CHAPTER

8 DATA ANALYTICS IN SYSTEMS ENGINEERING FOR INTELLIGENT TRANSPORTATION SYSTEMS

Ethan T. McGee and John D. McGregor
Clemson University, Clemson, SC, United States

8.1 INTRODUCTION

Products ranging from everyday consumer devices to vehicles are being attached to the growing, interconnected web known as the Internet of Things (IoT). The IoT promises to provide new interaction schemes enabling "things" to report their status and have that status modified remotely over the Internet. For example, it is currently possible to view and modify the temperature of your home without being physically present inside. The IoT also promises to provide data that will augment the basic functionality of things. This promise is the basis for the movement to realize Intelligent Transportation Systems (ITSs), advanced systems providing services for traffic management/control among various transportation methods which better inform users and permit safer, coordinated use of transportation networks [1].

Connected Vehicles (CVs) are elements of an ITS. A CV is a vehicle capable of communicating with remote entities, such as other vehicles, roadside infrastructure, or traffic management centers. This communication allows the CV to have access to extra information enhancing the driving experience or allowing the driver to make better informed decisions. As an example, a driver wishes to go to a location across town; unfortunately, traffic is congested due to recent storms. The vehicle communicates the driver's location and chosen destination to a traffic management center in order to receive the best route. The shortest route is currently closed due to a weather-related accident blocking all lanes. The traffic management center calculates several alternate routes, selects one based on preferences set by the driver, and communicates that route to the vehicle. To fulfill its mission an ITS consumes/produces data at large scales, and it is necessary for the system to be engineered to support the management of real-time data at such scales.

System engineers are responsible for the analysis and design of entire systems; for ITSs, this includes accounting for the need of real-time analysis and performance while designing for other system attributes like safety and security [2]. These responsibilities are carried out in a variety of ways. For example, systems engineers are responsible for determining the data flow paths between system components. This involves both ensuring that information can be received from all necessary sources and ensuring that the information arrives in a timely manner. A systems engineer might accomplish this task using tools to analyze the system or more commonly a model of the system potentially derived from a reference architecture, a domain-specific architecture solution

Data Analytics for Intelligent Transportation Systems. DOI: http://dx.doi.org/10.1016/B978-0-12-809715-1.00008-0

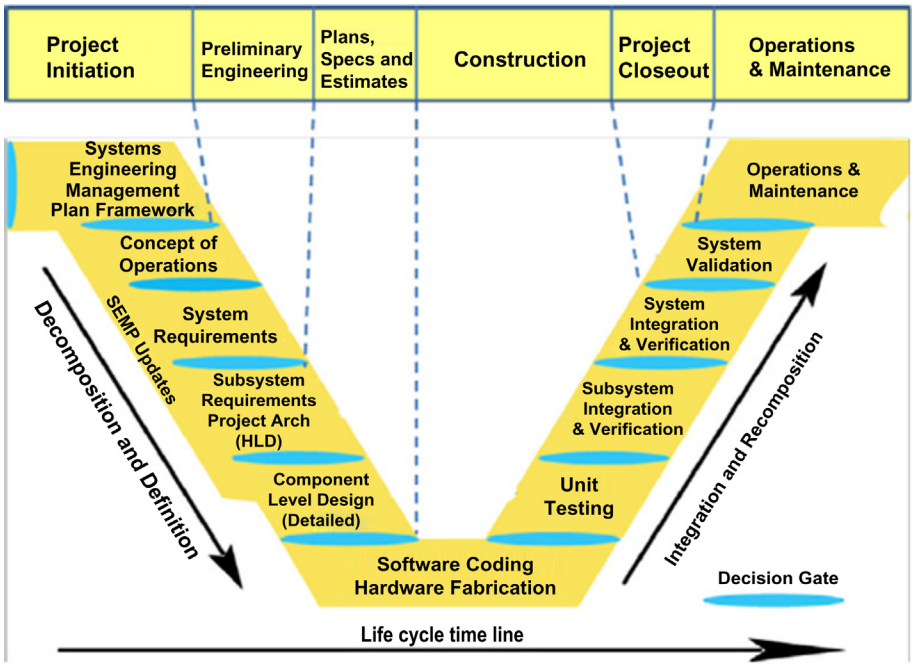

FIGURE 8.1

CVRIA and systems engineering [3].

emphasizing the common vocabulary of the domain. Whatever the tasks involved, the system engineer defines and executes a process to achieve the goals of the system.

The systems engineering process is heavily influenced by the domain of application. Fig. 8.1 shows the correspondence between an ITS-specific process including CV applications and the traditional systems engineering process. The ITS blends a number of disciplines including distributed computing, mobile computing, wi-fi/cellular communication technologies, and security protocols. The systems engineer also considers the constraints imposed on system engineering by the nature of an ITS. In the next section, we survey information needed as a background before presenting the data analysis-focused systems development scenario. In the development scenario, we will introduce requirements and map those requirements to an Architecture Description Language (ADL) showing how the ADL supports verification and analysis activities on the modeled system. Finally, we summarize and suggest directions that future research will take.

8.2 BACKGROUND

8.2.1 SYSTEMS DEVELOPMENT V MODEL

The Systems Development V Model, pictured in Fig. 8.1, represents a standard development workflow consisting of multiple phases representing conceptually different actions. The diagram

illustrates functional dependencies; that is, one activity comes after another because the second activity depends in some way on the actions of the previous activity. However, the V diagram does not illustrate a strict chronological order. For example, designing to meet a set of requirements may result in the identification of new requirements, which should be analyzed prior to further design work.

Broadly speaking, the system, or some portion of it, is first conceptualized and then engineering work is performed. Plans, specifications, and design are formed from the initial engineering work. These plans and designs are funneled into the construction process during which the system is realized. After realization, the system is tested to ensure conformance to the specified design, and if passed, the system is deployed. Finally, maintenance and upgrades are periodically executed as needed. We now illustrate each phase of the system engineering process in greater detail using the Advanced Automatic Crash Notification Relay application defined in CVRIA as an example.

8.2.1.1 Project initiation

In this phase, the benefits the system will produce are weighed against the costs of building the system to determine if building the system is both wise and feasible. This phase would also involve determining if the system to be built is the best response to the need.

