A Method to Formally Evaluate Safety Case Arguments against a System Architecture Model

Stefan Björnander and Rikard Land
System Safety
CrossControl AB
Västerås, Sweden
{stefan.bjornander,rikard.land}
@crosscontrol.com

Patrick Graydon and Kristina Lundqvist
School of Innovation, Design, and Technology
Mälardalen University
Västerås, Sweden
{patrick.graydon,kristina.lundqvist}
@mdh.se

Philippa Conmy
Department of Computer Science
University of York
York, Great Britain
philippa.conmy
@york.ac.uk

***Abstract*—For a large and complex safety-critical system, where safety is ensured by a strict control over many properties, the safety information is structured into a *safety case*. As a small change to the system design may potentially affect a large section of the safety argumentation, a systematic method for evaluating the impact of system changes on the safety argumentation would be valuable.**

We have chosen two of the most common notations: the Goal Structuring Notation (GSN) for the safety argumentation and the Architecture Analysis and Design Language (AADL) for the system architecture model. In this paper, we address the problem of impact analysis by introducing the GSN and AADL Graph Evaluation (GAGE) method that maps safety argumentation structure against system architecture, which is also a prerequisite for successful composition of modular safety cases.

In order to validate the method, we have implemented the GAGE tool that supports the mapping between the GSN and AADL notations and highlight changes in impact on the argumentation.

***Index Terms*—Safety Argumentation; GSN; AADL**

I. Introduction

For a large complex industrial safety-critical system, the safety is ensured by a strict control over many properties, some related to management and processes, others to the technical characteristics of the system and its components. All this information is structured into a *safety case*, see e.g. [1]. The sheer size of the information to be included in the safety case requires it to be structured in a systematic way. A small change to the system design during development, or a change requested after the system has been put in operation may potentially affect a large section of the safety argumentation. A systematic method for evaluating the impact of system changes on the safety argumentation would be very valuable.

When tracing potential changes in the system onto the safety argumentation, a number of questions arise. Will the safety claims still hold or will the system violate previous premises? How much reverification and revalidation is required?

In this paper, we present the GAGE method, which we have developed recently. It helps the safety engineer to answer these, and related questions, by mapping elements of a *system model* to the affected parts of the corresponding *safety arguments* (GAGE stands for GSN and AADL Graph Evaluation).

There are many ways to organize and present a safety case. In its simplest form, it can be written in natural language using a word processor. A more methodical approach is to use a semi-formal structure, such as the Claims, Arguments and Evidence (CAE) notation, which is a simple yet effective notation, or the Goal Structuring Notation (GSN), which is a graphical notation. We have chosen to use GSN for defining the safety case. In GSN, claims and evidences are termed goals and solutions, respectively.

We have chosen AADL for modeling the system architecture. AADL is commonly used in industry, e.g., in the fields of avionics and automobile. However, the general idea is applicable to other modeling languages, e.g., UML [2].

The contributions of this paper are: (1) the GAGE method, which maps a GSN safety argument to an AADL model of the system architecture; and (2) the GAGE tool, which assesses consistency between the argument and architectural model.

The rest of this paper is organized as follows: in Section II, we give some background information regarding GSN and AADL. Section III explains the GAGE method, and Section IV discusses its validity. In Section V, we discuss formalization of different kinds of safety arguments, and the paper is concluding with related work (Section VI) and conclusions and further work (Section VII).

II. Background

This section gives some background information about GSN and AADL. Our intention is that the description and examples provided will give sufficient information to understand the context of our research and our method. For full reference, see [3] and [4].

A. The Goal Structuring Notation

GSN is a notation designed for organizing and communicating a safety case by providing an *argument structure* with the purpose of convincing the reader that the system is reasonably safe. One important point of GSN is that the structure only states that the system is *safe enough*, absolute safety is regarded an unobtainable goal. Another important point is that the *safety context* of the system always must be defined, since context-free safety is impossible to argue.

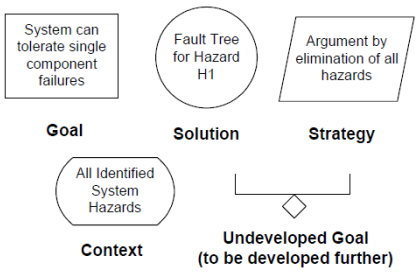

Figure 1: The elements of a GSN safety case as illustrated by Kelly and Weaver [4].

