Combined Real-Time, Safety and Security Model Analysis

P Dissaux, Frank Singhoff, L Lemarchand, H. Tran, I Atchadam

Combined Real-Time, Safety and Security Model Analysis

P. Dissaux[1], F. Singhoff[2], L. Lemarchand[2], H.N. Tran[2], I. Atchadam[2]

1: Ellidiss Technologies, 24, quai de la douane, 29200 Brest, France
2: Lab-STICC, CNRS UMR 6285, Univ. of Brest, 20, av Le Gorgeu, 29200 Brest, France

Introduction

Model Driven Engineering (MDE) practices have been subject to many improvements since the last tens of years as much for modelling languages as model analysis and verification solutions.

In the context of critical systems, the role of early verification [14] appears to become one of the most challenging approaches to cope with the increase of size and complexity of embedded software.

One of the subsequent issues concerns the need to perform model verifications referring to several analysis domains that may lead to contradictory conclusions about the correctness of the model.

This paper describes a tool supporting practical experiment based on the use of an architecture description language and combining analysis techniques covering Real-Time, Safety and Security criteria to be used for identifying architectural trade-offs.

1. Multi-criteria model analysis

Even when it is not subject to validation by a certification authority, a characteristic of critical software is that it must comply with a set of additional requirements specifying its correct behaviour in terms of performance, safety and, more and more frequently, security. Other criteria such as power consumption or cost may also have to be considered.

However, it often occurs that improving safety or security objectives of a system may decrease its performances (and conversely), for instance by introducing additional software functions to support redundancy or data confidentiality.

Similarly, safety and security goals may sometimes lead to contradictory design choices. A trivial example is the door locking dilemma: keeping the door locked satisfies security requirements by preventing intrusions, whereas it also prevents firemen to access the building in case of a fire, which is a safety discrepancy.

Finally, compliance with other non-functional requirements in terms of power consumption weight or cost may also impact the design decisions.

It thus appears highly profitable to provide capabilities to achieve multi-criteria analysis at design time to define and justify the best trade-offs for the system architecture. Analysis methods that can drive design space exploration by computing trade-off have been proposed by the past. PAES multi-objective meta-heuristic is one of such method and [12][20] have shown how to apply it for design space exploration of critical software.

Nevertheless, the temptation is high to create a dedicated model of the same system that fits with each analysis domain. Such an approach may facilitate the use of each specialized analysis tool but complicates the elaboration of a multi-criteria trade-off.

2. Selected modelling and analysis technologies

In order to avoid the issue mentioned above, we decided to use a common architecture model with relevant annotations or sub-languages for each analysis domain.

Several solutions could be considered to achieve this goal. For the purpose of this work, we selected the Architecture Analysis and Design Language (AADL) for the modelling phase and a set of existing AADL compliant solutions for the model verification phase.

2.1. AADL

The AADL has been defined to describe software intensive real-time systems and to embed a sufficient level of semantics to enable the use of early analysis or production tools.

AADL is an international standard of the SAE, Aerospace Division, under reference AS-5506C, January 2017 [1].

An AADL model can be fully described by its textual representation, which makes it scalable and appropriate for version and configuration management.

The language standard is composed of a core and several optional annexes. The core language addresses the description of multi-threaded, distributed software architectures. Currently standardized annexes cover in particular Time and Space Partitioned architectures, Real-Time behaviour and Error modelling. Additionally, a Security annex is in preparation. In a more general way, specific model analysis annotations can be specified under the form of Property Sets.

Finally, more specific annexes can be defined. This has especially been done to introduce inline verification sub-languages such as REAL or RESOLUTE. Our experiment makes use of the LAMP annex, offering the processing power of Logic Model Processing (LMP) [8][10], while having the ability to be directly embedded inside the AADL specification.

2.2. Real-Time performance

To enable Real-Time performance analysis, the input model must embed enough information about concurrent software tasks and their interactions (software architecture) as well as some knowledge about the computing and communication resources provided by the hardware execution platform (hardware architecture).

All that can be expressed with AADL core language with an optional use of Behavior Annex descriptions that can act as pseudo-code to get a more accurate representation of the functional behaviour of the software, e.g. the best and worst case execution time (WCET) values that are usually considered for scheduling analysis.

During our experiment, we have considered three cases of timing analysis:
- Scheduling analysis provided by the Cheddar tool [3]. This computation is based on theoretical feasibility tests and static simulation. This analysis case fits well for a set of periodic tasks.
- Model execution provided by the Marzhin simulator [9]. This solution provides less deterministic results but can be applied to any kind of tasks set, including asynchronous interactions with the user.
- Scheduling Aware end to end Flow Latency Analysis (SAFLA) uses the concept of AADL End to End Flow and the threads Response Time computed by one of the two preceding methods (i.e. with Cheddar or Marzhin) as well as a LAMP verification program.