An example of this deliberation might be as follows. Consider a stretch of roadway that does not have many travelers, however, despite the low usage, the roadway sees an inordinate number of severe accidents. Due to the low frequency of travel, it is sometimes several hours before an accident is reported to medical personnel. The local county government discusses instrumenting an Advanced Automatic Crash Notification Relay system on that stretch of roadway to detect accidents and immediately dispatch emergency personnel should they occur. The system will cost over one million dollars to construct over the course of six months. Consulting medical evidence, the planners see that more timely treatment would drastically reduce the number of fatalities. The county council and state government agree that the instrumentation would save lives. It is decided that the cost will be split (30/70) between the county and state governments and the system is commissioned.

8.2.1.2 Preliminary engineering

During preliminary engineering, the requirements for the system and the concept of how the system is to operate are decided. The requirements in particular are important as they will constrain the possible decisions available to engineers in later phases. For example, once again consider our stretch of an infrequently traveled, accident-prone roadway. It is not possible for the notification system to be connected to the power grid due to the system's isolated location. Therefore, each sensor will have to harvest its own electricity via solar, geothermal, wind, or some other means. This also means that the power available for sending notifications will be limited so high-power radio transmission systems cannot be used. This will prevent certain cost-saving decisions, such as using fewer sensors or high-power radios, from being options later.

8.2.1.3 Plans, specifications, and estimates

This phase turns the requirements into concrete models that are analyzed to ensure conformance. The models show the various properties of the system under consideration, for example, latency of communication channels. During this step, the models are verified to ensure that they satisfy the requirements, and plans for testing and deploying the system are also created.

Going back to our example, at the current time, we know that we are forced to use a low-power transmission radio, specifically an XBee 2 mW Wire Radio. These radios have a limited range, approximately 120 m [4]. It will therefore be necessary to have beacons at least every 120 m to relay a message to the traffic management center. In order to account for possible interference, it is decided that a beacon will be placed every 90 m instead.

8.2.1.4 Construction

During this phase, the hardware and software elements of the system models are created or acquired. Testing is also done to ensure that the implementations conform to the model's specifications. Implementations not conforming to models will have to be changed to ensure the desired behavior is achieved. As the system components are completed, successively larger integrations of components are tested together to ensure that the combination of separately developed components cannot cause the system to fail.

For our example, the collision detection hardware is created/acquired and programmed during this phase. Pilot deployments of the system might also be done in order to collect small quantities of real-world data that can be used for testing.

8.2.1.5 Project closeout

Project closeout sees one final round of testing before the stakeholders approve the project for full installation. Once approved, hardware/software are configured and the project is deployed to its operational environment following the previously created plans. Responsibility now shifts from the development staff to operations.

8.2.1.6 Operations and maintenance

The final phase of the development process is to maintain the system and ensure that it is operating successfully. In some situations, the deployed system must be tested on a regular basis to ensure that it is still operating successfully. Any components of the system not operating correctly would need to be replaced in order to bring the system back into compliance.

In a relatively static business environment, the requirements and architecture might be studied and reworked until they were relatively robust and the implementation generated from them could be considered sufficiently correct. In this case, other than direct coding errors there would be no issues with the implementation. However, in current practice, the business environment is very dynamic and moves too quickly for this method to be feasible. Conditions change and so do requirements. The majority of errors introduced into a system come from gaps between what is given in the requirements and architecture and what is actually needed.

There are many possible explanations for this gap. Whatever the reason, it is advantageous to correct these errors in the requirements and architecture as early as possible before they are propagated to the implementation and testing phases, where correcting the errors is far costlier. One possible approach for quick detection and correction of such inconsistencies is Continuous Engineering.

8.2.2 CONTINUOUS ENGINEERING

Continuous Engineering [5] is a system development approach leveraging the V Model for System Engineering. In this approach, systems are treated as entities that are in constant flux, adapting to

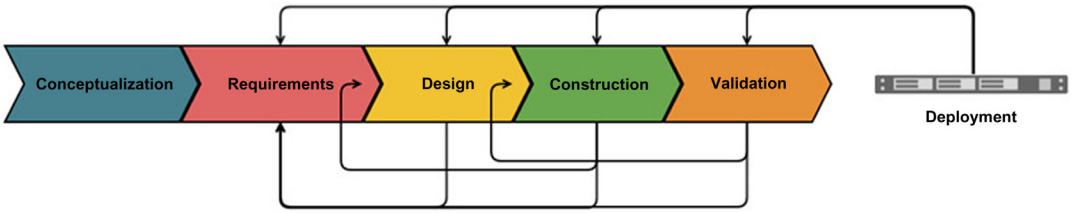

FIGURE 8.2

Continuous engineering pipeline.

the world around them. Continuous Engineering uses the flow of data produced by some of the billions of devices that comprise the IoT in order to fine-tune requirements and business decisions for a product or product family. The goal is to use the data generated by these devices to validate decisions as early as possible during the development of a system. The data is also used as an additional input into the Requirements/Design phase. This allows systems to react to sudden changes in usage or to take advantage of new data streams. This is of particular importance to the domain of ITSs. ITS is an emerging field relying on limited infrastructure and incomplete data inputs. As time advances, new data sources will be deployed and new information will be available for collection. System developers want the created systems to be capable of taking advantage of new data sets and information as they are made available rather than having to wait for extended periods before the data streams can be utilized. To put it another way, we want our systems to continuously evolve based on the flow of data into our systems.

This is accomplished as is shown in Fig. 8.2. During the development of a system, data is generated in each phase. With traditional development processes, the flow of data is always forward. Continuous Engineering takes an approach similar to that of the Agile development community. Data now flows in many directions during development and deployment, not just forward.

Continuous Engineering is also supported by specific design decisions. For example, using a bus design to connect system components allows additional connections without changing component interfaces. Other designs such as publish and subscribe also support evolution with minimal disruption.

We also collect data from our deployed systems which feeds into every phase of the development life cycle. This allows the development process to remain knowledgeable about the repercussions of decisions that have been made so that they can be adjusted quickly if necessary.

8.2.3 AADL

The Architecture Analysis & Design Language (AADL) is an architecture modeling language that uses a common description language to bind together both hardware and software elements into a unified model [6]. The language is a Society of Automotive Engineers (SAE) standard, and is actively maintained, at the time of this writing being in its second revision. AADL features a robust type system, and the language can be extended through the use of annexes, each of which can have their own independent syntax and functionality and each of which is standardized independently.