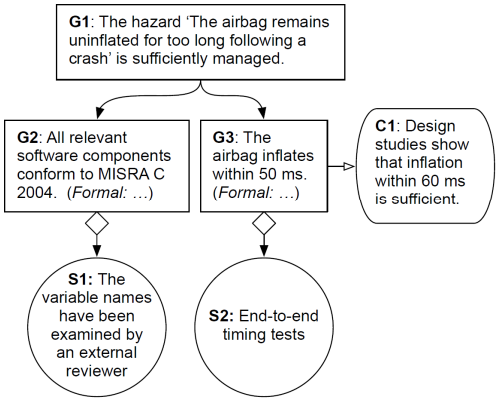

Figure 2: A GSN safety case involving an airbag system (the formal argument specifications have been abbreviated and are stated in Listing 2).

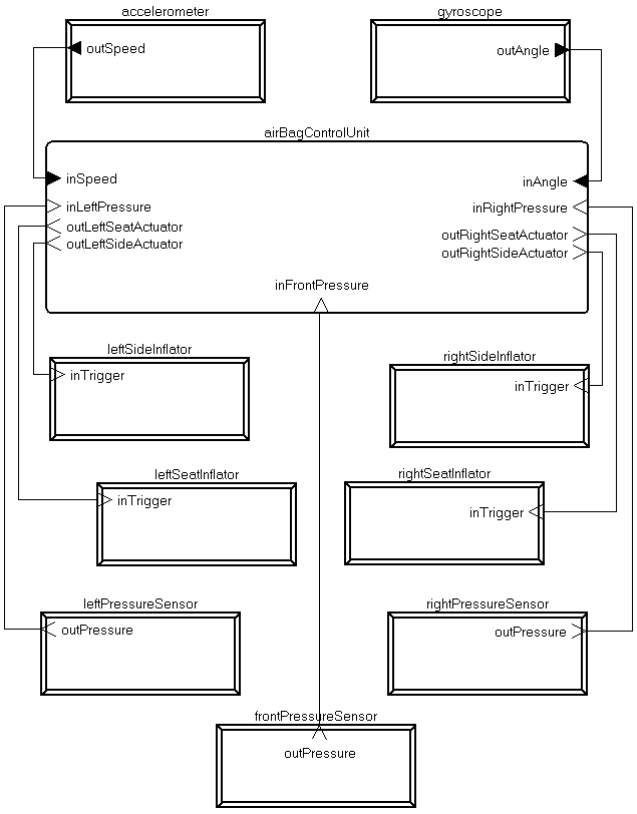

Figure 3: System architecture of an airbag control system.

GSN represents the individual elements of the safety case; that is, *strategies*, *goals*, *solutions*, and *contexts*. It also shows the relationship between evidence and safety goals; that is, how strategies are supported by specific goals, how goals are supported by solutions, and the contexts defined for the goals. Figure 1 illustrates the graphical notation of the elements.

Figure 2 illustrates part of an example GSN safety argument that illustrates why developers believe that their *mitigation* of a *hazard* is *adequate*. In this case, the airbag remains uninflated for too long following a crash. Figure 2 includes three goals expressed in natural language. Of these, goals G2 and G3 are also expressed formally in terms of the system's AADL model. They are more closely described in Section IV.

The safety case of Figure 2 illustrates only part of a real complete safety case. Some of the goals have been omitted due to space limitations.

B. Architecture Analysis and Design Language

AADL is an Architecture Description Language (ADL) intended for the design of both the system hardware and software. The component abstractions of AADL are separated into three categories: application software (thread, thread group, process, data, and subprogram), execution platform (processor, memory, device, and bus), and the system component, which allows systems to include other systems as well as software or hardware components.

Components communicate with each other through ports and it is possible to define physical port-to-port connections. Component definitions are divided into component types holding the public (visible to other components) features, and component implementations that define the private parts of the component and can hold subcomponents that are instances of other components (equivalent to classes and objects in object-oriented languages).

Figure 3 illustrates an example of an airbag control system. The system consists of three pressure sensors placed at the front and the two sides of the car, four inflators for airbags in the front seats and at the sides. There are also connections to the accelerometer and a gyroscope. However, the central part of the system is the airbag control unit, which receives information from the pressure sensors and notifies the airbag inflators in case of a collision.

