Electronic Preprint for Journal of Information Processing Vol.21 No.1

Regular Paper

Qualitative Comparison of ADL-Based Approaches to Real-World Automotive System Development

Shin'ichi Shiraishi[1,a)]

Received: February 17, 2012, Accepted: September 10, 2012

Abstract: This paper presents two different model-based approaches that use multiple architecture description languages (ADLs) for automotive system development. One approach is based on AADL (Architecture Analysis & Design Language), and the other is a collaborative approach using multiple languages: SysML (Systems Modeling Language) and MARTE (Modeling and Analysis of Real-Time and Embedded systems). In this paper, the detailed modeling steps for both approaches are explained through a real-world automotive development example: a cruise control system. Moreover, discussion of the modeling steps offers a qualitative comparison of the two approaches, and then clarifies the characteristics of the different types of ADLs.

Keywords: automotive systems, architecture description languages, AADL, SysML, MARTE

1. Introduction

The electrical and electronic (E/E) components of automotive systems are continuously growing in number and complexity. In the case of luxury cars, the number of ECUs (Electronic Control Units) is approaching 100, so that the ratio of the production cost of E/E components to that of the other types of components is expected to exceed 40% in 2015. In particular, the size of software of E/E components is expanding exponentially. More objectively, several reports [1], [2] independently predict that 100 million lines of in-vehicle software should appear around in 2015. This fact implies that the software complexity has already become a critical concern in effective automotive system development.

Based on this background, model-based approaches have attracted considerable attention of automotive engineers due to their potential improved productivity. In most cases, however, the expectation of engineers is ambiguous. Thus, we must recognize what we need to do through model-based approaches for the purpose of solving the above concern.

There exist lots of modeling languages which support model-based approaches. MATLAB/Simulink [*1] and ESTEREL SCADE [*2] are the two greatest *proprietary* modeling languages. In the *public* domain, UML is the most popular language, and lots of tools for the UML-based modeling are available. On the other hand, engineers working for car manufacturers (OEMs) usually need to have a higher viewpoint overlooking the whole vehicle systems than that the above well-known languages give. In other words, the architecture of systems is essential to automotive OEM engineers rather than the details of each system. When we apply the same discussion to model-based approaches, we can realize that architecture modeling techniques, more precisely, modeling languages and methodologies for system architecture, are necessary.

Therefore, this paper tries to derive a practical approach (modeling languages and methodologies) to architecture modeling of automotive systems. In this paper, we choose architecture description languages (ADLs) from among several kinds of modeling languages, and focuses on their practical application to the automotive domain. Specifically, two types of approaches based on different ADLs are introduced. We will explain how these approaches attain the development phases of automotive systems by using several example models, and thereafter clarify their characteristics by comparing from several viewpoints. The same topic has been roughly discussed in our previous paper [3]; however, we provide here more precise discussion and more detailed comparison than the previous paper [3]. The improved fidelity should be helpful for automotive engineers to understand the practical application of ADLs.

Although this paper focuses on automotive system architecture, system behavior, which is orthogonal to the architectural aspect, is also important for system development. While the system architecture is still central to this paper, we briefly discuss the behavioral aspect in some sections in a complementary way.

2. Survey on Architecture Description Languages

Architecture description languages (ADLs) have been widely studied across several research fields and industries. Medvidovic et al. have already conducted an exhaustive survey of ADLs from a purely technical point of view [4]. In contrast to the previous research [4], we performed our survey from a practical point of view as follows.

[1] TOYOTA InfoTechnology Center, U.S.A., Inc., Mountain View, CA 94043, U.S.A.
[a)] sshiraishi@us.toyota-itc.com
[*1] http://www.mathworks.com
[*2] http://www.esterel-technologies.com

Firstly, we defined seven requirements of ADLs that are our expectations from an OEM point of view. Next, we collected lots of ADLs from both industry and research fields. Finally, we selected promising ADLs for us through a three-step selection process while comparing the features of the collected ADLs and the requirements. The following sections give the details of each stage of our survey.

2.1 Requirements of ADLs from the Automotive Manufacturer's Viewpoint

This section summarizes the requirements of ADLs from an OEM viewpoint. In order to cope with the problems mentioned in Section 1 either in a direct or an indirect way, we expect ADLs to fill the following requirements.

2.1.1 System Characteristics Aspect

First, we consider the *system characteristics aspect*, and derive the following three requirements are derived.

(1) Large-scale systems

While automotive systems are widely distributed systems consisting of lots of ECUs, each ECU has a certain (large or small) amount of software. In order to describe such large systems (systems of systems), ADLs shall describe complex hierarchical and/or layered architecture while capturing the precise definition of components within the architecture. Therefore, the derived sub-requirements are the modeling capability of *hierarchical architecture*, *layered architecture*, and *component specification*.

(2) Real-time systems

Most automotive systems, e.g., cruise control systems discussed later, are real-time systems. These systems have critical timeliness requirements such as hard deadlines. Thus, we need to embed real-time properties, like latencies of components, deadlines of paths, etc. onto architecture models. It is preferred that timeliness analysis techniques based on the embedded properties are available. Therefore, the derived sub-requirements are the capability of *real-time property modeling* and *formal real-time analyses* [*3].

(3) Distributed embedded systems

As shown in **Fig. 1**, ECUs consist of hardware components such as microcontrollers, bus controllers and devices; and software. These ECUs are interconnected to each other via in-vehicle buses e.g., CAN buses, LIN buses, etc. Therefore, we need to describe the different types of components (software, hardware, and networks) in ADLs. Therefore, the derived sub-requirements are the modeling capability of *software architecture* and *hardware architecture* (execution platform including networks).

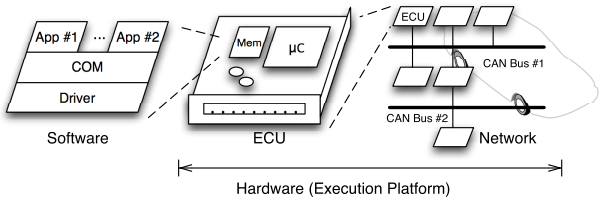


Fig. 1 Basic structure of automotive systems.

[*3] These kinds of technologies such as analysis, verification, and transformation can be discussed independently from modeling languages. However, the existence of these technologies, e.g., analysis tools, implicitly shows a sort of capability of modeling languages. Thus, we directly incorporate the tool availability into the requirements of ADLs.

2.1.2 System Development Aspect

The second aspect is the *system development aspect*. We derive the following requirements along with this aspect.

(4) Source code handling

As has discussed in Section 1, high-level modeling is required in any OEM company. On the other hand, certain limited divisions and groups working for our core competencies, e.g., engine systems or hybrid systems in our case, need to handle low-level artifacts, that is, source code. Moreover, legacy code handling is another concern which comes from the more than thirty-year-long history of E/E automotive systems. Therefore, ADLs shall be equipped with scalability from the high- to low-level modeling. Therefore, the derived sub-requirements are the capability of *modeling at multiple abstraction levels* and *code generation*.

(5) Distributed development

Automotive system development is widely distributed which involves lots of stakeholders, e.g., an OEM, suppliers (ECU manufacturers), device suppliers, etc. This means that architecture models are shared by such a variety of stakeholders. Therefore, ADLs should be a common language and facilitate seamless development among the stakeholders. Therefore, the derived sub-requirements are the openness of language specifications (*open de facto* or *de jure standards*) and the capability of *seamless refinement*.

2.1.3 Manufacturer Business Aspect

Third, we consider the *manufacturer business aspect* and the following requirements are derived.

(6) Large-scale product line

While both use cases and users of vehicles are diverse, the automotive market is worldwide. Considering this diversity, preparing large-scale product lines of vehicles is mandatory for OEMs. Obviously, a wide variety of vehicles lead to a wide diversity of automotive system specifications. This requirement can be translated into the capability of *variability modeling*.