8.2.3.1 Language overview

AADL is a description language used to formally represent architectures. The language has keyword constructs for representing hardware (*device*, *processor*, *bus*, *memory*), software (*process*, *thread*, *subprogram*), and the integration of the two (*system*). The language is particularly suited for a Cyber-Physical System (CPS), systems where the hardware is directly controlled and monitored via software. At its highest level, AADL allows for the intermixing of hardware and software elements within the *system* construct. It is also possible to create *abstract* components that can be refined into either hardware or software at a later time. This is beneficial in the system engineering process for an evolving domain such as ITS where you know that a specific component is needed, but you are unsure whether this component will be implemented and maintained as a hardware or software artifact. The component and its functionality can be declared and used in the architecture as an abstraction. Then, once a decision is made, it can be refined into either a hardware or software component. An example system in AADL can be seen in Fig. 8.3.

In Fig. 8.3, we have a description/representation of an ITS CPS that posts an event if the distance between the vehicle to which the system is attached and another vehicle falls below some threshold. Such a device might be used on a vehicle to determine if the vehicle is following another vehicle too closely. The system described has one output, an alert named *too_close_alert*.

The system has two subcomponents, a distance sensor named *sensor* and a distance evaluator named *evaluator*. The *connections* section of the system defines how the subcomponents are connected together. In this example, the distance value of the sensor is connected to the input of the evaluator. The system's alert value is also directly connected to the alert value of the evaluator.

AADL components are split into two separate definitions. The first is a specification of the features, inputs and outputs, that a component consumes and produces respectively. The complete set of features is referred to as a contract. The contract of Fig. 8.3 is shown in Fig. 8.4.

The second definition is an implementation of the contract. Contract implementations are represented by using the *implementation* keyword as well as providing a name for the implementation which comprises two pieces. The first piece is the name of the contract implemented, and the second piece is a unique identifier for the implementation. The two are concatenated using a single

```
system proximity_alert_system
   features
      too_close_alert: out event port;
end proximity_alert_system;

system implementation proximity_alert_system.impl
   subcomponents
      sensor: device distance_sensor.impl;
      evaluator: process distance_evaluator.impl;
   connections
      sensor_to_processor: port sensor.distance -> evaluator.distance;
      processor_out: port evaluator.too_close_alert -> too_close_alert;
end proximity_alert_system;
```

FIGURE 8.3

AADL snippet.

```
system proximity_alert_system
    features
        too_close_alert: out event port; end
proximity_alert_system;
```

FIGURE 8.4

AADL contract snippet.

```
system implementation proximity_alert_system.impl
    subcomponents
        sensor: device distance_sensor.impl;
        evaluator: process distance_evaluator.impl;
    connections
        sensor_to_processor: port sensor.distance -> evaluator.distance;
        processor_out: port evaluator.too_close_alert -> too_close_alert;
end proximity_alert_system;
```

FIGURE 8.5

AADL contract implementation snippet.

dot in the form $<$*contract name*$>$.$<$*unique identifier*$>$. The implementation of the contract of Fig. 8.4 is shown in Fig. 8.5. It is possible and common for contracts to have more than one implementation. It is also common for the internal components of each implementation to vary drastically from one another.

AADL's functionality is augmented through the language's provision for extensions, or annexes. An annex in AADL provides additional functionality not native to the core language. They are generally maintained by outside groups that have a vested interest in the AADL language, although some annexes are published and maintained by the language maintainers. We will provide an overview of four annexes to the AADL language. The first annex we will cover is the behavior annex.

8.2.3.2 Behavior annex

The behavior annex [7] allows users to specify how a component reacts under normal circumstances. Let's return to our example of a distance sensor for a moment. From our previous AADL snippet, we don't know how our evaluator should act. We know that it will receive an input value, *distance*. If that input value should fall below a specific threshold, an event should be fired. In Fig. 8.6, we show the description for this behavior along with the associated behavior specification.

The behavior annex defines a component's reactions to inputs via a finite state machine syntax. The user defines a set of internal variables which represent the data that describe the component. These variables maintain their value between transitions, and unless they are deliberately modified, they do not change their value.

The user also defines a set of states. A state defined in the state machine of the behavior annex has four primary attributes: initial, complete, final, and "normal." Each state is defined to have one or more of these attributes. The initial state is analogous to the start state of a finite state machine. It is the state into which the machine is initially placed when the component is instantiated.

```
process implementation distance_evaluator.impl
   annex behavior_specification {**
      variables
         min_distance: int;
      states
         start: initial state;
         ready: complete state;
         calculating: state;
      transitions
         start -[]-> ready { min_distance := 5 };
         ready -[on dispatch]-> calculating;
         calculating -[distance < min_distance]-> ready { too_close_alert! };
         calculating -[distance >= min_distance]-> ready;
      **};
end distance_evaluator.impl;
```

FIGURE 8.6

AADL behavior annex snippet.

In Fig. 8.6, the initial state is labeled as the *start* state. A complete state can be thought of as a yield state, in which the component has completed its work for the time being and is suspended until an event or input causes it to begin execution anew. A final state is a state at which execution has completed and the component "turns off." Normal states are simply intermediary states between any of the other three primary state types. A behavior description may contain many states with the complete and final attributes but only one state may be the initial state. Let us consider Fig. 8.6 as an example. From the *start* state to the *ready* state, there is an unconditional transition, meaning as soon as we enter the *start* state, we immediately transition to the *ready* state. However, this transition also has a side effect indicated by the expression in curly braces to the right of the transition. In this case, the *min_distance* variable is set to 5. From the *ready* state, there is a single conditional transition. The keywords *on dispatch* here mean an input received. We transition from *ready* to *calculating* whenever we receive input. The final two transitions are also conditional, but they are based on the value received from the distance sensor. If the value from the distance sensor is less than our minimum distance, we transition back to the *ready* state, but we, as a side effect, fire the *too_close_alert* event. If the value is equal or greater, we simply transition back to the *ready* state with no side effects.