III. The GAGE Method

We have developed the GAGE method that parses and maps the safety case against the system architecture. The basic idea is to bridge the gap between the safety case and the system architecture by traversing the safety arguments and evaluating them against the properties of the components of the system. The safety case and the system architecture are orthogonal, one GSN element might correspond to many AADL elements and vice-versa. Thus, it is difficult to isolate one part of the safety argument structure and compare it to one part of the

system. Instead, our approach is to evaluate the whole safety case in order to find areas of argument that are inconsistent with the AADL model.

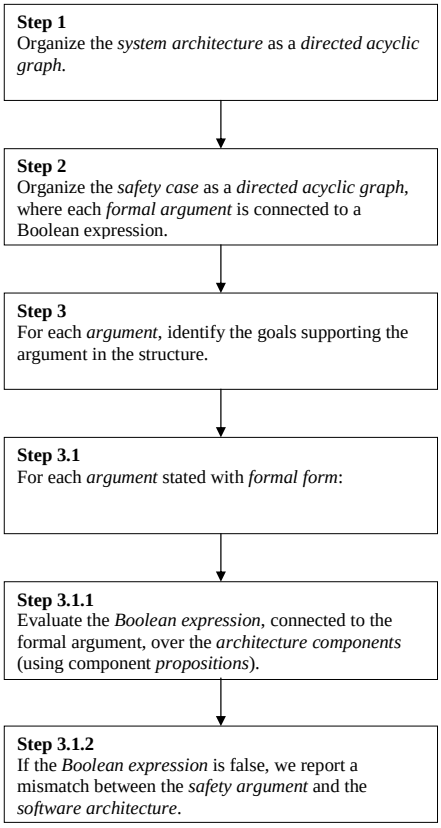

Figure 4: A Flow Chart of the GAGE Method.

The GAGE method is illustrated in Figure 4. Step 3 (with sub-steps) is a deterministic step-by-step procedure suitable for automated execution.

IV. Validation

In order to validate our approach we have developed the GAGE tool, which we have tested on the airbag example of Section II-B (Figure 3). As a next step, we plan to apply the method to an industrial safety-critical system. The three stages of validation, tool implementation, application to an example, and application to an industrial system, are described in the three following subsections.

A. Tool Support

The GAGE tool evaluates a safety case against a system architecture in accordance with the GAGE method. The input is a GSN safety case and an AADL system architecture model. The tool is written in Java and reads source code defined in XML. An AADL model can be stored in three different file formats: plain source code (.aadl), source code in XML format (.aaxl), and source code in XML with graphical annotations (.aaxldi). The tool reads the source code in XML format. The tool implements the GAGE method of Section III.

Listing 1 The AADL model definition of a pressure sensor as part of an airbag system.

```
device PressureSensor
  features
    outPressure: out event port;
  properties
    Info => "ReactionTime=5";
    Info => "MisraC=true";
end PressureSensor;
```

Each AADL component can be annotated with one or several properties named *Info* holding a name-value pair. For instance, the pressure sensor (which is part of the airbag definition of Section IV-B) defined in Listing 1 hold the properties *ReactionTime* and *MisraC*. For each formal goal, the tool traverses the subcomponents and evaluates the value of the Boolean expression by inspecting the property values.

B. The Airbag Example Revisited

In this section, we take another look at the airbag example of Section II-B and Figure 3. It can be argued that an airbag system constitutes a safety-critical system, since if it becomes inflated too late (or not at all) a person runs the risk of serious injuries. On the other hand, if it does not inflate as expected, the situation does not become worse, compared with a non-present airbag in the first place.

Moreover, the airbag may cause damage if it inflates at the wrong time. For instance, if the driver seat airbag becomes inflated when the car is driven at high speed, it may cause a serious traffic accident. The AADL source code of the airbag system is given in Listing 4.

In order for the airbag to work properly, it has to become inflated within 50 milliseconds after one of the pressure sensors detects a collision. This condition (which is stated as goal G2 in the safety case of Figure 2) can formally be stated in the summary attribute in the XML-code of the safety case, see Listing 2. Since less-than (<) and greater-than (>) characters are not allowed in XML elements, we have instead used the FORTRAN relational operators ('.le.' instead of '<=').

In the claim, we want to make sure the total reaction time for the airbag system is less-than or equal to 50 milliseconds. This can be viewed as a graph searching problem; we want to find the longest path (corresponding to the maximum reaction time) from any of the pressure sensors to any of the airbag inflators. We can also make sure the software of each component has been developed in compliance with the MISRA-C standard [5] (goal G3 in Figure 2).