(7) High-level system assurance

Some automotive systems are highly safety critical systems, so that we need to ensure high-level dependability in such systems. Moreover, the new automotive safety standard [5] published in 2011 requires OEMs to build safety cases for the purpose of justifying their safety claims. In this context, we expect ADLs to facilitate building of safety cases such that architecture models themselves are regarded as trustworthy evidence, and formal and/or semi-formal analyses based on the architecture models form convincing arguments [*4]. Furthermore, it is necessary that the system dependability is considered in the architecture models. For this purpose, we need to create models explicitly describing errors of systems. In other words, ADLs shall be equipped with vocabulary for error modeling [*5]. Therefore, the derived sub-requirements

[*4] For example, we can find a model-based approach to assurance cases in Ref. [6].

[*5] Analysis techniques based on error modeling are also necessary to realize high-level system assurance. Certainly, there are several analysis techniques in the research field [7]; however, this paper focuses on practical approaches. Therefore, this paper does not have any further discussion on the analysis based on error modeling.

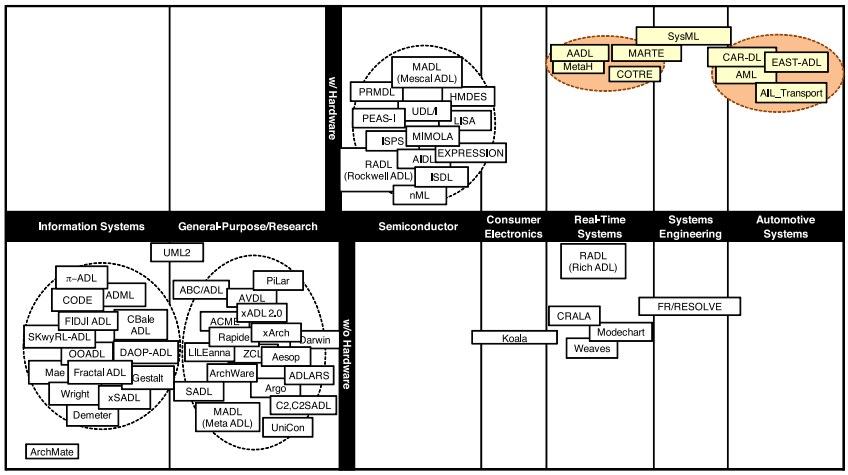


Fig. 2 Classification of architecture description languages.

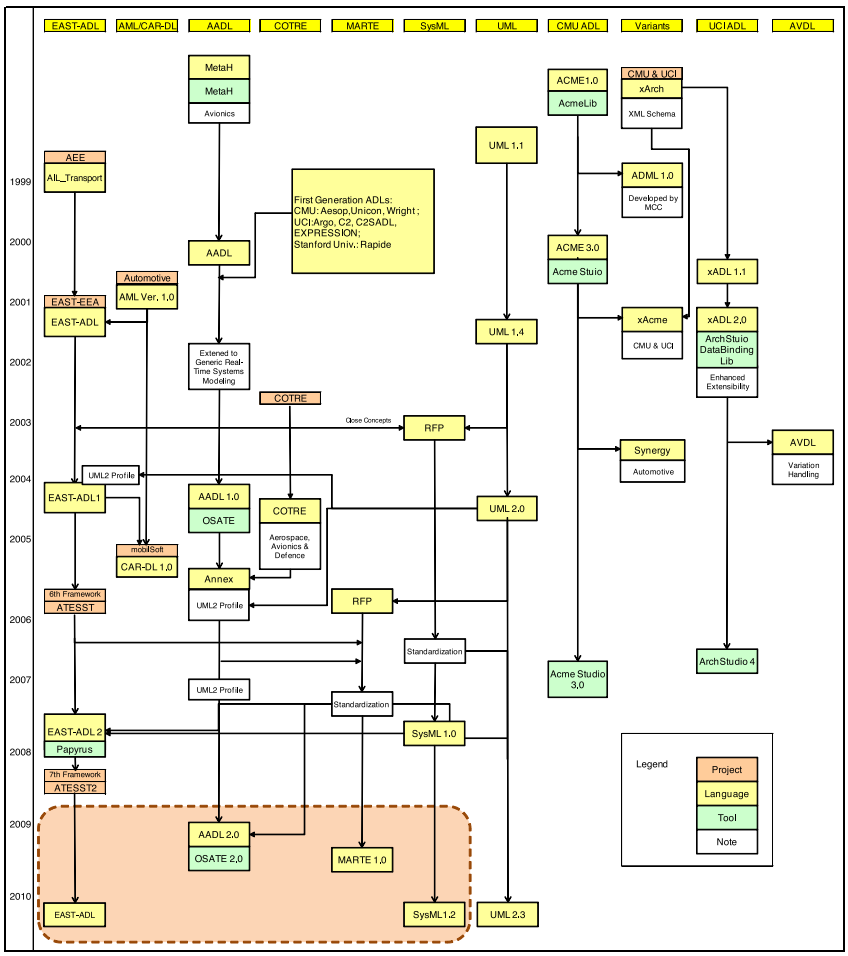


Fig. 3 Family tree of firstly selected architecture description languages.

are the formalism of language specifications (*formal notation*), and the capability of *formal verification* and *error modeling*.

In summary, all the requirements and sub-requirements discussed above are described in the first and second columns of Table 3. In the table, the sub-requirements related to supporting technologies (tools) are shown with annotations for readability.

2.2 Classification and Qualification of ADLs: The First and Second Selections

In our survey, we collected more than sixty ADLs from published research papers and project reports. **Figure 2** shows the two-dimensional classification of ADLs based on their objectives, e.g., information system development, real-time system development, etc.

When we remember the first three requirements (1)–(3), we can choose the nine ADLs that are highlighted in the orange el-

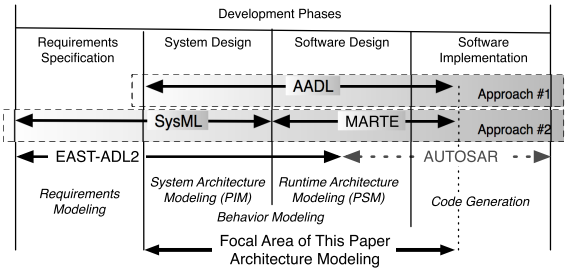


Fig. 4 Comparison of design phase coverage of ADLs.

lipses. The list of the first selection consists of: AADL (Architecture Analysis and Design Language) [8], AIL_Transport (Architecture Implementation Language Transport) [9], AML (Automotive Modeling Language) [10], CAR-DL (Combined Architecture Description Language), COTRE [11], EAST-ADL [12], MARTE (Modeling and Analysis of Real-Time and Embedded systems) [13], MetaH [14], and SysML (Systems Modeling Language) [15] (in alphabetical order). Because they are developed for either large-scale, real-time, or automotive systems, they satisfy at least one of the first three requirements.

Figure 3 shows the family tree of the selected nine ADLs. The family tree clearly shows that some ADLs are no longer maintained, such as discontinued or merged. In these cases, we cannot acquire their latest information even if it exists, and further discussion is difficult. In other words, the list of the second selection consists of the four ADLs enclosed by the orange oval box in Fig. 3: AADL, EAST-ADL, MARTE, and SysML. AADL is standardized by SAE International (Society of Automotive Engineers). SysML and MARTE are published by OMG (Object Management Group). EAST-ADL is an outcome of the ATESST2 (Advancing Traffic Efficiency and Safety through Software Technology) project funded by an EU committee.