2.3. Safety

Whereas the Real-Time subset of the input model describes the nominal behaviour of the software, additional details about its erroneous behaviour must be provided to perform Safety analysis.

For that purpose, the AADL standard includes the Error Model annex (EMV2) that provides a Property Set, a library of predefined error models and an annex sub-language. This sub-language expresses how errors can be generated, propagated and trigger state changes in an error automaton.

The safety assessment process that we put in place consists in an evaluation of the Reliability of the system using Fault Tree Analysis (FTA) based on the information provided by the AADL model and its error annex. It operates as follows:
- Add error model annex to the existing AADL components of the architecture.
- Convert the AADL architecture into an OpenPSA model and launch the Arbre Analyste [4] specialized tool to perform Fault Tree Analysis and compute Safety indicator such as Mean Time Between Failures (MTBF) of the system.
- Optionally add redundancy components in the architecture to increase the safety indicator.

2.4. Cyber Security

According to the Common Criteria [7], a security analysis process should address in particular threats such as unauthorized disclosure, modification or loss of use of the software. We will thus attempt to maximize the level of Confidentiality, Integrity and Availability of our system to reduce these threats.

Interestingly, one possible method to evaluate Availability is to perform flow latency analysis for each critical service provided by the software and to check that they comply with the corresponding requirements. Practically, the solution presented in section 2.2 (SAFLA) can be used for this purpose.

Addressing Confidentiality and Integrity can be performed at least at two different levels. First level consists in making architectural design choices that intrinsically reduce the risk of unauthorized access to critical data. Amazingly, this mostly implies enforcing software engineering good practices, such as modularity, data hiding and low coupling principles that have been promoted during several tens of years, not always successfully. An example of solution that strongly enforces these principles is the Hierarchical Object-Oriented Design (HOOD) method [2] that has been used for the development of critical embedded software in many programs like the Rosetta probe, the Airbus aircrafts or the Eurofighter Typhoon. Moreover, it is possible to apply the HOOD method for AADL projects using the Stood for AADL tool [5].

However, it may be recommended to complement these architectural good practices with specific security rules analysis. The Security Annex for AADL is still being discussed by the standardization committee, so we applied the following verification approach:

- Use a dedicated AADL Property Set with relevant security wise information, such as trust level and data access rights [11], or the more abstract notion of confidentiality and integrity levels.
- Develop LAMP verification rules to check data access rights or implement Bell-La Padula and Biba rules checker algorithms [12] to evaluate Confidentiality and Integrity indicators in terms of number of security breaches for the overall software.
- Introduce additional components such as encryption devices or data filters to increase the global Security indicator.

2.5. Other non-functional requirements

As discussed above, possible solutions to improve Performance, Safety and Security levels consist in adding new hardware or software components to the original design. Such enhancements have a negative impact on general characteristics of the system like power consumption, weight and cost. These characteristics may contribute to the estimate of a global Feasibility indicator.

In the context of our experiment, these criteria can be managed in a simple way as follows:
- Introduce a dedicated AADL Property for each criterion and provide an estimate value to each unitary component.
- Perform a summation according to each architectural variant to get a global value for the global system. Such calculation can be easily implemented by a LAMP rule and provides what may be seen as a kind of Feasibility indicator.

A more sophisticated solution would be to estimate power consumption from processor load that can be computed as an outcome of scheduling analysis. This approach has not been applied during our experiment.

2.6. All in One: the AADL Inspector framework

The assurance case defined for our experiment is restricted to the four quality assurance factors and the corresponding assessment solutions that have been discussed in the preceding sections:

- Availability assessed by simulation (Marzhin, LAMP)
- Safety assessed by Fault Tree Analysis (Arbre Analyste)
- Security assessed by static analysis (LAMP)
- Feasibility assessed by static analysis (LAMP)

All these tools are available inside the standard AADL Inspector framework. AADL Inspector [6] can either parse native AADL models or translate foreign models into AADL and then connect them to a variety of verification and generation tools, such as Cheddar for scheduling analysis, Marzhin for event-based simulation, Arbre Analyste for Fault Tree Analysis and Ocarina [21] for source code generation.

All the processing features (model navigation, queries, constraints or transformations) are implemented with the LMP toolbox. LMP applies the benefits of Logic Programming to Model Driven Engineering. The main idea consists in defining any meta-model under the form of a set of Prolog facts and to process corresponding models with Prolog rules. LMP is especially appropriate to increase the tool capabilities at configuration level, such as implementing model transformations from AADL to remote analysis tools.

LAMP brings the benefits of LMP to the AADL user. The LAMP annex enables the end user to add his own AADL processing features inline at design time. It becomes thus possible to adjust the assurance case assessment for each project.

3. Case study design

This section introduces a simple case study developed in AADL for which the various analysis techniques presented above can be applied to evaluate quality assurance indicators in terms of Availability, Safety and Security. Feasibility aspects are not covered by the example, however corresponding design and analysis approaches would be quite similar to the ones used to deal with Security analysis.