8.2.3.3 Error annex

The error annex [8] of AADL allows the specification of how components react under abnormal conditions. It also allows specification of which types of errors are anticipated, and if those errors are handled within the component or if they are propagated to connected components. The error annex ships with a structured set of predefined errors types (referred to as an ontology), shown in Section 8.6. These existing types can be aliased or extended to create user-defined error types. The large number of error types in the ontology provides a rich set of considerations when evaluating a system for potential hazards. For example, given a system, an engineer could walk throughout the ontology asking if a specific error or error type from the ontology could be encountered by the system. Each error that could be encountered should be listed in the error annex so that the behavior of the system when encountering that error can be specified.

```
process implementation distance_evaluator.impl
   annex EMV2 {**
      use types ErrorLibrary;
      use behavior EvaluatorBehavior;
      error propagations
         distance: in propagation {OutOfRange, StuckValue};
         too_close_alert: out propagation {StuckValue};
      end propagations;
      component error behavior
         transitions
            normal -[distance{OutOfRange}]-> transient_failure;
            normal -[distance{StuckValue}]-> full_failure;
         propagations
            full_failure -[]-> too_close_alert;
      end component;
      **};
end distance_evaluator.impl;
```

FIGURE 8.7

AADL error annex snippet.

Let us once again consider the distance evaluator. We will assume that the sensor which sends us a value can encounter two types of errors. It could become stuck, perhaps hitting a floor or a ceiling, never being able to update its value. Or, it could send us a value that is out of the range of possible or expected values, say a negative value. In the case of a negative value, we will "fail," but we can recover as soon as the value is updated. In the case of a stuck value, we will need to permanently fail because something has happened from which we cannot recover. If this were to occur, it is likely that the sensor will need to be cleaned or replaced in order for normal operations to resume. In either case, the system will be powered down and put through a hard reset, so we will fail and not provide an option for recovery. A snippet of the AADL error annex, EMV2, is shown in Fig. 8.7 which models the error behavior described above.

Figure 8.7 references an error type library which defines the standard ontology including the two specific errors we mentioned previously, *StuckValue* and *OutOfRange*. We also define which errors are propagated to us, given by the in propagation. In the distance evaluator, both of the error types we can encounter are propagated to us by the distance sensor. The distance evaluator can handle an *OutOfRange* error so we do not propagate it, but we cannot handle the *StuckValue* error so it is propagated. The transitions section of the error annex is similar to the specification of the behavior annex in that it is also a finite state machine. However, this is only a partial definition of the state machine. The rest of the machine is imported by the *use behavior* statement and is shown in Fig. 8.8. The primary machine is defined outside of the component so that it can be reused in other components. Notice, however, that components can introduce their own transitions. In our case, the distance evaluator introduces two, one for the occurrence of an *OutOfRange* error and one for the occurrence of a *StuckValue* error.

8.2.3.4 AGREE

Even when specifying the normal behavior and abnormal behavior of the model, it is still necessary to perform verification and validation activities. AADL facilitates these activities in multiple ways.

```
annex EMV2 {**
   error behavior EvaluatorBehavior
      events
         failure: error event;
         self_recover: recover event;
      states
         normal: initial state;
         transient_failure: state;
         full_failure: state;
      transitions
         normal -[failure]-> (transient_failure with 0.9, full_failure with 0.1);
         transient_failure -[self_recover]-> normal;
   end behavior;
**};
```

FIGURE 8.8

AADL error annex behavior snippet.

Its strong syntax and typing have allowed for the creation of multiple simulators that can be used to confirm the architecture. There are two annexes, AGREE and Resolute, dedicated to this purpose.

AGREE is a compositional verification tool that can be used to confirm the behavior of a component [9]. It follows the popular assume-guarantee reasoning model, which states that provided the assumptions about a component's inputs are met the component can provide certain guarantees about its output. AGREE works by using the model to construct a state machine that is fed into a Satisfiability Modulo Theorem (SMT) prover which confirms that the state machine, given the assumptions the contract makes, can produce the port values guaranteed by the contract. As it is a compositional verification tool, it can also be used to ensure that all subcomponents of a component correctly contribute towards the parent component's goal. The annex utilizes the fact that AADL components are split between a contract and contract implementation. A component's assume-guarantee contracts are attached to the component contract. The component's AGREE model is then placed in the contract implementation. It is necessary to specify the AGREE model even if you are using the behavior annex as AGREE and the behavior annex are not compatible with one another. It is also necessary to install a third-party SMT solver as one is not currently prepackaged with the tool.

As an example, let's consider the distance sensor device. The particular sensor that we are using has a maximum range of 25 m. Any values beyond that limit are likely erroneous and distance cannot be negative. We will therefore provide a guarantee that our device will return a value between 0 and 25. A snippet of AGREE that matches this description is shown in Fig. 8.9. In this case the error model could handle an *OutOfRange* error by ignoring it because we will have a sufficient gap at that point.

8.2.3.5 Resolute

In addition to performing compositionality checking on the architectural model, AADL also permits checking the structure of the architectural model through the use of the Resolute annex [10]. Unlike AGREE, Resolute is structured as claim functions that take parameter(s) then return whether

```
device distance_sensor
   features
      distance: out data port Base_Types::Integer;
   annex agree {**
      guarantee "the output distance is between 0 and 25":
         distance >= 0 and distance <= 25;
   **};
end distance_sensor;

device implementation distance_sensor.impl
   annex agree {**
      assert distance = if distance < 0 then
                           0
                        else
                           if distance > 25 then
                              25
                           else
                              distance;
   **};
end distance_sensor.impl;
```

FIGURE 8.9

AADL AGREE annex snippet.

```
annex resolute {**
    Req1(self : component) <= **"all threads should have measure of meters"**
       forall(x : union({self}, subcomponents(self))) .
          if (x instanceof device) then
             property(x, DistanceMeasure::Measure, "<not set>") = "meters"
          else
             true
*};
device implementation distance_sensor.impl
   properties
      DistanceMeasure::Measure => "meters";
   annex resolute {**
      prove(Req1(this))
   **};
end distance_sensor.impl;
```

FIGURE 8.10

AADL resolute annex snippet.

the claim is true. This might be used, for example, to ensure that every thread of the architecture has a dispatch protocol and, if appropriate, a period defined. Also, unlike AGREE, the implementation of Resolute does not currently rely on external dependencies.

Returning to our proximity alerting system, it would be problematic if multiple devices within our project used different units of measure. We will use Resolute to ensure that all devices report distance in meters. An example of Resolute is shown in Fig. 8.10.

In Fig. 8.10, we first define a claim function that collects all of the subcomponents of the given component. Since we only care about ensuring that our devices use meters, our claim limits the property check to only devices. Note that the method for invoking a claim on a component involves passing the claim function and the parameters you are using for the claim function to the *prove* function. Prove calls must be used in a contract implementation.