2.3 Quick Comparison of ADLs: The Final Selection

Here let us discuss the requirements along with the system development aspect: (4) source code handling and (5) distributed development. Regarding the requirement (4), **Fig. 4** shows the development phase coverage of each ADL. As shown in the figure, AADL covers the software design phase adjacent to the software implementation (coding). Unlike AADL, SysML focuses only on system design; however, MARTE can be used for the software design in a complementary way. On the contrary, EAST-ADL does not provide a bridge between models and source code, and instead delegates this task to AUTOSAR (AUTomotive Open System ARchitecture) [16]. Although AUTOSAR is a useful standard to handle brand-new systems at a low abstraction level, it does not offer any method to capture a great deal of legacy code. We must therefore conclude that EAST-ADL does not satisfy the requirement (4).

Moreover, EAST-ADL is not yet standardized, and is difficult to regard as a de facto standard due to its limited supporting tools. Therefore, it is not easy to use EAST-ADL as a common language among stakeholders involved in automotive system development. That is, the requirement (5) is not filled by EAST-ADL. Certainly, EAST-ADL is an outstanding research outcome of an EU project; however, we need to remove it from our list. Consequently, based on the whole discussion in this section, the list of the final selection consists of three ADLs: AADL, MARTE and SysML [*6].

3. Case Study: ADL-Based System Development

This section shows detailed development steps based on different ADLs by using a case study. We will apply the obtained ADL models to the qualitative comparison of ADL-based approaches in the next section.

3.1 Real-World Example: Adaptive Cruise Control SystemA12,B3

In our case study, we use an automotive control system example. The example system is an adaptive cruise control (ACC) system whose specifications are borrowed from a particular luxury production car. The ACC system provides the following two major functions: (i) constant-speed cruise (CSC) control that maintains the desired vehicle speed; and (ii) constant-distance cruise (CDC) control that maintains an adequate follow distance.

Fig. 5 shows a hardware configuration of the ACC system. The Driving Support Computer (DSC), which is an ECU (Electrical Control Unit), is the central component of the ACC system. Thus, the software running on DSC is the main topic of the following sections. While some peripherals such as Skid Control Computer (SCC) and Engine Control Computer (ECC) are used for both the CSC and CDC control, the other components like Radar Sensor (RADAR) and Distance Setting Switch (DISTSET_SW) are dedicated to the CDC control.

3.2 ADL-Based Development Process

In considering the design phase coverage of ADLs shown in Fig. 4, we can have the following two approaches: an AADL-based method (hereafter *approach #1*) and another method based on the combination of the SysML and MARTE (*approach #2*). However, ADLs are no more than modeling languages, so that we need to define adequate development processes and methodologies. In our case study, we adopt the premise that we will follow the development process shown in **Table 1**, which takes the design phase coverage into account.

3.3 Approach #1: AADL-Based System Development

Before starting detailed discussion, we explain the modeling fundamentals of AADL. AADL descriptions consist of basic elements called *components*. An interface design (external specification) of components is given by a *type* definition in AADL. In contrast, an internal specification is specialized by an *implementation* of the type. By alternatively repeating the two steps: the *type definition–implementation*, we eventually arrive at a hierarchical description of the system architecture.

Listing 1 displays that a sample system (`system1`) is a **`system`** component and it consists of two **`processes`**: `prod` and `cons`. The interface of `prod` is defined in the **`features`** part of the type of `producer`. On the other hand, `system1.imp` indicates that `system1` is *implemented* by `producer` and `consumer` (**`subcomponents`** part),

[*6] The requirements (6) and (7) will be discussed in Sections 4.1 and 4.2.

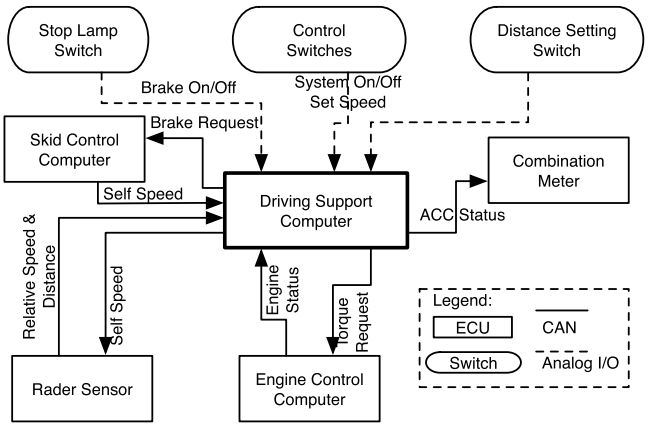


Fig. 5 Example: Adaptive cruise control system.

Table 1 ADL-based development processes.

#	Development Phases	Tasks	Languages	
			Approach #1	Approach #2
1)	**System Design**	Modeling of system architecture (peripheral devices and systems, an execution platform of software, logical software architecture, etc.) and system-level behavior.	AADL	SysML
2)	**Software Design**	Modeling of runtime architecture (physical software architecture including processes, threads, etc.) and component-level behavior.	AADL	MARTE
3)	**Software Implementation**	Code generation (automatic or manual).	C Language	

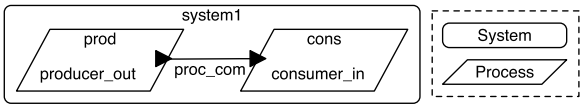


Fig. 6 Example AADL model (Graphical).

and their interconnection (`connections` part). **Figure 6** displays a graphical model, which is the same as Listing 1.

Listing 1: Example AADL Model (Textual).

```
system system1 -- Type Definition of system1 (No Interface )
end system1;
system implementation system1.imp -- Implementation of System1
subcomponents -- Two processes
  prod: process producer;
  cons: process consumer;
connections
  proc_com: data port prod.producer_out -> cons.consumer_in;
end system1.imp;
process producer -- Type Definition of producer
features -- Interface (output)
  producer_out: out data port;
end producer;
process consumer -- Type Definition of consumer
features -- Interface (input)
  consumer_in: in data port;
end consumer;
```

3.3.1 System Architecture Modeling

The system architecture indicates system functions by combining logical components. Therefore, we can use the `system` component of AADL as a symbol for logical components. **Figure 7** and Listing 2 show the same system architecture model.

In this model, we define an empty type of *ACCSys* followed by its implementation *ACCSys.SysArch*. This means that *ACCSys* is the top-level component and *ACCSys.SysArch* shows its system architecture. The system architecture *ACCSystem.SysArch* is described as the combination of components such as the logical software component (*ACCSW*: an instance of *ACCSoftwareLog*) and the execution platform (*DSCHW*: an instance of *DSCHardware*). The required peripherals like the skid control computer (*SCC*), the engine control computer (*ECC*), and the control switches (*CTRL_SW*), are also described in the same model. These peripherals are connected to the the central component (*ACCSW*) as shown in the `connections` part. The implementation of each connection is also defined e.g., the connection between *SCC* and *ACCSW*, which exchanges the vehicle speed, is realized on the CAN bus (*hs_can_bus_v*).

Listing 2: AADL System Architecture Model (Textual).

```
system ACCSys -- Top-Level Component
end ACCSys;
system implementation ACCSys.SysArch -- System Architecture
  subcomponents
    ACCSW: system ACCSoftwareLog; -- Subsystems
    DSCHW: system DSCHardware;
    CTRL_SW: system control_switch; -- Peripherals
    ECC: system engine_control;
    SCC: system skid_control;
    RADAR: system radar_sensor;
    (snip)
  connections
    GC0000: port group CTRL_SW.pg_ctrl_sw_status->ACCSW.
              pg_ctrl_sw_status;
    DC0100: data port SCC.self_speed->ACCSW.self_speed_in;
    DC0500: data port ACCSW.required_drive_force->ECC.
              required_drive_force;
    can_msg_v_r_0004: bus access DSCHW.hs_can_bus_v
       -> ACCSW.hs_can_bus_self_speed_in;
    (snip)
  flows
    EE0100: end to end flow SCC.FS0100 -> (snip) -> ECC.FS0500
       {Latency=>70 Ms;};
  (snip)
end ACCSys.SysArch;
```