The goal of this example is not to provide a generic solution to find the best architectural trade-off that optimizes all the quality assurance indicators, but much more modestly to show modelling and analysis solutions that may help to improve the engineering process of such systems.

3.1. Top level architecture

Proposed example consists of a classical control system composed of four subsystems deployed over a network (cf. Figure 1). This example does not represent any specific real case application. It has been invented for the sole purpose of illustration and demonstration.

This generic design includes sensors, a control unit and actuators as well as a monitoring dashboard. It

can for instance represent a room access control system, where sensors would be a pin code input keyboard, actuators would be the electrical door locking mechanism and the control unit would be the authentication module that is configured and monitored by an administration console represented by the dashboard.

The graphical AADL design has been done with the Stood for AADL tool and the textual notations have been automatically generated from this design model.

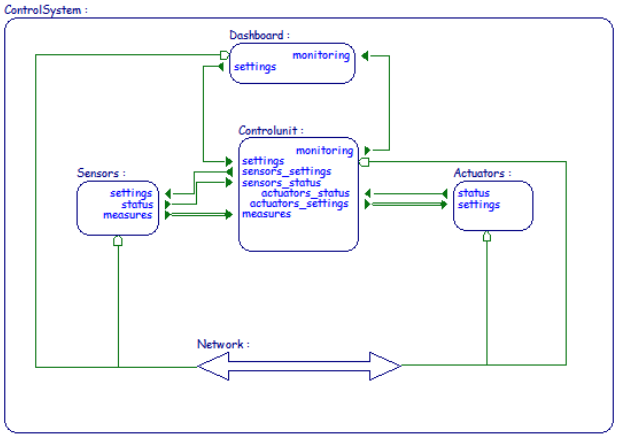

Figure 1: control system (top level)

In the corresponding textual notation (fragment), the main architectural description is shown in black colour. It includes the four subsystems and the network declared in the *subcomponents* section, the various point to point logical connections between the subcomponents and network access links in the *connections* sections and a dedicated property indicating that the logical connections are carried by the network.

```
SYSTEM IMPLEMENTATION ControlSystem.others
SUBCOMPONENTS
 Sensors:      SYSTEM Sensors.others;
 Controlunit:  SYSTEM Controlunit.others;
 Actuators:    SYSTEM Actuators.others;
 Dashboard:    SYSTEM Dashboard.others;
 Network:      BUS Network;
CONNECTIONS
 cnx1:  PORT Dashboard.settings -> …
 cnx2:  PORT Controlunit.monitoring -> …
 cnx3:  PORT Controlunit.sensors_settings -> …
 cnx4:  PORT Sensors.status -> …
 cnx5:  PORT Sensors.measures -> …
 cnx6:  PORT Controlunit.actuators_settings -> …
 cnx7:  PORT Actuators.status -> …
 cnx8:  BUS ACCESS Network -> Dashboard.Nwk;
 cnx9:  BUS ACCESS Network -> Sensors.Nwk;
 cnx10: BUS ACCESS Network -> Actuators.Nwk;
 cnx11: BUS ACCESS Network -> Controlunit.Nwk;
FLOWS
 f1: END TO END FLOW
  Sensors.f1 ->
  cnx5 -> Controlunit.f1 -> cnx6
  -> Actuators.f1;
PROPERTIES
 Actual_Connection_Binding =>
  (reference(Network))
  applies to cnx1,cnx2,cnx3,cnx4,cnx5,cnx6,cnx7;
 Timing => Immediate
  applies to cnx5,cnx6;
ANNEX EMV2 {**
 use behavior errorlibrary::failstop;
 composite error behavior
 states
  [ Dashboard.FailStop or
    Sensors.FailStop or
    ControlUnit.FailStop or
    Actuators.FailStop or
    Network.FailStop ]-> FailStop;
 end composite;
**};
END ControlSystem.others;
```

Additional information has been added to the top level architectural information to support richer model analysis as introduced in section 1.

Firstly, an end-to-end Flow declaration as well as timing Properties for Connections are shown in blue colour and can be used to support timing analysis and compute the latency between a sensor measure to the corresponding actuator reaction.

Secondly, an Error annex section defines how the main system may fail. In our case, it consists of a disjunction of subsystems or network failures. This will take part to the elaboration of the fault tree for safety analysis (FTA). The AADL Error Modeling annex (EMV2) can also be used for other aspects of safety engineering such as Functional Hazard Assessments (FHA), Failure Mode and Effect Analysis (FMEA), and Common Cause Analysis (CCA). However, only Fault Tree Analysis is considered in this experiment.

Lastly, a very simplified security model has been used here by adding dedicated Properties to associate exchanged data types with a given Security level. In this example, we consider that the measurement data may contain sensitive information and must thus be protected, whereas monitoring data may remain unprotected. More acurate security models could include other Properties such as data access rights that may differ for the various software functions of the system (access groups) [11].

