, is to ensure that each data element flows from the source component that produces it to the target component where the data is used (ref. 11). However, the UPPAAL model derived from the AADL specification models only AADL threads and interconnection among their interfaces: data components that represent static data sharable among threads and their bindings are abstracted away. For the fuel level estimation system, the RTDB is a shared data component (as shown in Figure 2). The RTDB facilitates a communications path between components. But in the model checking it is assumed that the components communicate directly with each other and that the RTDB provides an appropriate (complete, consistent and coherent) concurrently accessible service. Once an implementation is available, this assumption would need to be validated. The end-to-end automata paths used for model checking can be used to generate test cases to check system implementation conformity to its architecture (to meet the assumption ImplToArchChk in Figure 3). Such evidence completes the argument since it shows that the system implementation conforms to its architecture and it is free from the classes of defects in question. Data flow analysis can be extended further to include the investigation of the values within the system, how they are associated to their variables and whether the associations affect the execution.

Control Flow Analysis (CtrlFloAn): The objective of control flow analysis, according to the verification technique, is to ensure that architectural elements are executed in an order that is consistent with the constraints of the control flow (ref. 11). The actual model checking done in our work simulates the control flows among the threads only, where all tasks and their flows contained within the threads are abstracted. The model checking confirms that the modelled architecture is schedulable and free from livelock and deadlock. However, we still need evidence to support the claims about control flows among the tasks contained in the threads. For the system implementation, a static analysis for the source code is required.

Preliminary timing analysis (PreTimAn): Given timing budgets (bounds on allowed execution times of tasks) and timing requirements, conventional schedulability analysis (ref. 15) can be used to show that the timing requirements are satisfied. The results are valid for the final system assuming the tasks' execution times conform to the ranges defined by their budgets and the other timing requirements.

Syntactic Analysis (SyntAn): The syntactic analysis is to check that the information contained in the AADL model is the same as in the final system. The risk of it not being the same could be reduced if autocode generators are used to produce the basic structure of the software. However this necessitates that the autocode generator is a trusted component, others manual review and testing would be needed to check the final implementation conforms to the architecture.

Summary: The following table summarizes where model checking contributed to the evidence and where other analyses contributed.

Table 3 - Analyses Contributions to Evidence

Claim Satisfied	**Evidence Source(s)**	**ISO26262 Clause Satisfied**
DataAccess	Model Checking	§7.4.5-b: Dynamic design aspects address the data flow between the software components; data flow at external interfaces.
DeadlocksFree	Model Checking	Part 6 – Annex D.2.2 - Timing and execution: effects of faults deadlocks can be considered for the software elements executed in each software partition
LivelocksFree	Model Checking	Part 6 – Annex D.2.2 - Timing and execution: effects of faults livelocks can be considered for the software elements executed in each software partition
ThrReleaseSpec	Syntactic Analysis, Preliminary Timing Analysis	§7.4.5-b: Dynamic design aspects address the functionality and behaviour; the control flow and concurrency of processes; and the temporal constraints
ThrDeadlineSpec	Syntactic Analysis, Preliminary Timing Analysis	
TaskOrderSpec	Syntactic Analysis, Preliminary Timing Analysis	
ArchPrioSpec	Syntactic Analysis, Preliminary Timing Analysis	

Conclusions

This paper describes our experience in building a partial ISO 26262 conformance argument for an industrial safety-critical system that we analysed using architecture-based model checking technique. The conformance argument fragment shows how we operationally interpreted one of the three objectives of software architecture verification (i.e., software architecture compliance with software safety requirements) and how the model checking evidence contributes to satisfying this objective. Moreover, analysis of the conformance argument illustrates where further evidence is needed to complement that provided by model checking.

References

[1] P. Bishop, and R. Bloomfield. “A Methodology for Safety Case Development”. Proceedings of the Sixth Safety-critical Systems Symposium, 1998.

[2] A. Johnsen, P. Pettersson, and K. Lundqvist. “An Architecture-Based Verification Technique for AADL Specifications”. Proceedings of the 5th European Conference on Software Architecture, pp. 105–113, 2011.