3.3.1.1 Logical Software Architecture Modeling

Next, we discuss the logical software architecture *ACCSoftwareLog*. The interface of *ACCSoftwareLog* is defined in its type definition part in Listing 3. These interface specifications are equipped with the units of data exchanged via the interfaces. For example, the vehicle speed input (*self_speed_in*) in *kph_type* [km/h] and torque demand output (*required_drive_force*) in *Nm_type* [Nm]. The interface between *ACCSoftwareLog* and *CTRL_SW*

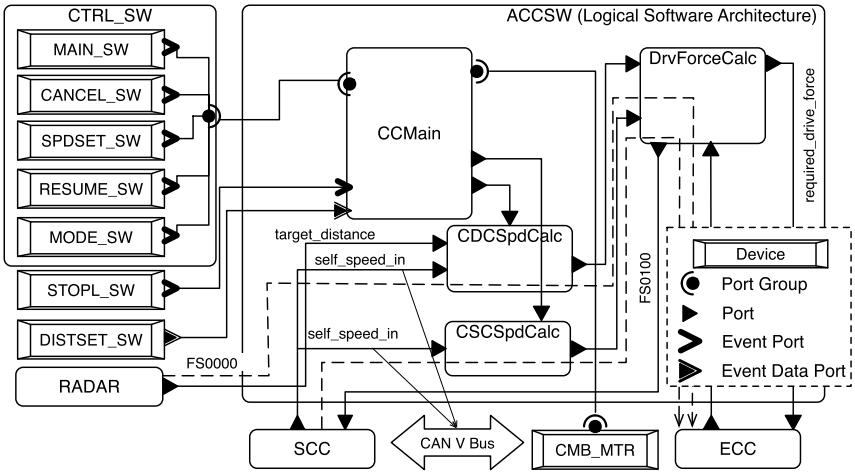


Fig. 7 AADL system architecture model (Graphical).

is described by **port group** that includes multiple interfaces (see the bottom of Listing 3). The internal structure of *ACCSoftwareLog* is described in its implementation (*ACCSoftwareLog.imp*). In this model, *ACCSoftwareLog.imp* is the combination of the following four logical components (functions): *CCMain*, *CSCSpdCalc*, *CDCSpdCalc*, and *DrvForceCalc*. If necessary, the internal structure of the logical components can be drilled down. The **flows** part in Listings 2 and 3 defines event flows, data flows, etc. These flows are exploited in the runtime architecture analyses in the next section.

As have been above, we can perform system architecture modeling of complex systems just by repeating the two simple steps: (i) define the type of a logical component (**system**) and (ii) implement the component.

Listing 3: AADL Logical Software Architecture Model.

```
system ACCSoftwareLog -- Type Definition of Logical Software
                         Architecture
  features
    pg_ctrl_sw_status: port group
      pg_receptacle_ctrl_sw_status; -- Multiple Interfaces
    self_speed_in: in data port kph_type; -- [km/h]
    required_drive_force: out data port Nm_type; -- [Nm]
    (snip)
  flows
    FS0100: flow path self_speed_in -> required_drive_force;
    (snip)
end ACCSoftware;
system implementation ACCSoftwareLog.imp -- Implementation
  subcomponents
    CSCSpeedCalc: system csc_speed_calculator in modes (
                  csc_mode);
    CDCSpeedCalc: system cdc_speed_calculator in modes (
                  cdc_mode);
    (snip)
  modes -- CSC Mode and CDC Mode
    csc_mode: initial mode;
    cdc_mode: mode;
    csc_mode -[pg_ctrl_sw_status.mode_sw_status]-> cdc_mode;
    cdc_mode -[pg_ctrl_sw_status.mode_sw_status]-> csc_mode;
end ACCSoftwareLog.imp;
data kph_type -- Data Types
end kph_type;
port group pg_receptacle_ctrl_sw_status_csc
  features
    main_sw_status: in event port;
    mode_sw_status: in event port;
    (snip)
end pg_receptacle_ctrl_sw_status_csc;
(snip)
```

3.3.1.2 Behavior Modeling

While AADL provides enough vocabulary to describe architecture of systems, a limited vocabulary for behavior modeling is also available. For example, the **modes** part in Listing 3 indicates two internal states (**csc_mode** and **cdc_mode**) of *ACCSoftwareLog.imp*. As explained in Section 3.1, the ACC system has two types of control, which can chosen by a driver. Regarding this functionality, the AADL model shows that the transition between two modes is triggered by the event of *mode_sw_status* from *CTRL_SW*. Synchronizing with the mode transition, an active component is also switched between *CSCSpeedCalc* and *CDCSpeedCalc*. As shown in this example, AADL offers a kind of state-based modeling. Moreover, event chains such that an event propagates from a component through its subcomponents enable hierarchical modeling aligning with the corresponding system architecture.

The behavior modeling example shown above is trivial but important because we cannot expect AADL to provide further complex behavior modeling. Fortunately, if necessary, we can use state-of-the-art behavior modeling technologies, e.g., AADL Behavior Annex [17], a collaboration technique with Simulink [18], etc.

3.3.2 Runtime Architecture Modeling

The runtime architecture of software consists of physical software entities such as processes and threads. Therefore, we can straightforwardly use the **process** and **thread** components from the AADL vocabulary. Here, we need to replace *ACCSoftwareLog* by a combination of these physical entities. This substitution can be easily started with **extends** and **refined to** as shown in Listing 4. More precisely, the system architecture *ACCSys.SysArch* is extended such that the runtime software architecture *ACCSoftwarePhy.imp*, which is an implementation of *ACCSoftwarePhy*, substitutes for the logical software architecture *ACCSoftwareLog.imp*. Together with the above substitution (refinement), the type definition of *ACCSoftwarePhy* also refines its interfaces. For instance, the vehicle speed input (*self_speed_in*) is refined to *kph_fxp_type* which is 16-bit data with two fraction digits.

Listing 4: AADL Runtime Architecture Model (System).

```
system implementation ACCSys.RunArch
  extends ACCSys.SysArch -- Runtime Architecture (System)
  subcomponents
    ACCSW: refined to system ACCSoftwarePhy.imp;
end ACCSys.RunArch;
system ACCSoftwarePhy -- Software Runtime Architecture
  extends ACCSoftwareLog -- Logical -> Physical
  features
    self_speed_in: refined to in data port kph_fxp_type;
    FS0100: refined to flow path {Latency => 60 Ms;};
end ACCSoftwarePhy;
data kph_fxp_type -- Refine Data Type Definition
  extends kph_type
```

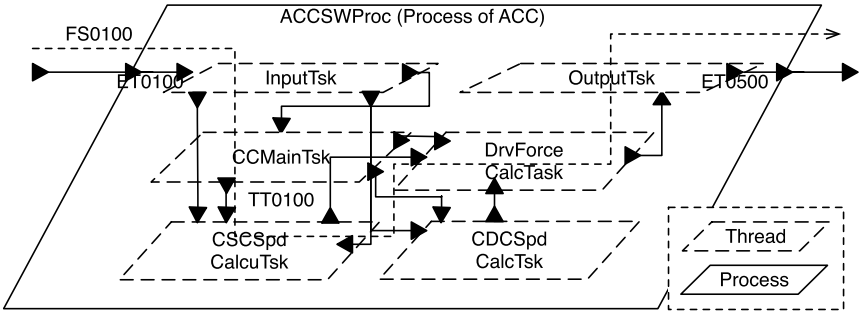


Fig. 8 AADL runtime software architecture model (cf. Fig. 7).

```
  properties
    Source_Data_Size => 16 Bits;
    Data_digits => 5;
    Data_scale => 2;
end kph_fxp_type;
```

Next, let us consider the implementation of the runtime software architecture (*ACCSoftwarePhy.imp*). In this case study, we only develop the ACC system on DSC so that we can adopt a single-process architecture. The single-process architecture is easily described as shown in Listing 5.