Data types are specified here in a separate Package and can be referenced by Data ports and Data components of the main design. Security level properties have been added for the purpose of model analysis. The higher the value of this property is, the higher the confidentiality and integrity of corresponding Data ports or Data components is.

```
PACKAGE ControlSystemTypes
PUBLIC

  DATA T_settings
    -- Medium security level
    LAMP::Security_Level => 3;
  END T_settings;
```

```
  DATA T_status
    -- Low security level
    LAMP::Security_Level => 2;
  END T status;

  DATA T_measures
  PROPERTIES
    -- High security level
    LAMP::Security Level => 5;
  END T_measures;

  DATA T_monitoring
  PROPERTIES
    -- Low security level
    LAMP::Security Level => 2;
  END T_monitoring;

END ControlSystemTypes;
```

Each of the subsystems can now be described in a similar way. We will only provide design details for one of them.

3.2. Sensors subsystem

The Sensors subsystem (cf. Figure 2) represents a separate node of the network. It is composed of hardware electronics such as a Processor, local Memory, Bus, a Device component representing the sensors themselves, and an acquisition software application depicted by an AADL Process that is running on the Processor.

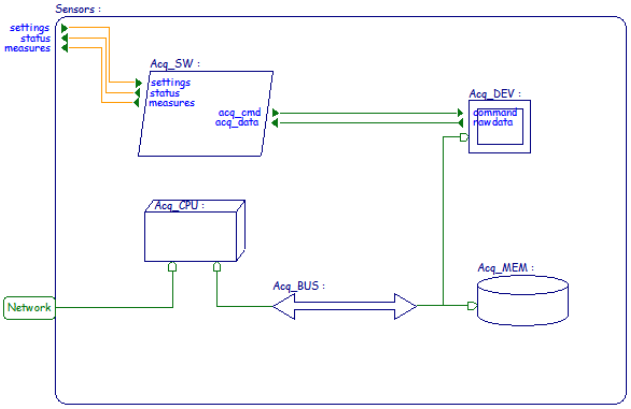

Figure 2: sensors subsystem

In a similar way as for the encompassing system, the architecture description of the Sensors subsystem can be represented textually by the following AADL fragment and additional Flows and Error annex sections can be added for the purpose of analysis.

```
SYSTEM IMPLEMENTATION Sensors.others
SUBCOMPONENTS
 Acq_CPU: PROCESSOR Acq_CPU;
 Acq_MEM: MEMORY Acq_MEM;
 Acq_SW:  PROCESS Acq_SW.others;
 Acq_DEV: DEVICE Acq_DEV;
 Acq_BUS: BUS Acq_BUS;
CONNECTIONS
 cnx1: PORT settings -> Acq_SW.settings;
 cnx2: PORT Acq_SW.status -> status;
 cnx3: PORT Acq_SW.measures -> measures;
 cnx4: PORT Acq_SW.acq_cmd -> Acq_DEV.command;
 cnx5: PORT Acq_DEV.rawdata -> …
 cnx6: BUS ACCESS Network -> Acq_CPU.Nwk;
 cnx7: BUS ACCESS Acq_BUS -> Acq_CPU.Acq_BUS;
 cnx8: BUS ACCESS Acq BUS -> Acq DEV.Acq BUS;
 cnx9: BUS ACCESS Acq_BUS -> Acq_MEM.Acq_BUS;
FLOWS
 f1: FLOW SOURCE
  Acq_DEV.f1 ->
  cnx5 -> Acq SW.f1 -> cnx3
  -> measures;
PROPERTIES
 Actual_Processor_Binding =>
  (reference(Acq_CPU))
  applies to Acq_SW;
ANNEX EMV2 {**
 use behavior errorlibrary::failstop;
 composite error behavior
 states
  [ Acq_CPU.FailStop or
    Acq_DEV.FailStop or
    Acq BUS.FailStop ]-> FailStop;
 end composite;
**};
END Sensors.others;
```

This description process must be performed hierarchically until we reach the lowest level components. In order not to complexify our example too much, we stopped at AADL Thread level.

Textual notation fragments of the Sensors subsystem subcomponents is shown below with their corresponding Flows and Error annex details.

The detailed description of the other subsystems has been volontarily omitted in this paper due to the lack of space. However, they would be quite similar.