The function $path\_set(start\_set, end\_set)$ returns the set of paths (lists of components) from any component in the start set to any component in the end set. The function $evaluate(path\_set, function, property)$ applies the given function (sum and and in Listing 2) and the given property value ($ReactionTime$ and $MicraC$ in Listing 2) on each path, and returns a set of resulting values (one value for each path). The function $sum(value\_set)$ return the sum of all values in the set, $max(value\_set)$ returns the largest value

Listing 2 A formal GSN safety case as input to the GAGE tool.

```
<goal name="G3" formal=
      "max(evaluate(path_set(set(leftPressureSensor,
                                 frontPressureSensor,
                                 rightPressureSensor),
                             set(leftSeatInflator,
                                 rightSeatInflator,
                                 leftSideInflator,
                                 rightSideInflator)),
           sum, ReactionTime)).le.50"/>
<goal name="G2" formal=
      "and(evaluate(path_set(set(leftPressureSensor,
                                 frontPressureSensor,
                                 rightPressureSensor),
                             set(leftSeatInflator,
                                 rightSeatInflator,
                                 leftSideInflator,
                                 rightSideInflator)),
           and, MisraC))"/>
```

Listing 3 The result of the GAGE tool execution.

```
Goal G2: Conditional expression satisfied.
Goal G3: Conditional expression satisfied.
```

of the set, $and(value\_set)$ returns true if all values in the set are true[1].

The tool then gives the output in accordance to Listing 3. Each formal GSN claim is evaluated against the AADL properties in order to decide whether the claim is consistent with the architectural model.

By this example, we have validated the GAGE method internally; that is, we have shown that it is possible to implement the method with the expected result. However, the formal goals of Listing 2 are to some extent simplifications, see Section V for a discussion.

C. Case Study

CrossControl AB is a Swedish company manufacturing safety-critical products. We plan to, in the near future, develop a safety case based on one of CrossControl's safety-critical products. The product is a display computer including both software and hardware. It comes equipped with an Intel processor, a touch screen, the Linux operating system, and the Qt graphical system. It does also provide safety-critical functionality monitoring the execution, such as supervision of safe display rendering and safe sound management. The product specification includes eight safety functions. We are in the process of modeling the product in AADL and its safety argumentation in GSN, which will give us a comprehensive validation and provide us with feedback regarding the applicability of the GAGE method to a real-world case [6] as well as its limitations, which will help us improve the method and the tool. The case study will include a model of the system architecture in AADL and a safety case in GSN based on the safety functions. We also plan to investigate the *strategy* and *context* GSN elements.

[1]The sum, max, and and functions can also be called with a list as argument. In that case, the list becomes converted into a set and all duplicates are removed.

V. Discussion and Limitations

Most safety arguments up to date are informal and intended for human readers, and clearly not all arguments are possible to formalize. However, since there are obvious advantages with arguments that can be automatically evaluated, an interesting research question is to examine which arguments are suitable to formalize and what limitations there are for formalization.

One ever-present challenge when modeling is to abstract away details while avoiding over-simplification. Crucial for the argument whether the system (not the model) is sufficiently safe for humans is how well any model reflect the reality, and not least how well the chosen properties can represent the richness and complexity of the system (including issues related to processes, management, humans, training, competence, etc.). For instance, in the example of Section IV-B we chose to require MISRA-C [5] compliance for all software components, which is in line with many safety standards. Furthermore, we chose to represent it as a Boolean value, which makes it relatively straightforward to parse, map, evaluate, etc. MISRA-C does indeed includes rules which are required and can be automated, such as the prohibition of the *goto* statement in the C source code, which can easily be checked by a static analysis tool. However, other rules are advisory, and/or require human judgment. For instance, pointers and interrupts should be avoided but may be allowed if well motivated and used with caution, variable names should be well chosen, etc. Thus, the source code needs to pass through an appropriate review process, finally resulting in the "MISRA-C compliance" statement.

Another challenge is how to represent the (un)certainty of values, such as the timing values in the example. Execution times (e.g. shortest, longest, average) may be based on (a combination of) statistical testing or measurements which may depend on environmental conditions, static analysis based on some assumptions, etc. The safety argument should be partly used to justify why we believe that a Boolean expression has been sufficiently met. This becomes especially important to address when discussing compositional safety cases, i.e. where statements about components (made out of system context, or based on a previous system's context) are used in the argumentation of a new system as *solutions* to meet the system's *goals*. In isolation, it may be easy for us to examine these properties, but when there are multiple components, with varying degrees of certainty and confidence in the arguments it becomes more complex to justify some of these formalized statements. This is discussed in, e.g. Sentilles et al. [7].