Listing 5: AADL Runtime Architecture Model (Software).

```
system implementation ACCSoftwarePhy.imp -- Implementation
  subcomponents -- Single Process
    ACCSWProc: process ACCSoftwareProcess.imp;
  connections
    (snip)
  flows
    FS0100: flow path self_speed_in -> (snip) ->
              required_drive_force;
end ACCSoftwarePhy.imp;
process ACCSoftwareProcess -- Type Definition of Process
  features
    pg_ctrl_sw_status: port group pg_receptacle_ctrl_sw_status
              ;
    self_speed_in: in data port kph_fxp_type;
    required_drive_force: out data port Nm_fxp_type;
    (snip)
  flows
    FS0100: flow path self_speed_in -> required_drive_force
    {Latency => 50 Ms;};
end ACCSoftwareProcess;
```

On the other hand, the process consists of multiple threads (tasks). In Listing 6, we prepare the following four threads: *CCMainTsk*,*CSCSpdCalcTsk*, *CDCSpdCalcTsk*, and *DrvForceCalcTsk*; which are derived from the four logical components in Fig. 7. Moreover, we prepare two more additional threads: (*InputTsk*,*OutputTsk*) for input and output signals processing.

Listing 6: AADL Process Model.

```
-- Implementation of Process
process implementation ACCSoftwareProcess.imp
  subcomponents
    InputTsk: thread input_controller_thread.imp;
    CCMainTsk: thread ccmain_thread.imp;
    CSCSpdCalcTsk: thread csc_speed_calculator_thread.imp;
    CDCSpdCalcTsk: thread cdc_speed_calculator_thread.imp;
    DrvForceCalcTsk: thread drive_force_calculator_thread.imp;
    OutputTsk: thread output_controller_thread.imp;
  connections
    ET0100: data port self_speed_in -> InputTsk.self_speed_in;
    ET0500: data port OutputTsk.required_drive_force_out ->
              required_drive_force;
    TT0100: data port InputTsk.self_speed_out -> CSCSpdCalcTsk
              .self_speed_in;
     (snip)
  flows
    FS0100: flow path self_speed_in -> ET0100 -> (snip)
    -> ET0500 -> required_drive_force in modes (csc_mode);
    (snip)
end ACCSoftwareProcess.imp;
```

Finally, we describe thread models as leaf components of the runtime architecture. For example, the thread *CSCSpdCalcTsk*, which implements control algorithm for the CSC mode, is described as shown in Listing 7. As have seen above, by repeating the two step modeling again, we can eventually obtain the runtime architecture model, which is located at the adjacent level to the source code level (see Fig. 4).

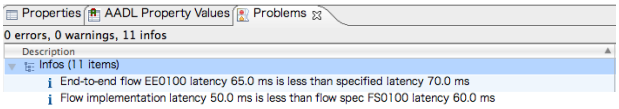


Fig. 9 Latency analyses for end-to-end and flow paths.

Listing 7: AADL Thread Model.

```
thread csc_speed_calculator_thread -- Type Definition of Thread
  features
    self_speed_in: in data port kph_fxp_type;
    target_speed: out data port kph_fxp_type;
  flows -- w/ Expected Latency
    FS0100: flow path self_speed_in
    -> target_speed {Latency => 5 Ms;};
end csc_speed_calculator_thread;
thread implementation csc_speed_calculator_thread.imp --
            Implementation
  flows
    FS0100: flow path self_speed_in -> target_speed;
  properties
    Dispatch_Protocol => Periodic;
    Period => 10 Ms;
    Deadline => 10 Ms;
    Compute_Execution_Time => 3 Ms .. 5 Ms;
end csc_speed_calculator_thread.imp;
```

Figure 8 shows a graphical model of the runtime software architecture. Although the graphical notation is easy-to-understand, its expressiveness is limited as compared to the textual model in Listings 4–7.

3.3.2.1 Formal Analyses

In Listing 7, real-time properties such as latencies and deadlines, are embedded onto thread models. By integrating these embedded properties, we can perform some types of formal real-time analyses, e.g., a schedulability analysis, a latency analysis, etc. **Figure 9** shows the results of latency analyses of the end-to-end path of *EE0100* in Listing 2 and the flow path of *FS0100* in Listing 4. The results show that the expected latencies satisfy the corresponding deadlines specified in the model. These analysis techniques are available in a standard modeling tool called OSATE (Open Source AADL Tool Environment) [*7], so that we can use them anytime if necessary.

3.3.3 Code Generation

We can derive source code templates from the runtime architecture model obtained in Section 3.3.2. For example, if we use the OSEK/VDX-C platform [19], we can produce a configuration file in OIL (OSEK Implementation Language) shown in Listing 8 and the skeleton code shown in Listing 9. The AADL specification [8] itself contains the guidelines for translation from AADL descriptions into source code. Moreover, a source code genera-

[*7] http://www.aadl.info/aadl/currentsite/tool/osate.html

tor [*8] has been developed for some platforms. However, it should be noted that these translations depend heavily on the platform used; that is, different target platforms require different translation strategies.

In summary, AADL allows us to perform the following modeling steps seamlessly: (i) system architecture modeling, (ii) refinement of runtime architecture models, and (iii) translation into source code templates.

Listing 8: Generated Configuration in OIL.

```
/* Tasks */
TASK InputTsk {
  TYPE = BASIC;
  /* Snip */
}
TASK CSCSpdCalcTsk{
  TYPE = BASIC;
  /* Snip */
}
/* Messages */
Message GC0000 { /* CAN Message (External Communication) */
  TYPE = EXTERNAL;
  ACCESSNAME = {InputTsk_self_speed_in};
  CAN_ADDRESS = {can_msg_v_0004};
  /* Snip */
};
Message TT0010 {/* Inter-Task Message (Internal Communication) */
  TYPE = INTERNAL;
  ACCESSNAME = {InputTsk_self_speed_out,
                CSCSpdCalcTsk_self_speed_in};
  /* Snip */
};
```

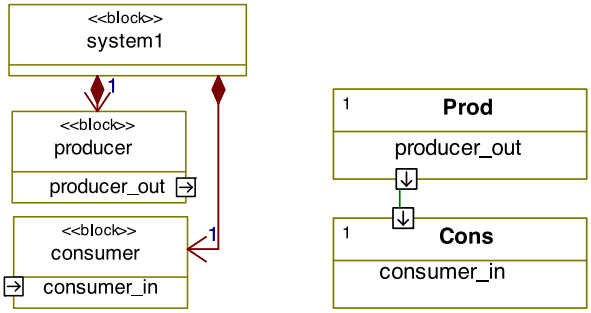
Listing 9: Generated Skelton Code of Tasks.

```
TASK (InputTsk) {
  short l_self_speed_in = 0;
  while(ReceiveMessage(GC0000, InputTsk_self_speed_in) ==
              E_OK){
  /*CAN Message (External Communication) */
    l_self_speed_in = InputTsk_self_speed_in;}
    /* Snip */
  SendMessage(TT0100, InputTsk_self_speed_out) ;
  /* Inter-Task Message ( Internal  Communication)*/
}
TASK (CSCSpdCalcTsk) {
  short l_s_speed_in = 0;
  while(ReceiveMessage(TT0100, CSCSpdCalcTsk_self_speed_in)
              == E_OK){
    l_self_speed_in =CSCSpdCalcTsk_self_speed_in; }
    /* Snip */
}
```

3.4 Approach #2: SysML / MARTE-Based System Development

First of all, we explain the modeling fundamentals of SysML. SysML provides two typical diagrams that are useful for system architecture modeling: BDD (Block Definition Diagram) and IBD (Internal Block Diagram). These two diagrams are similar to the type definition and implementation of AADL, respectively.