3.3. Sensors subsystem hardware

The hardware parts that are considered for the Sensors subsystem are a computing resource represented by an AADL Processor and a raw data acquisition set represented by an AADL Device.

```
PROCESSOR Acq_CPU
FEATURES
 …
PROPERTIES
 Scheduling_Protocol =>
  (RATE_MONOTONIC_PROTOCOL);
ANNEX EMV2 {**
 use behavior errorlibrary::failstop;
 properties
  EMV2::OccurrenceDistribution =>
   [ ProbabilityValue => 1.0e-5;
     Distribution => Poisson; ]
   applies to Failure;
**};
END Acq_CPU;
```

The Processor component is characterized by a Property specifying the scheduling policy of its Operating System and an Error Model annex giving its probability of failure. Note that this Error Model makes use of a predefined error automaton defined in a library.

```
DEVICE Acq_DEV
FEATURES
 …
FLOWS
 f1: FLOW SOURCE rawdata;
ANNEX EMV2 {**
 use types errorlibrary;
 use behavior errorlibrary::failstop;
 error propagations
  rawdata : out propagation {NoValue};
 end propagations;
 component error behavior
 propagations
  p1 : FailStop -[]-> rawdata{NoValue};
 end component;
 properties
  EMV2::OccurrenceDistribution =>
   [ ProbabilityValue => 1.0e-3;
     Distribution => Poisson; ]
  applies to Failure;
**};
END Acq_DEV;
```

The Device component declaration indicates that it contributes as the source of the end to end Flow through its Port *rawdata* and contains an Error Model annex specifying its probability of failure and the consequence such failures have to the data flow.

3.4. Sensors subsystem software

The Sensors software that is running on the Processor is represented by an AADL Process containing a single AADL Thread.

```
PROCESS IMPLEMENTATION Acq_SW.others
SUBCOMPONENTS
 Acq_Driver : THREAD Acq_Driver;
CONNECTIONS
 cnx1: PORT settings -> Acq_Driver.settings;
 cnx2: PORT Acq_Driver.status -> status;
 cnx3: PORT Acq_Driver.measures -> measures;
 cnx4: PORT Acq_Driver.acq_cmd -> acq_cmd;
 cnx5: PORT acq_data -> Acq_Driver.acq_data;
FLOWS
 f1: FLOW PATH
  acq_data ->
  cnx5 -> Acq_Driver.f1 -> cnx3
  -> measures;
ANNEX EMV2 {**
 use types errorlibrary;
 use behavior errorlibrary::failstop;
 error propagations
  acq_data : in propagation {NoValue};
  measures : out propagation {NoValue};
 end propagations;
 component error behavior
 transitions
  t1: Operational
   -[ acq_data{NoValue} ]-> FailStop;
 propagations
  p1: FailStop -[]-> measures{NoValue};
 end component;
**};
END Acq_SW.others;
```

The Process component behaves as a gateway for the end to end flow (AADL Flow Path) and for the Error Model (AADL Error Propagation).

```
THREAD Acq_Driver
FEATURES
 …
FLOWS
 f1: FLOW PATH acq_data -> measures;
PROPERTIES
 Dispatch_Protocol => Periodic;
 Compute_Execution_Time => 5ms..5ms;
 Deadline => 100ms;
 Period => 100ms;
END Acq_Driver;
```

The Thread component is characterized by its real-time attributes. Note that it would have been possible to specify a more precise execution behaviour by adding Subprogram subcomponents and Behavior annexes [18].

4. Case study analysis

In this section, we describe how the architectural design of the example, associated with the various specialized properties and annexes add-ons can be used to perform combined performance, safety and security analysis thanks to the advanced features provided by the AADL Inspector tool. However, as this architectural design has been fully exported as a standard AADL model, any other AADL compliant analysis tool could be used instead.

4.1. Performance

In the scope of this case study, we have put the focus on the Scheduling Aware end to end Flow Latency Analysis (SAFLA) technique. This technique consists in computing an estimate of the maximum end to end reaction time between the ultimate source and the ultimate sink of a data flow across the overall system. In our case, the ultimate flow source is the *rawdata* output port of the Sensors Device and the ultimate sink is the *command* input port of the Actuators Device.

Between these two ends, the dataflow traverses several threads and connections, each of them contributing to the global end to end latency.

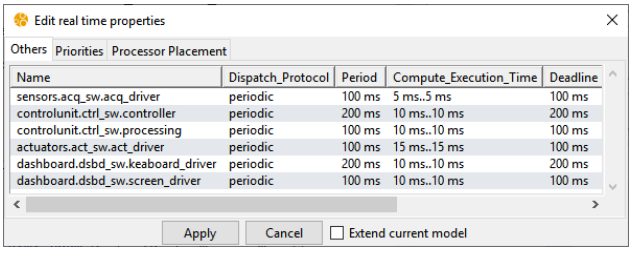

Figure 3: Threads real-time attributes

Although each Thread is given a set of real-time attributes such as its period and Worst-Case Execution Time (WCET), its actual contribution to an end to end Flow latency is its response time that

takes into accounts scheduling policies, Threads dependencies and interferences.

Regarding Connections, we make the assumption that only those that are spread over the network will add a significant delay to the end to end Flow. We can then estimate each Bus message response time in a similar way we do it for Threads.

Several tools can be used to compute the actual maximum response time for Threads and Bus messages. For our experiment, we used the Marzhin AADL simulator [9] for that purpose (cf. Figure 4).