An additional challenge is to put the models, properties etc., under strict configuration management, and ensure a process is in place where actual changes to system components are indeed reflected in the component properties in the model. The SafeCer project [8] is aiming to a tool-chain framework for exactly this purpose, see related work in Section VI.

One preliminary classification of safety arguments is the following:

- Arguments based on properties that can be objectively measured, such as length, weight, and temperature, can clearly be formalized.
- Arguments based on properties that can be subjectively assessed, such as competence of the personnel, can to a certain extent (level of training, course certificates) be measured and formalized.
- Arguments that complies with standards, used for components developed using the standard as a reference point. For instance, the Safety Integrity Level (SIL) 3 of the IEC61508 standard [9] would dictate a particular design flow. This kind of arguments could be formalized.
- "Proven-in-use" arguments based on experience showing that the component works under the specified conditions are harder to formalize.
- Reasoning about verification and validation evidence (even formal verification evidence) requires human insight. For instance, goal G3 in the safety case of Figure 2. If we perform regression analysis, one important question is how many tests need to be redone.

As this is a work in progress, we plan to elaborate on a more fine-grained classification, and describe how these are interrelated. For example, the verification method used to produce a value for a property (for instance, worst-case execution time) is related to certainty, and explicit (and to some extent, formalized) context dependencies are needed to know whether an argument is valid in a new system.

At the moment, the GAGE tool cannot detect portions of a safety argument that refer to recently-removed architectural elements, which could be a subject for the next version. Another future goal would be to perform traceability analysis between the safety case and the model; that is, to detect the safety cases affected by a change in a component property, and to detect the components affected by a change in a safety case.

VI. Related Work

The problem of a small change causing large impact of the safety case is an important challenge for future modular safety cases, which is being studied in the SafeCer project (http://www.safecer.eu). The project is researching the use of Component Based Software Engineering (CBSE) to develop safety critical systems, and support reuse of components. In SafeCer, the aim is to support the reuse of certification data, assisted by the use of safety arguments which capture the context, quality and extent of the argument [8]. The authors are part of the project, and this paper holds clear relevance for the project as our purpose is reuse of component safety cases.

There have been a number of approaches to extend the GSN notation. Kelly and Weaver [4] discuss the foundation of GSN and describes some extensions, such as maintenance of safety arguments, safety case patterns, and assurance of safety arguments as well as modular safety cases. Similar to this paper, the authors aim at reuse of safety cases. However, they do not try to formalize the safety case elements.

Attwood, Kelly, and McDermid [10] describe the refinement of requirements into specifications as a recursive process. In their paper, the authors offer a critique of standard traceability techniques and propose a method for developing traceability structures for requirements reuse, including an argument classification similar to our classification in Section V.

Fenn et al. [11] present an approach to modular and incremental certification, including the trial deployment on an aircraft programme, developed by the Industrial Avionics Working Group (IAWG). The authors discuss a method, similar to our method in Section III, including modular and incremental certification.

VII. Conclusions and Further Work

In this paper, we have presented the GAGE method that maps a safety case against a system architecture in order to perform impact analysis. We have also developed the GAGE tool that reads the GSN safety case and the AADL system architecture model and, in accordance with the GAGE method, traverses the arguments of the safety case and decides which of them are satisfied by evaluating the properties of the model.

We plan to further exam and categorise the different kinds of safety arguments in order to decide which safety claims are suitable to formalize. We also plan to use one of CrossControl's safety-critical products as a case study, in order to validate our approach.

One important goal is to apply our approach to component-based systems, i.e. partial or incomplete safety arguments that have to correspond to component information providing evidences that the claim holds.

Acknowledgement

This research work was partially supported by Swedish Research Council (VR) and the Swedish Foundation for Strategic Research via the ARTEMIS JU, Vinnova, and CrossControl AB in the SafeCer project (JU Grant Agreement number 269265) as well as the ITS-EASY Research School.

References

[1] EN50129, *Railway applications. Communication, signalling and processing systems. Safety related electronic systems for signalling*. European Standards in English, 2003.

[2] R. Miles and K. Hamilton, *Learning UML 2.0*. O'Reilly Media, 2006.