The AADL example shown in Fig. 6 is translated into **Fig. 10** (a), which is a BDD model of system1. In this diagram, the interface of system1 (no interface in this case) and its subcomponents are described [*9]. As shown in the diagram, the primitive components are components whose stereotype is ≪block≫ [*10]. In the diagram, the interfaces of the subcomponents are also defined as *flow ports* (e.g., producer_out). On the other hand, the internal structure of system1 is described as the interconnection of its subcomponents by using IBD as shown in Fig. 10 (b). Similarly to the AADL case, SysML also offers hierarchical modeling by repeating two modeling steps that are based on BDD and IBD, respectively. We do not discuss the MARTE case here due to the space limitation; however, we can adopt the same approach as the SysML case.



(a) Example of Block Definition Diagram (BDD).

(b) Example of Internal Block Diagram (IBD).

Fig. 10 Example SysML Models (cf. Fig. 6).

3.4.1 System Architecture Modeling

As shown in Table 1, we use SysML, mainly, BDD and IBD models, in the system architecture modeling. **Figure 11** is an IBD model that is equivalent to the AADL model in Section 3.3.1. Certainly, the BDD model corresponding to Fig. 11 is also necessary; however, it was omitted due to space limitations. In addition to the target hardware DSCHardware, the peripheral systems and devices such as SCC and CTRL_SW are described in the diagram. The target software DSCSoftware is deployed on DSCHardware by using *allocation*. While the following stereotypes: ≪system≫, ≪device≫,≪bus≫, and ≪hardware≫ are newly introduced for readability; ≪HwResource≫ comes from HRM (Hardware Resource Modeling) of MARTE and indicates a shared hardware resource.

In the figure, the interfaces (*flow ports*) of DSCHardware are specified by ≪flowSpecification≫. For example, the interface to CTRL_SW has the type of fs_ctrl_sw_status that is specified by ≪flowSpecification≫ as shown in **Fig. 12** (a). The flow specification shows that DSCHardware receives the inputs from the main switch, mode setting switch, etc. The functionality of *flow specification* is quite similar to that of **port group** of AADL. The connections between interfaces are also specified precisely in the model, e.g., the connection between DSCHardware and SCC is implemented by a CAN message on the CAN V bus (see the allocation part of hs_can_bus_v). The unit of data exchanged over this connection is specified as kph [km/h] by *valueType* as shown in Fig. 12 (b). Although this example uses *flowSpecification* and *valueType* exclusively, we can use them together.

3.4.1.1 Logical Software Architecture Modeling

Next, we consider logical software architecture. In this case study, all the required functionalities are implemented by software. Thus, the top-level software component of DSCSoftware and its logical variant (DSCSoftwareLog) are obtained by specializing DSCHardware because the specialization allows the software components to inherit all the interfaces from DSCHardware. This interface inheritance is described as the BDD model in **Fig. 13** (a). On the other hand, an IBD model is used in order to describe the internal structure of DSCSoftwareLog. The IBD model shown in Fig. 13 (b) describes the equivalent model to Fig. 7. As have seen here, the modeling procedure of DSCSoftwareLog is exactly the same as that of the upper level component of ACCSys_SysArch.

[*8] For example, http://penelope.enst.fr/aadl/

[*9] In contrast, the type definition of AADL does not describe subcomponents.

[*10] ≪≫ indicates a stereotype.

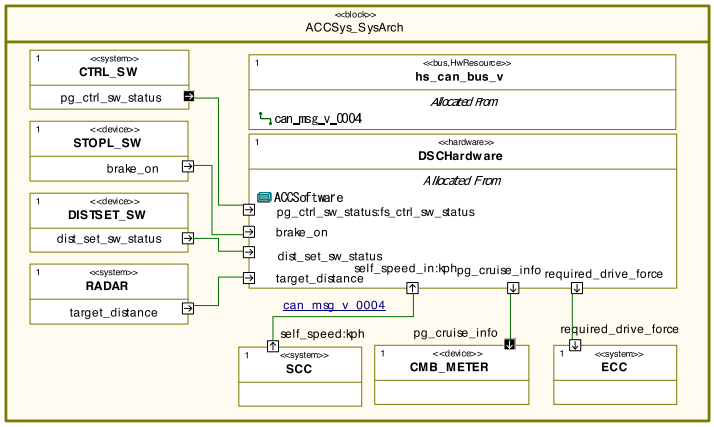
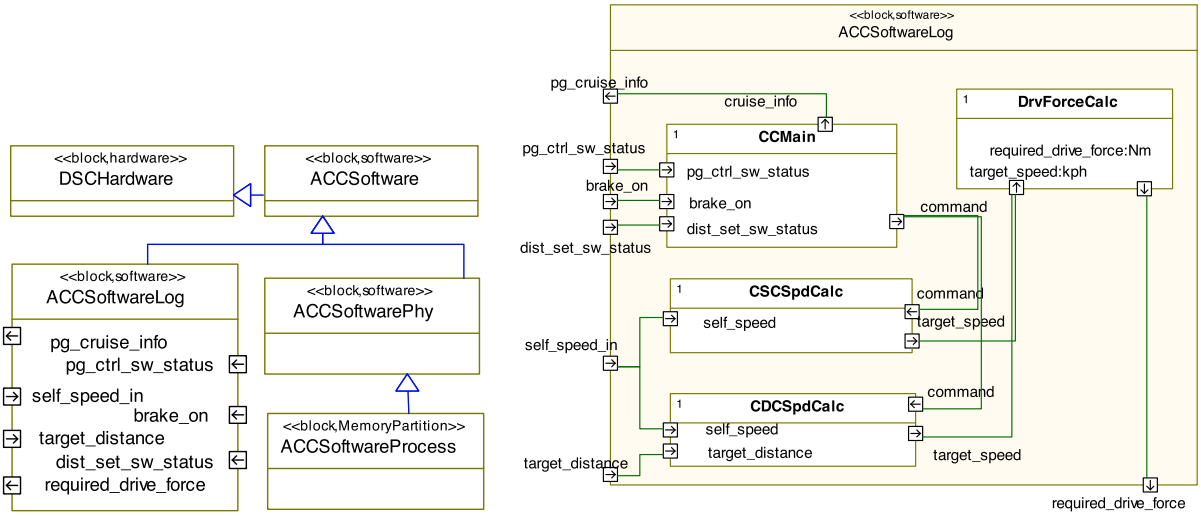


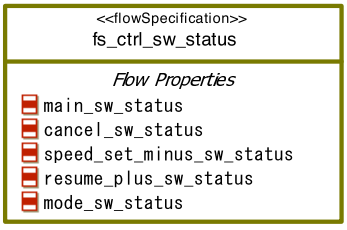
Fig. 11 SysML system architecture model in sysML (cf. Fig. 7).



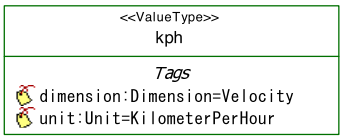
(a) Top-Level Logical and Physical Software Components.

(b) Logical Software Architecture Model (cf. Fig. 7).

Fig. 13 SysML logical software architecture models.



(a) Flow Specification.



(b) Value Type.

Fig. 12 SysML interface specifications.

This means that we can eventually obtain hierarchical architecture by simply repeating the same procedure.

3.4.1.2 Behavior Modeling

SysML provides several diagrams for behavior modeling such as the sequence diagram (SD), the activity diagram (ACT), and the state machine diagram (STM); which are inherited from UML. We cannot provide specific diagrams here due to space limitations; however, SysML obviously has strong capability of the system behavior modeling. In Section 3.3.1.2, we saw that AADL provides state-based behavior modeling aligning with architectural hierarchy by exploiting event chains. We can adopt the same approach in the case of STM of SysML.

3.4.2 Runtime Architecture Modeling

As shown in Fig. 4, SysML cannot describe the runtime architecture. On the other hand, MARTE is equipped with a sufficiently wide vocabulary of elements to allow runtime architecture modeling, e.g., ≪MemoryPartition≫ and ≪swSchedulableResource≫ represent processes and threads, respectively. Thus, here we use MARTE in a complementary way.