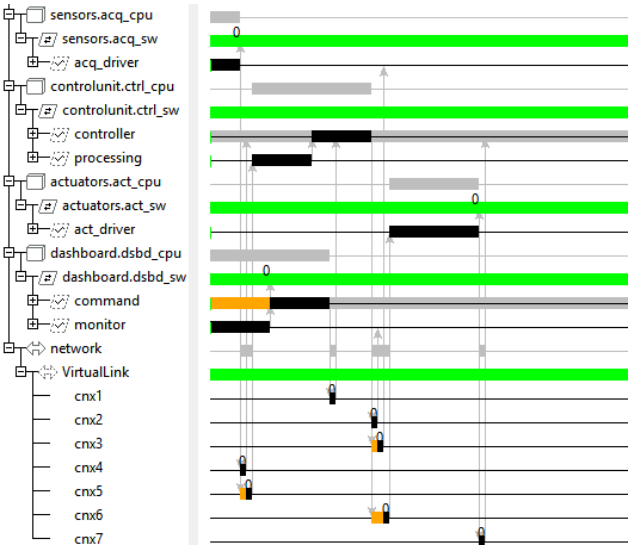

Figure 4: AADL real-time simulation

Finally, a LAMP rule is used to collect all the required timing information from the simulator and compute the global end to end Flow latency. The result of this performance analysis process is shown in the AADL Inspector console (cf. Figure 5).

```
Static Analysis | LAMP Analysis | Timing Analysis | Safety Analysis | Code Generation | Doc Generation | Scripts

*** LAMP: AADL model predicates loaded.
*** LAMP: response time predicates loaded.
*** LAMP: simulation events predicates loaded.
*** LAMP: library rules loaded.
*** LAMP: goal rules loaded.
*** LAMP: execution started.

FLOW LATENCY ANALYSIS
- Max Response Time of Thread root.sensors.acq_sw.acq_driver: 5 ms
- Max Response Time of Connection root.cnx5: 2 ms
- Max Response Time of Thread root.controlunit.ctrl_sw.processing: 10 ms
- Max Response Time of Thread root.controlunit.ctrl_sw.controller: 10 ms
- Max Response Time of Connection root.cnx6: 2 ms
- Max Response Time of Thread root.actuators.act_sw.act_driver: 15 ms
=> Maximum Latency of End to End Flow root.fl: 44 ms
```

Figure 5: Flow latency analysis with LAMP

The end to end Flow latency can then be compared with the software requirements in terms of time of reaction between the Sensors and the Actuators.

4.2. Safety

The performance analysis presented above concerns the nominal behavior of the software. In our experiment, we use safety analysis technique to evaluate the Mean Time Between Failures of our system.

The AADL Error Model statements that are disseminated within the various component descriptions can be compiled together to generate a fault tree. Such a fault tree is composed of Or and And gates as well of Basic Events with a probability of occurrence.

AADL Inspector uses a LMP model transformation [8][10] to convert the AADL architecture and its Error Models into a Open PSA standard format file [19]. This file is then automatically loaded into the Arbre Analyste tool [4] to show the corresponding fault tree graphically and compute the MTBF value (cf. Figures 6 and 7).

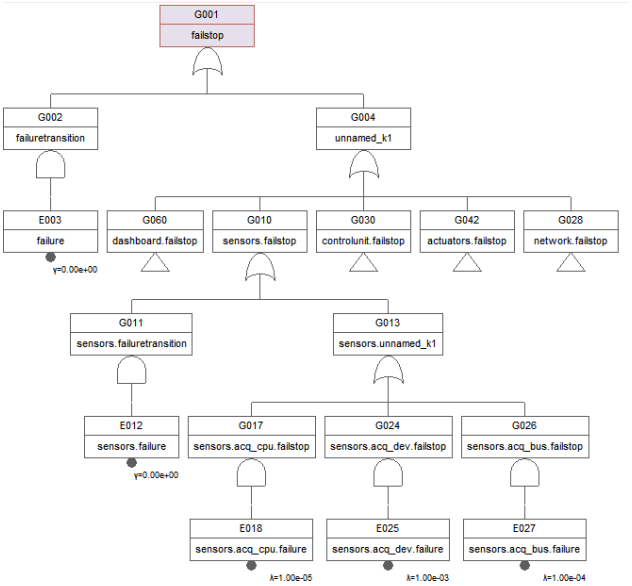

Figure 6: Fault tree in Arbre Analyste

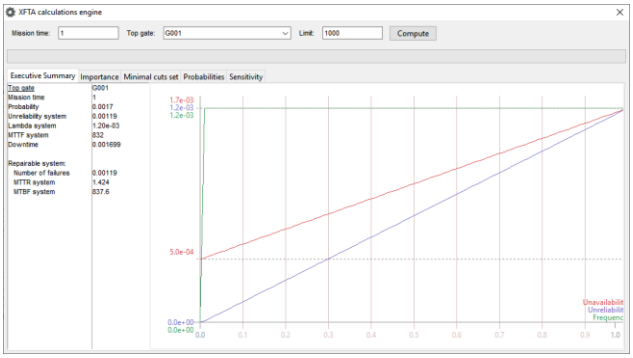

Figure 7: MTBF computation in Arbre Analyste

In case the obtained MTBF values do not fit the requirements, the software architecture may be revised by introducing redundancies for instance.