8.3 DEVELOPMENT SCENARIO

Systems in a CV produce large quantities of data some of which is consumed internally for immediate decision making and some of which is communicated to external resources for longer term planning and sharing. The vehicle also consumes data from roadside units and other vehicles for its decision making. In this section, we illustrate the use of the Connected Vehicle Reference Implementation Architecture (CVRIA) [11], which provides a description of high-level interactions and data flows among the significant pieces of a CV application, in designing applications for CVs. An in-depth examination of CVRIA is beyond the scope of this chapter, but we will illustrate its role in the systems engineering of CVs.

For the purposes of this scenario, we assume the perspective of a systems engineer who works for an Original Equipment Manufacturer (OEM) and is working on the autonomous navigation module sited in the vehicle's On Board Equipment (OBE) as defined in CVRIA. The engineer needs to understand *how* the data being produced by the vehicle is consumed internally, *what* of that data is related to external sources, and *where* it is consumed. The engineer uses flows defined in CVRIA to obtain an architecture view that includes the vehicle and its environment. Data is collected and aggregated by roadside units (RSUs) and exchanged with the vehicle via a Dedicated Short-Range Communication (DSRC) radio.

8.3.1 DATA ANALYTICS IN ARCHITECTURE

Architectures for CV applications encompass one or more mobile computing units and possibly one or more fixed computing units. A CV is considered a mobile computing unit with tens to hundreds of processors attached to multiple buses. The vehicle is also a complex CPS with numerous sensors to detect environmental conditions, such as weather, and physical elements, such as surrounding vehicles and roadway obstacles. A fixed computing unit would be any unit the vehicle communicates with which does not physically move. Examples would include RUs, traffic management centers, or servers within the IoT cloud.

The data flows between mobile computing units and mobile computing units/fixed computing units are complex and potentially numerous. If we consider CVRIA, there are tens of applications that a CV could implement. Each of these applications contains multiple data flows with each flow encompassing a unique set of data. Each flow is also associated with some computation. The system engineer must decide which computations are done on-board the CV and which are done remotely along with the value of the flow's required properties, such as latency and security.

One of the concerns of a system engineer faces when deciding where to locate computations is the latency involved in getting the data necessary for the computation to the processor and the

power of the processor. For example, computations done on-board the vehicle will have a relatively low latency, but the processing power of computing units on board the vehicle will likely not be as powerful as computing units in the IoT cloud. If extra processing power is needed, the data might be communicated over wireless to RUs which will in turn communicate the data to cloud-based processing units. However, hundreds of vehicles could be communicating with the same RU meaning that latency would be significantly increased. It is the responsibility of the system engineer to understand these trade-offs and their effect on data processing. AADL provides a means whereby latencies for individual links can be estimated, captured, and analyzed giving the engineer a method of determining where to locate a computation even before system development begins. As processes are implemented latency estimates for links are replaced with actual latencies.

8.3.2 THE SCENARIO

We will now select one of the CVRIA's applications and design it using AADL and its annexes, introduced in Section 8.2. The application we have chosen for this example is Do Not Pass Warning (DNPW) [12], in the CVRIA safety applications subarea. DNPW is designed to alert the driver that passing should not be attempted. This could be due to the passing area being occupied by another vehicle, or it could be due to road rules which prevent passing in the geographic area within which the vehicle is currently located. The system should be proactive, alerting the driver that passing should not be attempted even if the driver has not requested that a pass be attempted. The reference architecture from CVRIA for DNPW is shown in Fig. 8.11.

From the reference architecture, in Fig. 8.11, we can see that our concrete system architecture will need four primary entities, among others. The first entity is the driver, the agent in control of the vehicle. The second type of entity is the OBE of the vehicle. Among this equipment should be an entity that is responsible for the safety of the vehicle and its occupants. This entity will be responsible for deciding if passing would result in a safety-compromised scenario, such as the vehicle crashing with the vehicle in the opposing lane or losing control. The third type of entity would be remote vehicles that are connected to our vehicle as well as others. The remote vehicles will likely not provide the same granularity of information that the current vehicle does. For example, the safety system of the local vehicle will likely be able to determine the temperature of the engine or wear of the tires. It is unlikely that this information will be broadcast by the remote vehicles. The final type of entity would provide communication among the different systems of the local vehicle. Due to the evolving nature of these systems a data bus is a good choice. For our purposes, we will assume that the bus is a CAN Bus, a common bus type among automobile manufacturers.

CVRIA includes many high-level goals and requirements for each application for which a reference architecture is provided [14]. From the high-level goals/requirements given by CVRIA, we can determine some of the sensors that will be needed. We will need a sensor for determining the current lane the vehicle is traveling in (from 3.036), and a sensor, such as a GPS, for determining the current speed and direction of travel is necessary (from 3.037). We also need a method for determining the vehicle's current location (from 3.037), and a method of broadcasting and receiving messages, such as a DSRC, from nearby vehicles will be required (from 3.217). Finally, since vehicles will be communicating with one another, a method of identifying vehicles is