As has discussed in Section 3.3.1, the target software has a single-process architecture. That is, the top-level software component is a process that can be described by ≪MemoryPartition≫ of MARTE. The BDD model in Fig. 13 (a) shows that the specializing mechanism of SysML allows us to define the process of ACCSoftwareProcess via the physical software architecture (ACCSoftwarePhy) while inheriting all the required interfaces. Moreover, each thread within ACCSoftwareProcess can be modeled by ≪swSchedulableResource≫. Therefore, we can have a runtime architecture model as shown in **Fig. 14**, which is quite similar to the AADL model in Fig. 8.

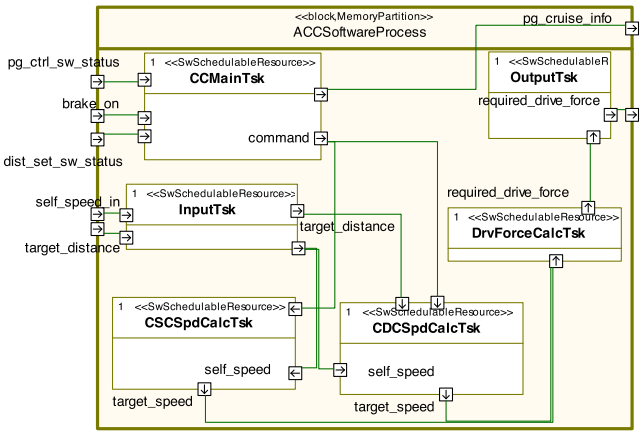
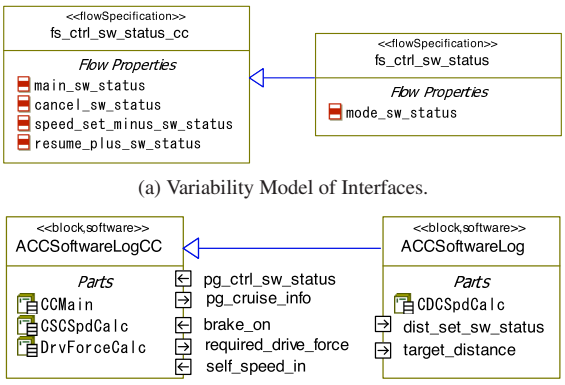


Fig. 14 MARTE runtime architecture model (cf. Fig. 11).



(a) Variability Model of Interfaces.

(b) Variability Model of Components.

Fig. 15 SysML/MARTE variability modeling.

3.4.2.1 Formal and Semi-Formal Analyses

Similar to the case of AADL in Section 3.3.2, we can add certain real-time properties. However, unlike the AADL case, there are no standard tool environments that can bring *formal* analyses based on these real-time properties. Therefore, if we need these kinds of analysis, we must translate the MARTE models into AADL models that are analyzable by OSATE. On the other hand, some tools provide *semi-formal* analyses of SysML models, namely, the simulation of SysML models. For example, the modeling tool of Rhapsody from IBM [*11] offers functionalities to simulate SD and STM models.

3.4.3 Code Generation

Similar to the method of Section 3.3.3, we can generate source code templates from the runtime architecture model shown in Fig. 14. Unfortunately, unlike in AADL, no code generation guidelines or automatic code generators are available. Therefore, hand coding that considers the target platform is necessary.

As is shown in Fig. 4, neither SysML nor MARTE can cover all the development phases of automotive systems by itself. However, the above discussion shows that the combined use of SysML and MARTE allows us to perform each step of model-based automotive system development.

4. Qualitative Comparison of ADL-Based Approaches

This section provides a qualitative comparison of the proposed two approaches by comparing the requirements in Section 2.1 and the case study in Sections 3.3 and 3.4. We have already acquired enough evidence to discuss the requirements (1)–(5) through the case study in Section 3. The upper half of Table 3 shows the comparison of the two approaches from the viewpoints of the requirements (1)–(5). The detailed comparison from each viewpoint has been described in the table. Thus, we will only provide a summary of the comparison in Section 4.3.

Besides the requirements (1)–(5), we also need to discuss the requirements: (6) large-scale product line and (7) high-level system assurance. However, we have not obtained appropriate evidence for these two requirements through our case study. Therefore, we will provide a supplemental case study below.

[*11] http://www-01.ibm.com/software/awdtools/rhapsody/

4.1 Variability Modeling

In order to discuss variability modeling, we need to prepare a suitable example of system variation. To this end, we define a cruise control system (CC system, hereafter) by simplifying the ACC system, namely, excluding the CDC function. Due to the simplification, some peripheral systems such as the mode setting switch and the radar sensor are removed in the CC system.

Approach #1 (AADL Case)

The variation of the specification of individual interfaces, *interface signature*, can be described by **`port group`** and **`extends`**. For example, Listing 10 shows that the interface of the ACC system to the control switches can be defined by adding an interface to the interface of the CC system. The added interface should be connected to the mode switch that is unnecessary in the CC system. Moreover, **`extends`** and **`refined to`** can be applied to the type definition of components (see the **`features`** part of *`ACCSoftwarePhy`* in Listing 4). In this case, we can add new interfaces or override existing interfaces by another specifications. That is, the variability of the combination of interfaces, *interface assembly*, can be modeled. When we apply **`extends`** and **`refined to`** to the implementation of components, the combination of subcomponents (*internal specification of components*) can be varied. In this case, we can add new subcomponents or override existing subcomponents (see *`ACCSW`* in Listing 4 for example). The same discussion on the AADL-based variability modeling can be found in our previous paper [20].

Listing 10: AADL Variability Modeling.

```
port group pg_receptacle_ctrl_sw_status_cc
  features -- Control Switches for CC
    main_sw_status: in event port;
    (snip)
end pg_receptacle_ctrl_sw_status_cc;
port group pg_receptacle_ctrl_sw_status
  extends pg_receptacle_ctrl_sw_status_cc
  features -- Add Mode Switch
    mode_sw_status: in event port;
    (snip)
end pg_receptacle_ctrl_sw_status;
```

Approach #2 (SysML / MARTE Case)

In the case of SysML and MARTE, we can exploit their inheritance mechanism. The variation of an interface signature can be described by specialization of ≪flowSpecification≫ as shown in **Fig. 15** (a). The interface assembly and the combination of subcomponents can be varied by using the inheritance again as shown in Fig. 15 (b). In the figure, interfaces: dist_set_sw_status and target_distance, and a component: CDCSpdCalc, which are not rele-

Table 2 Comparison of Variability Modeling Based on Approaches #1 and #2.

Comparison points		Approach #1: AADL	Approach #2: SysML / MARTE
External Specification	**Interface Signature**	**`Port group`** can be varied by **`extends`**. Interfaces can be added and overridden by **`refined to`**.	Interfaces can be added by the inheritance of ≪flowSpecification≫.
	Interface Assembly	Component type can be varied by **`extends`** and **`refined to`** (same as above).	Interfaces can be added by the inheritance of blocks.
Internal Specification		Subcomponents can be added by **`extends`** and overridden by **`refined to`**.	Subcomponents can be added by the inheritance of blocks.

Table 3 Comparison of Approaches #1 and #2.