[3] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecture analysis & design language (AADL): An introduction," Technical Note CMU/SEI-2006-TN-011, Software Engineering Institute, Pittsburg, PA, USA, February 2006.

[4] T. Kelly and R. Weaver, "The Goal Structuring Notation – A Safety Argument Notation," in *Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.

[5] MISRA, *Guidelines for the use of the C language in critical systems*. The Motor Industry Software Reliability Association, 2004.

[6] R. Land, J. Carlson, S. Larsson, and I. Crnković, "Towards guidelines for a development process for component-based embedded systems," in *Workshop on Software Engineering Processes and Applications (SEPA) in conjunction with the International Conference on Computational Science and Applications (ICCSA)*, pp. 43–58, Springer, June 2009.

[7] S. Sentilles, P. Štěpán, J. Carlson, and I. Crnković, "Integration of extra-functional properties in component models," in *12th International Symposium on Component Based Software Engineering (CBSE 2009)*, 2009.
[8] P. Conmy, J. Carlson, R. Land, S. Björnander, O. Bridal, and I. Bate, "Deliverable D2.3.1 – Extension of Techniques for Modular Safety Argument," tech. rep., SafeCer – Safety Certification of Software-Intensive Systems with Reusable Components, 2012.
[9] International Electrotechnical Commission, "IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems (Part 0-Part 7)," 2004.
[10] K. Attwood, T. Kelly, and J. McDermid, "The Use of Satisfaction Arguments for Traceability in Requirements Reuse for System Families," in *International Workshop on Requirements Reuse in System Family Engineering*, 2004.
[11] J. Fenn, R. Hawkins, P. Williams, T. Kelly, M. Banner, and Y. Oakshott, "The who, where, how, why and when of modular and incremental certification," in *Proceedings of the 2nd IET International Conference on System Safety*, 2007.

APPENDIX

In this appendix, we present the AADL source code of the airbag system example of Section IV-B.

Listing 4 The AADL airbag model source code.

```
device Accelerometer
  features
    outSpeed: out data port;
end Accelerometer;

device Gyroscope
  features
    outAngle: out data port;
end Gyroscope;

device PressureSensor
  features
    outPressure: out event port;
  properties
    Info => "ReactionTime=5";
    Info => "MisraC=true";
end PressureSensor;

device Inflator
  features
    inTrigger: in event port;
  properties
    Info => "ReactionTime=30";
    Info => "MisraC=true";
end Inflator;

system AirBagControlUnit
  features
    inSpeed: in data port;
    inAngle: in data port;
    inLeftPressure: in event port;
    inFrontPressure: in event port;
    inRightPressure: in event port;
    outLeftSeatActuator: out event port;
    outRightSeatActuator: out event port;
    outLeftSideActuator: out event port;
    outRightSideActuator: out event port;
  properties
    Info => "ReactionTime=10";
    Info => "MisraC=true";
end AirBagControlUnit;

system AirBagSystem
end AirBagSystem;

system implementation AirBagSystem.impl
  subcomponents
    airBagControlUnit: system AirBagControlUnit;

    accelerometer: device Accelerometer;
    gyroscope: device Gyroscope;

    leftPressureSensor: device PressureSensor;
    frontPressureSensor: device PressureSensor;
    rightPressureSensor: device PressureSensor;

    leftSeatInflator: device Inflator;
    rightSeatInflator: device Inflator;
    leftSideInflator: device Inflator;
    rightSideInflator: device Inflator;
  connections
    data port accelerometer.outSpeed ->
              airBagControlUnit.inSpeed;
    data port gyroscope.outAngle ->
              airBagControlUnit.inAngle;

    event port leftPressureSensor.outPressure ->
               airBagControlUnit.inLeftPressure;
    event port frontPressureSensor.outPressure ->
               airBagControlUnit.inFrontPressure;
    event port rightPressureSensor.outPressure ->
               airBagControlUnit.inRightPressure;

    event port airBagControlUnit.outLeftSeatActuator ->
               leftSeatInflator.inTrigger;
    event port airBagControlUnit.outRightSeatActuator ->
               rightSeatInflator.inTrigger;
    event port airBagControlUnit.outLeftSideActuator ->
               leftSideInflator.inTrigger;
    event port airBagControlUnit.outRightSideActuator ->
               rightSideInflator.inTrigger;
end AirBagSystem.impl;
```