4.3. Security

The goal of our security analysis is focused on detection of possible unauthorized access to sensitive data. Our design example has been produced using the Stood for AADL tool which enforces data hiding principles as part of its HOOD modeling rules compliancy. With that way to use this tool, it is not possible to generate direct external access to data components (i.e. AADL provides Data Access features). This restriction reduces significantly security breaches by construction.

However, it remains possible to access internal data in a controlled way through AADL Ports or Subprograms. It is thus necessary to verify that these indirect data access points are secure enough. For our simple case study, we have considered a very basic security model where a dedicated Property Set is used to associate a security level (integer value) to AADL Data (cf. section 3.1), as defined above:

```
PROPERTY SET LAMP IS
-- …
Security_Level : AADLINTEGER APPLIES TO
  (Data, Data Access, Port, Parameter);
-- …
END LAMP;
```

Note that this Property Set will be replaced by the one defined by the future AADL Security Annex standard when ready.

For our experiment, we expect to use these security level Properties to detect potential data confidentiality breaches. The first rule that can easily be checked is that all the ports contributing to a same end to end Flow are at the same security level. The second rule refers to the Bell-La Padula algorithm [16]. This algorithm is based on the verification of two rules denoted No-Read-Up and No-Write-Down.

No-Read-Up refers to the fact that a component at a given security level cannot read data that is tagged with a higher security level whereas No-Write-Down means that a component tagged with a given security level cannot write information to a lower security level.

Note that similar analysis can be performed for data integrity check, using the Biba algorithm [17] and its No-Read-Down and No-Write-Up rules.

Security rules policy may vary from one project to another. It is thus interesting to provide a flexible way to customize them in the scope of a given design. With the tools we are using for this experiment, these rules can be easily implemented in Prolog language within a LAMP annex attached to the AADL model.

An attempt to define a security rules policy for our example could be:

- *Sec_R1*: All components involved in a same end to end Flow must be at the same security level.
- *Sec_R2*: The security level of a component is the higher security level value associated with its Data ports.
- *Sec_R3*: When two components are connected via a shared Bus, they must comply with the No-Read-Up and No-Write-Down rules.

A fragment of the implementation of these rules in prolog using an AADL LAMP annex is given below:

```
PACKAGE ControlSystemAnalysis
PUBLIC

ANNEX LAMP {**
/* rule Sec_R1 */
checkFlowSecurity :-
 getRoot(R), getClassifier(R,P,T,I),
 getAncestorRec(P,T,I,Q,U,J),
 isFlowImplementation('END TO END',Q,U,J,E),
 concat('root.',E,F),
 getEndToEndFlow('root',E,M),
 getFlowSecurityLevels(M,[],L,0,N), N > 1,
 printMessageSec_R1(F,L).
checkFlowSecurity :- nl.

/* rule Sec_R2 */
checkMaxSecurityLevel :-
 getMaxSecurityLevel(X,L),
 printMessageSec_R2(X,L).
checkMaxSecurityLevel :- nl.

/* rule Sec_R3 */
checkNoWriteDown :-
 isAADLBusBinding(_,C,_),
 isAADLConnection(_,P,T,I,_,_,_,C,_,_,_,_),
 getConnectionEnds(P,T,I,C,Xs,Xd),
 getMaxSecurityLevel(Xs,Ls),
 getMaxSecurityLevel(Xd,Ld),
 Ls > Ld,
 printMessageSec_R3(C,Ls,Ld).
checkNoWriteDown :- nl.

-- …
END ControlSystemAnalysis;
```

After running the LAMP checker, the output of the verification process is displayed in the AADL Inspector console as shown in Figure 8.

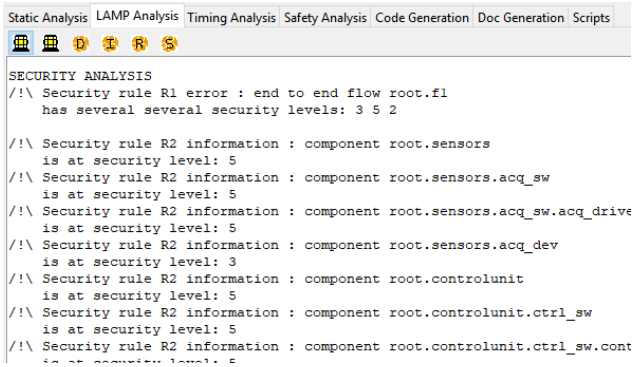

Figure 8: Security analysis with LAMP

Detected issues can be solved by modifying Property values or doing deeper changes in the architecture, e.g. adding new components to comply with specified security policy.