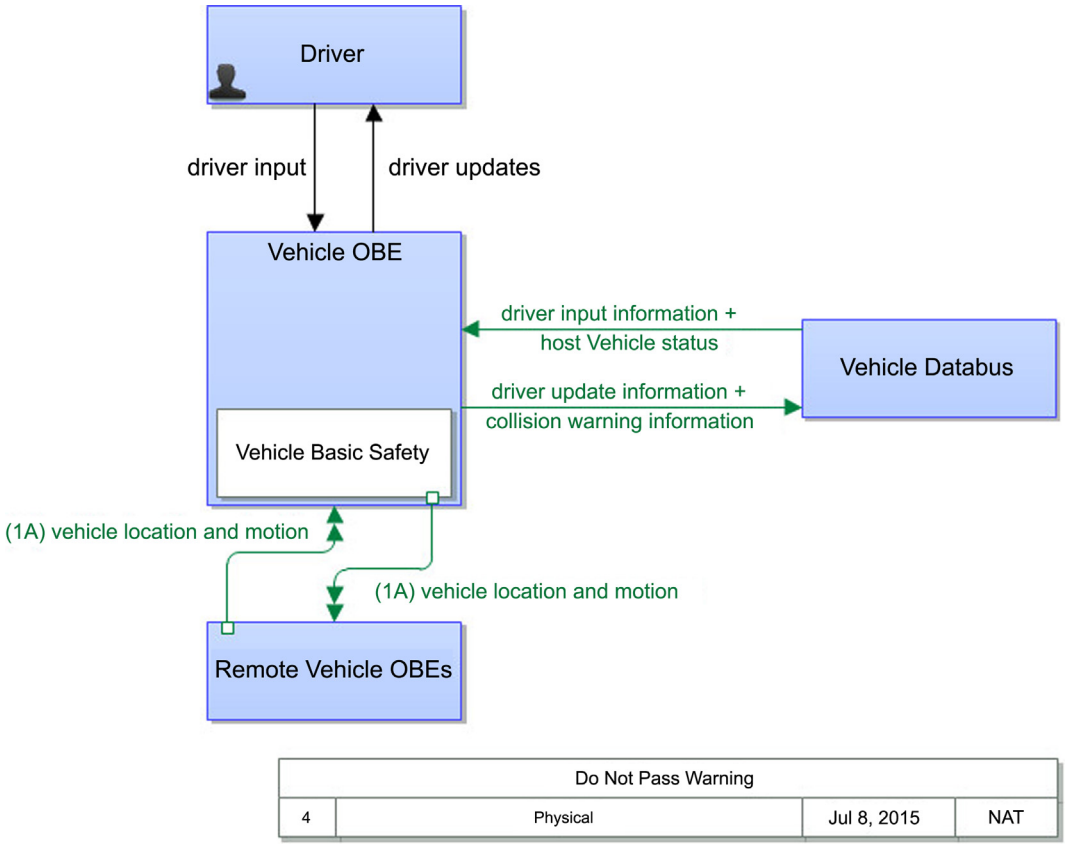

FIGURE 8.11

CVRIA DNPW [13].

needed. For simplicity, we will assume that each vehicle has a device that has been flashed with a Universally Unique Identifier (UUID). We will assume, for now, that the geometry of the passing area will be received from roadside infrastructure units, RUs. We will also assume that given the location of nearby vehicles, we can determine if the passing zone is occupied or will be occupied before we can safely pass.

Using this information, we create a high-level architecture of the DNPW system by replacing the abstract entities with actual entities. A more fully specified architecture will be created as we generate more specific requirements. We will now begin introducing specific requirements that contribute to the high-level requirements and goals of the DNPW system. In doing so, we will demonstrate how AADL can be used to realize those requirements in design, enabling the requirements to be verified and validated well before implementation. By catching errors as early as possible, the errors can also be corrected quickly before the errors become deeply integrated into the system and are more expensive to correct.

Our first requirement will center around the collision detector, and we will use the AADL behavior annex to implement the requirement. When passing, it is advised to increase the speed of the vehicle and complete the passing operation as quickly as possible. We want to fire a collision imminent event if we are passing and the user requests a decrease in speed before the operation completes. We also want to fire a collision imminent event if we are decreasing speed and the users requests that a pass operation commence. The behavior annex model demonstrating the implementation of this requirement is shown in Fig. 8.12.

In Fig. 8.12, we define a system named *collision_detector* that accepts two parameters, a speed and a lane data type, and two events, a request pass and request speed change event. The component is also capable of firing a single event which is meant to trigger an alert to the driver that a collision

```
system collision_detector
   features
      lane: in event data port Base_Types::Integer;
      speed: in event data port Base_Types::Integer;
      request_pass: in event port;
      request_speed_change: in event port;
      collision_imminent: out event port;
   annex behavior_specification {**
      variables
         is_passing: Base_Types::Boolean;
         is_changing_speed: Base_Types::Boolean;
      states
         start: initial state;
         ready: complete state;
      transitions
         start -[]-> ready;
            ready -[on dispatch request_pass]-> ready {
               is_passing := true;
               if (is_changing_speed)
                  collision_imminent!
               end if
            };
            ready -[on dispatch request_speed_change]-> ready {
               is_changing_speed := true;
               if (is_passing)
                  collision_imminent!
               end if
            };
            ready -[on dispatch speed]-> ready {
               is_changing_speed := false;
            };
            ready -[on dispatch lane]-> ready {
               is_passing := false
            };
   **};
end collision_detector;
```

FIGURE 8.12

Collision detection behavior.

is about to occur unless they take corrective action. The segment of an annex model shown in the figure defines the behavior described in the preceding paragraph. On receipt of a pass request or speed change request the vehicle will enter into a passing or speed changing mode respectively. If pass request is received in speed changing mode or a speed change is received in pass mode, the collision imminent event will fire as denoted by *collision_imminent!*.

The second requirement we will tackle centers around the speed sensor. The speed sensor will operate by taking the number of times the wheel rotates per second and, using the circumference of the wheel (in centimeters), the speed will be calculated. Since the wheel cannot rotate a negative amount, we will assume that the rotations per minute is always greater than or equal to zero, and provided that assumption is true, we can guarantee that the speed will always be greater than or equal to zero. The speed of the vehicle is also capped at a maximum of approximately 143 km/h, so we will also assume the maximum number of tire rotations per second is 110 rotations.

In Fig. 8.13, we have defined a simple AGREE contract and behavior specification. The circumference of the wheel in centimeters is multiplied by the rotations per second of the tire to obtain the number of centimeters traveled in one second. This value is multiplied by 3600 to convert to centimeters per hour which is converted to kilometers per hour before being returned by the sensor.

While this showcases AGREE used in the development scenario, the true power of AGREE comes largely into play later, when the individual components are composed into the system. For example, the architect of the speed sensor is using kilometers per hour as his measure of speed. Another architect designing a subsystem dependent on the speed sensor uses miles per hour as his measure of speed. When the two subsystems are composed, the range of values produced by the speed sensor (0−143) will greatly exceed the range of values expected by the consumer (0−89).

```
device speed_sensor
   features
      rotations_per_second: in data port Base_Types::Integer;
      speed: out data port Base_Types::Integer;
   annex agree {**
      assume "rotations_per_second between 0 and 110":
         rotations_per_second >= 0 and rotations_per_second <= 110;
      guarantee "speed >= 0": speed >= 0;
   **};
end speed_sensor;

device implementation speed_sensor.impl
   annex agree {**
      eq circumference: int = 360;
      eq centimeters_per_hour = circumference * rotations_per_second * 3600;
      eq meters_per_hour = centimeters_per_hour / 1000;
      eq kilometers_per_hour = meters_per_hour / 1000;
      assert speed = kilometers_per_hour;
   **};
end speed_sensor.impl;
```

FIGURE 8.13

Speed sensor AGREE.