Requirements	Sub-Requirements	Approach #1: AADL	Approach #2: SysML / MARTE	Relevant Sections
(1) Large-Scale Systems	Hierarchical Architecture	**`System`** components can have a hierarchical architecture.	≪Block≫ can have a hierarchical architecture.	Sects. 3.3.1 and 3.4.1
	Layered Architecture	**`Requires`**, **`provides`**, and **`access`** combine multiple layers.	*Allocation* combines multiple layers.	
	Component Specification	Component type definition provides formal interface specifications.	BDD, ≪flowSpecification≫, ≪flowProperty≫, and ≪ValueType≫ formally define interfaces.	
(2) Real-Time Systems	Real-Time Properties	Several properties such as deadline and execution time can be specified.		Sects. 3.3.2 and 3.4.2
	Formal Real-Time Analyses⋆	OSATE provides formal analyses, e.g., schedulability analysis, etc.	Not supported by a standard tool.	
(3) Embedded Systems	Software Architecture	Both logical and physical software architecture can be described.		Sects. 3.3 and 3.4
	Hardware Architecture (Execution Platform)	Several elements such as **`device`**, **`bus`**, and **`processor`** are available.	HRM (Hardware Resource Modeling) of MARTE is available.	Sects. 3.3.1 and 3.4.1
(4) Source Code Handling	Multiple Abstraction Levels	Lowest abstraction level is adjacent to the source code level.		Sects. 3.3.3 and 3.4.3
	Code Generation⋆	Automatic / Manual	Manual	
(5) Distributed Development	Open Standard	SAE Standard	OMG Standard	Sect. 2.2
	Seamless Refinement	By **`extends`** and **`refined to`**.	By the inheritance mechanism.	Sects. 3.3.2 and 3.4.2
(6) Large-Scale Product Line	*Variability Modeling*	Both external and internal specifications can be varied by **`extends`** and **`refined to`**.	Inheritance can vary both external and internal specifications in an *incremental* way.	Sect. 4.1 or more details in [20].
(7) High-Level System Assurance	Formal Notation	Specified by BNF (Backus-Naur-Form).	Specified by UML Metafile.	N/A
	Formal Verification⋆	Some kinds of formal verification, e.g., timeliness verification, are supported by OSATE.	Simulation-based (semi-formal) verification is supported by some tools.	Sects. 3.3.2 and 3.4.2
	Error Modeling	Supported by Error Model Annex [21].	Extension like EAST-ADL is necessary.	Sect. 4.2
(*) Behavior Modeling	Fundamentals	**`Mode`** gives simple state-based modeling.	Several diagrams, e.g., SD, ACT, and STM, are available.	Sects. 3.3.1.2 and 3.4.1.2
	Alignment with Architecture Models	Event propagation between state machines enables hierarchical and layered modeling.		

⋆These items are the requirements of supporting technologies (tools).

vant to A CCSoftwareLogCC, are newly installed in ACCSoftwareLog. However, unlike AADL, it is not easy to override existing interfaces or subcomponents [*12].

In order to summarize the above discussions, **Table 2** shows the comparison of the variability modeling based on both approaches.

4.2 Error Modeling

For the purpose of system assurance, dependability analysis is mandatory. Furthermore, from an architectural point of view, it is essential to clarify how component errors lead to hazardous events. This kind of analysis can be supported by error modeling techniques. In the case of AADL, the standardized annex for the error modeling [21] is available. In contrast, neither SysML nor MARTE are not yet ready for the error modeling, so that we need to extend them similarly to what EAST-ADL has done.

4.3 Comparison Summary

Table 3 summarizes the qualitative characteristics of the two approaches. The table shows that both approaches bring similar effects on each requirement except for the following items: code

[*12] This is possible from a language specification point of view; however, no tools support this functionality.

generation, formal verification, error modeling, and variability modeling. Based on these criteria, approach #1 (AADL) is superior to approach #2 (SysML and MARTE). Therefore, so far as these items are important to a given development project, AADL is a better candidate than SysML and MARTE. Otherwise, we can choose a suitable approach while considering the information literacy of the engineers who are involved in a project.

5. Conclusion and Future Work

In this paper we discussed ADL-based approaches to automotive system development. We derived two different kinds of approaches: (i) an AADL-based method, and (ii) a collaborative modeling method using SysML and MARTE. Detailed modeling steps of each approach were explained through modeling trials based on an ACC system. These trials also proved that both approaches can cover similar phases of automotive system development. Moreover, by comparing the two approaches from multiple viewpoints, we clarified the differences between the approaches, namely among the three ADLs. The comparison also showed that both approaches offer several different advantages to automotive system development.

While this paper mainly dealt with the architectural aspects, the behavioral aspect was discussed only briefly. Therefore, it is also necessary to perform a precise comparison from a behavioral angle as a future work.

Although this paper discussed only automotive system development, some of the discussion given in this paper possibly can be applied to other domains, e.g., medical devices, consumer electronics, etc. For example, some of the requirements of ADLs in Section2.1 can be common among these domains. Therefore, even in other domains, it might be possible to find practical ADL-based approaches by choosing common requirements and simply following the steps given in this paper.

References

[1] Charette, R.N.: This Car Runs on Code, *IEEE Spectrum*, Vol.46, No.2 (2009).

[2] Interim Report of the Society for the Study of Reliability and Security of Information Systems/Software in an Advanced Information Society, Technical Report, the Ministry of Economy, Trade and Industry (2009).

[3] Shiraishi, S. and Abe, M.: Automotive System Development Based on Collaborative Modeling Using Multiple ADLs, *ESEC/FSE 2011* (*Industial Track*) (2011).

[4] Medvidovic, N. and Taylor, R.: A classification and comparison framework for software architecture description languages, *IEEE Trans. Software Engineering*, Vol.26, No.1, pp.70–93 (online), DOI: 10.1109/32.825767 (2000).

[5] ISO 26262 Road vehicles – Functional safety, ISO (2011).

[6] Habli, I., Ibarra, I., Rivett, R. and Kelly, T.: Model-Based Assurance for Justifying Automotive Functional Safety, *2010 SAE World Congress*, No.10AE-0181, pp.1–16 (2010).

[7] Joshi, A., Vestal, S. and Binns, P.: Automatic Generation of Static Fault Trees from AADL Models, *DNS Workshop on Architecting Dependable Systems*, Edinburgh, Scotland (2007).

[8] *Architecture Analysis & Design Language* (*AADL*), AS5506A, SAE International (2009).

[9] Elloy, J.-P. and Simonot-Lion, F.: An Architecture Description Language for In-Vehicle Embedded System Development, *15th Triennial World Congress* (2002).

[10] Automotive Modelling Language (AML), The Project Automotive.

[11] COTRE (Real-Time Description Language), The Cotre Project (2004).

[12] EAST-ADL, ATESST2 (Advancing Traffic Efficiency and Safety through Software Technology) (2010).

[13] A UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE), Object Management Group (2009).

[14] Vestal, S.: *MetaH Programmer's Manual*, Honeywell Technology Center.

[15] OMG Systems Modeling Language (OMG SysML), Object Management Group (2010).

[16] Automotive Open System Architecture (AUTOSAR).

[17] Architecture Analysis & Design Language (AADL) Annex Volume 2, AS5506/2, SAE International (2011). The Behavior Annex.

[18] Delange, J., Pautet, L., Hugues, J. and de Niz, D.: An MDE-Based Process for the Design, Implementation and Validation of Safety-Critical Systems, *2010 15th IEEE International Conference on Engineering of Complex Computer Systems* (*ICECCS*), pp.319–324 (online), DOI: 10.1109/ICECCS.2010.12 (2010).

[19] Lemieux, J.: *Programming in the OSEK/VDX Environment*, CMP (2001).

[20] Shiraishi, S.: An AADL-Based Approach to Variability Modeling of Automotive Control Systems, *Model Driven Engineering Languages and Systems* (*MODELS2010*), Vol.I, Oslo, Norway, pp.346–360 (2010).

[21] Architecture Analysis & Design Language (AADL) Annex Volume 1, AS5506/1, SAE International (2011). Annex E: Error Model Annex.

Shin'ichi Shiraishi received his B.S., M.S., and Ph.D. degrees in Electronics Engineering from Hokkaido University, Japan, in 1997, 1999, and 2002, respectively. He is currently a Senior Researcher in Toyota InfoTechnology Center, U.S.A. Inc. His research interests include software assurance, software architecture, and modeling languages. He is a member of IEEE.