[3] ISO 26262:2011. Road Vehicles — Functional Safety. International Organization for Standardization, 2011.

[4] Defence Standard 00-56. Safety Management Requirements for Defence Systems, Issue 4, Part 1: Requirements. (U.K.) Ministry of Defence, June 2007.

[5] D. Zowghi and V. Gervasi. “On the Interplay Between Consistency, Completeness, and Correctness in Requirements Evolution,” Journal of Information and Software Technology, vol. 45, 2003.

[6] C. Baier and J.-P. Katoen. “Principles of Model Checking (Representation and Mind Series)”. MIT Press, 2008.

[7] SAE AS5506B. “Architecture Analysis & Design Language (AADL)”. Society of Automotive Engineers, Sept. 2012.

[8] P. Feiler, D. Gluch, and J. Hudak. “The Architecture Analysis & Design Language (AADL): An introduction”. Software Engineering Institute, Technical Report CMU/SEI-2006-TN-011, 2006.

[9] R. Varona, E. Villar, A.-I. Rodríguez, F. Ferrero, and E. Alaña. “Architectural Optimization & Design of Embedded Systems Based on AADL Performance Analysis”. American Journal of Computer Architecture, vol. 1, no. 2, pp. 21–36, 2012.

[10] T. P. Kelly and J. A. McDermid. “Safety Case Construction and Reuse Using Patterns”. Proceedings of the 16th International Conference on Computer Safety, Reliability, and Security (SAFECOMP), Sept. 1997.

[11] A. Johnsen, K. Lundqvist, P. Pettersson, and O. Jaradat. “Automated Verification of AADL-Specifications Using UPPAAL”. Proceedings of the 14th IEEE International Symposium on High-Assurance Systems Engineering (HASE), 25–27 Oct. 2012, pp. 130–138.

[12] J. Zhou, A. Johnsen, and K. Lundqvist. “Formal Execution Semantics for Asynchronous Constructs of AADL”. Proceedings of 5th International Workshop on Model Based Architecting and Construction of Embedded Systems, September 2012, pp. 43–48.

[13] G. Behrmann, A. David, and K. G. Larsen. “A tutorial on UPPAAL”. Department of Computer Science, Aalborg University, Denmark, Tech. Rep., 17 Nov. 2004.

[14] Weaver, R. A. and T. P. Kelly. “The Goal Structuring Notation - A Safety Argument Notation.” Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases, July 2004.

[15] I. Bate and A. Burns. “An Integrated Approach to Scheduling in Safety-Critical Embedded Control Systems", Real-Time Systems Journal, 25, (1), pp. 5-37, 2002.

Biographies

Omar T. Jaradat, Ph.D. student, School of Innovation, Design and Engineering, Mälardalen University, Högskoleplan 1, SE-72123, Västerås, Sweden, Tel: +46 21101369, Fax: +46 21101460 e-mail – omar.jaradat@mdh.se.

Omar Jaradat is a Ph.D. candidate in the Innovation, Design and Engineering department at Mälardalen University. His research interests include safety argumentation for safety critical systems, where the main focus is on modular safety cases and composable certification.

Patrick Graydon, Postdoctoral Research Fellow, School of Innovation, Design and Engineering, Mälardalen University, Högskoleplan 1, SE-72123, Västerås, Sweden, Tel: +46 21101421, Fax: +46 21101460 e-mail – patrick.graydon@mdh.se.

Dr Patrick Graydon is a postdoctoral Research Fellow at Mälardalen University. His research interests include safety and security argumentation, dependable software engineering, and certification processes. He is presently investigating component-based software engineering for critical systems.

Iain Bate, Senior Lecturer, Department of Computer Science, University of York, Heslington, York, YO10 5GH, UK, Tel: +44 (0) 1904 325572, Fax: +44 (0) 1904 325599, e-mail – iain.bate@york.ac.uk.

Dr Iain Bate is a Senior Lecturer in Real-Time Systems at the University of York and a Visiting Professor with Mälardalen University in Sweden. His research includes the design and certification of real-time embedded systems.