When AGREE is executed against the composed system, the assumptions of the subsystem using miles per hour will be violated allowing the discrepancy to be found and corrected before the individual components are implemented.

Third, we'll tackle a slightly different requirement concerning the structure of our architecture. Due to the large number of sensors on board the vehicle, the power draw of the vehicle may be an issue. We want to ensure that total power draw of all the sensors attached to a vehicle does not exceed the threshold of the power capacity of either the battery or the alternator. Doing so would cause the battery to drain significantly faster resulting in both the battery and alternator wearing out more quickly. Resolute is designed for handling structural verification of components so we will use it in this part of the scenario.

In Fig. 8.14, we provide the definition for a library and two components. The resolute library defines a predicate that takes a system and the maximum power available for drawing. The method in the library collects all of the subcomponents of the system and, if the subcomponent is of type *device*, sums the power draw. The total draw of all devices is then compared against the maximum draw available. If the total draw is less than the maximum draw, the verification succeeds, otherwise, it fails.

The two components defined are a device and a system containing the device as a subcomponent. The system defines the maximum draw it can sustain with the *Power_Available* property, and the device defines how much it draws with the *Voltage_Drawn* property.

```
annex resolute {**
   PowerDrawLessThanMax(self: component, max: real) <= **"""**
      let subs: component = subcomponents(self);
      SumPowerDraw(subs) <= max

   SumPowerDraw(components: component) : real =
      sum(property(x, Voltage_Drawn, 0.0) for (x : components) | x instanceof device)
**};

device front_distance_sensor
   features
      distance: out data port Base_Types::Integer;
   properties
      Voltage_Drawn => 0.5;
end front_distance_sensor;

system implementation vehicle_obe.impl
   subcomponents
      front_distance_sensor: device Devices::front_distance_sensor;
   properties
       Voltage_Available => 3.6;
    annex resolute {**
       prove(PowerDrawLessThanMax(this, property(this, Voltage_Available, 0.0)))
    **};
end vehicle_obe.impl;
```

FIGURE 8.14

Resolute power draw validation.

Finally, we will provide an example using the EMV2 error annex. In doing so, we will consider a simplified version of the data bus for the DNPW architecture, given in Fig. 8.15. If we consider the error ontology defined in the error annex, shown in Section 8.6, we can begin determining the types of errors that could occur. Since this is a primitive signed value, the port would be susceptible to all of the Detectable Value Error categories. Also, considering that the device functions as a bus, it would be susceptible to the Item Timing Error categories as well. We provide the error annex implementation that represents these errors in Fig. 8.16. Note that this figure uses the behavior previously defined in Fig. 8.8.

So far we have shown how the system engineer proceeds from CVRIA to requirements to a detailed design with varying degrees of specificity. We will now bring this back to continuous engineering with data analytics. Now that we have an architecture and detailed design, the implementation and testing phases can commence. Ultimately, requirements and design will likely change as a result of data that is produced by each of these phases. For example, during testing, it has been discovered that a distance sensor is also needed on the rear of the vehicle to determine if the vehicle needs to speed up in order to avoid a collision with a vehicle following too closely. This information would be fed back into the requirements which would facilitate a change in architecture. The architectural change would necessitate changes to the implementation and the tests.

Data is collected not only from the development process, but is also collected from other sources including deployed instances of the system. Consider the following scenario. Initially we had assumed that the geometry of the passing area could be determined from roadside infrastructure. The vehicle developed to use the DNPW system described in this scenario had a limited market

```
device can_bus
   features
      speed_in: in data port Base_Types::Integer;
      speed_out: out data port Base_Types::Integer;
end can_bus;
```

FIGURE 8.15

Resolute power draw validation.

```
annex EMV2 {**
   use types ErrorLibrary;
   use behavior EvaluatorBehavior;
   error propagations
      speed_in: in propagation {DetectableValueError, ItemTimingError};
      speed_out: out propagation {DetectableValueError, ItemTimingError};
   end propagations;
   component error behavior
      transitions
         normal -[speed_in{DetectableValueError, ItemTimingError}]-> transient_failure;
   end component;
*};
```

FIGURE 8.16

EMV2 annex.

deployment initially, only major cities in the Eastern United States. As production grows and vehicles are distributed in other regions, data from the DNPW eventually yields that the assumption does not hold for vast stretches of roadway in the Western United States where the distance between major cities is much greater than in the Eastern United States. These roadways have yet to be instrumented due to the large cost of doing so versus the fact that they are rarely traveled. This necessitates both a change in the requirements and design as the system now needs additional sensors to enable it to form an internal representation of the passing area.

One final note must be mentioned; the collection of data is not an automatic activity if continuous engineering is adopted. Rather continuous engineering is a practice that necessitates a culture of data collection and a culture of good analytics. In some cases, the amount of data collected can be overwhelming necessitating big data processing techniques. In some cases, the data may be sparse and extrapolations must be conducted in order for the data to be useful. Developing a culture that is capable of handling these tasks is essential to the success of continuous engineering. If such a culture is developed or already present, continuous engineering can enable better decision making at a faster and more reactive pace.

8.4 SUMMARY AND CONCLUSION

System engineers make fundamental decisions about the systems needed to support ITSs and CVs. They allocate responsibilities, in the form of requirements, to both hardware and software on all platforms that participate in the system. Through the use of ADLs like AADL, extensive models of the system are created, and analysis is performed with each refinement of the model allowing more errors to be caught early in system development. These description languages also allow fundamental changes to the system discovered as a result of CE to be inserted into the model rapidly allowing engineers to quickly determine all of the areas that will be affected by the necessitated change. However, CE and extensive models require good use of data collection and excellent data analysis skills. The flows of data in ITSs and CVs are extensive and as the technologies behind these systems mature, the number of data flows is likely to increase. The architectures and applications involved in data management/analysis for ITSs and CVs should evolve as the systems evolve. Current research is making progress towards this end by uncovering patterns common in CVs and the IoT infrastructure to improve reliability, security, and stability for all systems which rely on the IoT.