Conclusion

This paper has shown how it is possible to perform combined multi-criteria analysis to estimate real-time Performance, Safety and Security indicators for a same input model expressed in AADL and using the AADL Inspector framework. The goal is to help design teams to find the best architectural trade-offs for their critical embedded software.

As these criteria not only impact non-functional attributes related to each analysis domain but also the global design choices, using a common architecture description language brings significant benefit. Similar conclusions have been obtained by other initiatives, such as with the Architecture Centric Virtual Integration Process (ACVIP) [13].

The experiment presented in this paper has been applied to a small case study for demonstration purpose only. The choice of the quality assurance indicators and the way they are estimated must be studied deeply in the context of a real-life industrial project. Furthermore, we currently investigate the use of the PAES meta-heuristic to help designer to automatically run similar design space exploration on security and real-time performances with similar architecture models [12][20].

The AADL example that is used in this paper and the various verification results that are presented have been developed with the Stood and AADL Inspector tools. This example is available as part of the distribution package of these tools.

References

[1] AADL: Architecture Analysis and Design Language
http://www.aadl.info/
[2] HOOD: Hierarchical Object-Oriented Design:
http://www.esa.int/TEC/Software_engineering_and_standardisation/TECKLAUXBQE_0.html
[3]Cheddar: a flexible real-time scheduling framework. F. Singhoff, J. Legrand, L. Nana, L. Marcé, ACM SIGAda Ada Letters. Vol. 24. No. 4. ACM, 2004.
[4] Arbre Analyste:
https://www.arbre-analyste.fr/en.html
[5] Stood for AADL:
http://www.ellidiss.fr/public/wiki/wiki/stood
[6] AADL Inspector:
http://www.ellidiss.fr/public/wiki/wiki/inspector
[7] Common Criteria for Information Technology Security Evaluation:
https://www.commoncriteriaportal.org/cc/
[8] Model Verification: Return of Experience, P. Dissaux and P. Farail, 7th European Congress on Embedded Real Time Software and Systems (ERTS 2014), Toulouse.
[9] The SMART Project: Multi-Agent Scheduling Simulation of Real-time Architectures, P. Dissaux, O. Marc and all, 7th European Congress on Embedded Real Time Software and Systems (ERTS 2014), Toulouse.
[10] Merging and Processing Heterogeneous Models, P. Dissaux and B. Hall, 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016), Toulouse.
[11] Extending AADL for Security Design Assurance of Cyber-Physical Systems, R. Ellison, A. Householder and all, Technical Report CMU/SEI-2015-TR-014
[12] Combined security and schedulability analysis for MILS real-time critical architectures, I. Atchadam, F. Singhoff, H. N. Tran, N. Bouzid and L. Lemarchand, in 4th international workshop on Security and Dependability of Critical Embedded Real-Time Systems/CERTS, Stuttgart, Germany, 2019.
[13] Joint Common Architecture (JCA) Demonstration Architecture Centric Virtual Integration Process (ACVIP) Shadow Effort, A. Boydston, P. Feiler and all. In AHS 71st Annual Forum. 2015.
https://www.army.mil/e2/c/downloads/414601.pdf
[14] National Institute of Standards and Technology. The Economic Impacts of Inadequate Infrastructure for Software Testing, NIST Planning report 02-3, May 2002.
http://www.nist.gov/director/prog-ofc/report02-3.pdf
[15] Model-Based Verification of Security and Non-Functional Behavior using AADL, J. Hansson, B. Lewis, J. Hugues, L. Wrage, P. Feiler and J. Morley, in IEEE Security & Privacy, 2009.

[16] Secure computer system: Unified exposition and multics interpretation, D. E. Bell and L. J. La Padula. Technical report, MITRE CORP BEDFORD MA, 1976.
[17] Integrity considerations for secure computer systems, K. J. Biba, Technical report No. MTR-3153-REV-1). MITRE CORP BEDFORD MA, 1977.
[18] Virtual Execution of Real Time Software Architecture Models, P. Dissaux, SAE Technical Paper 2015-01-2530, 2015
[19] The Open PSA initiative:
http://www.open-psa.org/
[20] Multi-Objective Design Exploration Approach for Ravenscar Real-time Systems. R. Bouaziz, L. Lemarchand, F. Singhoff, B. Zalila, M. Jmaiel. Real-Time Systems, Springer Verlag, 2018, 54 (2), pp 424-483.
[21] From the prototype to the final embedded system using the Ocarina AADL tool suite. J. Hugues, B. Zalila, L. Pautet and F. Kordon, (2008). ACM Transactions on Embedded Computing Systems (TECS), 7(4), 42.