8.5 EXERCISES

1. In Systems Engineering, it is often necessary to look at the expected latency involved with calling a remote subroutine. Using Fig. 8.17, calculate the total cost incurred when calling the remote subroutine represented by the square in the diagram.
   - **a.** 15 Ms
   - **b.** 144 Ms
   - **c.** 167 Ms
   - **d.** 38 Ms

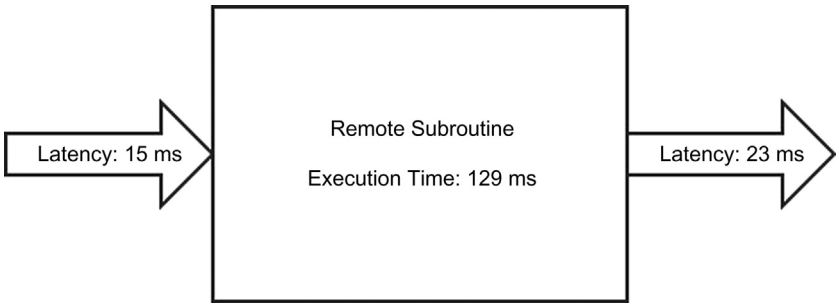

FIGURE 8.17

Exercise 1 diagram.

2. In the system discussed in question 1 there exists a requirement that states "Remote subroutine calls should not take longer than 200 Ms." There is also another requirement that states "The average response time of a remote subroutine should be 150 Ms total to allow for some long-processing requests." If we say that Fig. 8.17 represents the average response time of that subroutine, will the subroutine represented in Fig. 8.17 need to be redesigned?
   a. Yes
   b. No
3. In Systems Engineering, it is also common to look at the update and receive rates of different systems to ensure that data values won't be missed or to calculate how many will be missed. Take the systems in Fig. 8.18, System A produces a new value every 20 Ms while System B reads a new value from A every 25 Ms. How many updates will System A produce in 1 minute?
   a. 3000 updates
   b. 500 updates
   c. 1500 updates
   d. 2500 updates

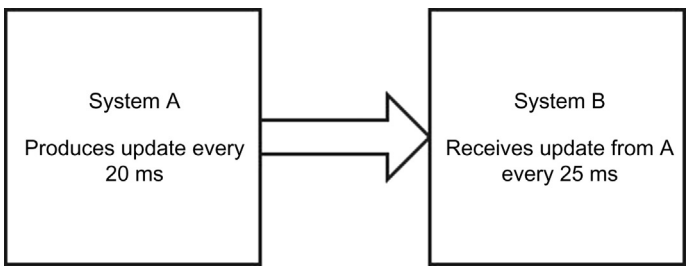

FIGURE 8.18

Exercise 3 diagram.

4. Once again considering Fig. 8.18, how many updates will System B read in one minute?
   a. 400 updates
   b. 2400 updates
   c. 1000 updates
   d. 1200 updates
5. Once again considering Fig. 8.18, how many data values will be lost per minute?
   a. 1000 updates
   b. 800 updates
   c. 600 updates
   d. 400 updates
6. The majority of system engineering takes place during what phase(s) of the Systems Development Life Cycle?
   a. Requirements
   b. Design
   c. Construction
   d. Validation

Answers:

1. (c) The latencies from both the request and response as well as the execution time are added together to produce the total cost incurred from the remote subroutine call.
2. (a) The remote subroutine calls meet the first requirement, but it does not meet the second requirement. It will be necessary to either revise the requirement or to modify the subroutine in order to bring it into compliance.
3. (a) There are 60 * 1000 = 60,000 Ms in one minute. A new value produced every 20 Ms would produce 60,0000/20 = 3000 values.
4. (a) There are 60 * 1000 = 60,000 Ms in one minute. A new value read every 25 Ms would read 60,0000/25 = 2400 values.
5. (c) 3000 values produced−2400 values read = 600 values lost.
6. (a & b) Most of the engineering work in the Systems Development Life Cycle occurs up front in the first two phases. This does not mean that no engineering work is done in later phases, quite the opposite.

8.6 APPENDIX A

8.6.1 EMV2 ERROR ONTOLOGY

- ServiceError
  - ItemOmission
  - ServiceOmission
  - SequenceOmission
    - TransientServiceOmission
    - LateServiceStart
    - EarlyServiceTermination
    - BoundedOmissionInterval

- ItemCommission
- ServiceCommission
- SequenceCommission
  - EarlyServiceStart
  - LateServiceTermination

- TimingRelatedError
  - ItemTimingError
    - EarlyDelivery
    - LateDelivery
  - SequenceTimingError
    - HighRate
    - LowRate
    - RateJitter
  - ServiceTimingError
    - DelayedService
    - EarlyService
- ValueRelatedError
  - ItemValueError
    - UndetectableValueError
    - DetectableValueError
      OutOfRange
      BelowRange
      AboveRange
  - SequenceValueError
  - BoundedValueChange
    - StuckValue
    - OutOfOrder
  - ServiceValueError
    - OutOfCalibration
- ReplicationError
  - AsymmetricReplicatesError
    - AsymmetricValue
      AsymmetricApproximateValue
      AsymmetricExactValue
    - AsymmetricTiming
    - AsymmetricOmission
      AsymmetricItemOmission
      AsymmetricServiceOmission
  - SymmetricReplicatesError
    - SymmetricValue
      SymmetricApproximateValue
      SymmetricExactValue
    - SymmetricTiming
    - SymmetricOmission
      SymmetricItemOmission
      SymmetricServiceOmission

- ConcurrencyError
  - RaceCondition
  - ReadWriteRace
  - WriteWriteRace
  - MutExError
  - Deadlock − Starvation

REFERENCES

[1] https://en.wikipedia.org/wiki/Intelligent_transportation_system.
[2] http://automotive.cioreview.com/cxoinsight/data-analytics-for-connected-cars-nid-6142-cid-4.html.
[3] https://www.pcb.its.dot.gov/eprimer/module2.aspx.
[4] https://cdn.sparkfun.com/datasheets/Wireless/Zigbee/ds_xbeezbmodules.pdf.
[5] S. Arafat, Continuous innovation through continuous engineering, 2015.
[6] P. Feiler, D. Gluch, and J. Hudak, The architecture analysis & design language (AADL): An introduction, 2008.
[7] Z. Yang, K. Hu, D. Ma, L. Pi, Towards a formal semantics for the AADL behavior annex, In: *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09*, April 2009, pp. 1166−1171.
[8] J. Delange and P. Feiler, Architecture fault modeling with the AADL error-model annex, in *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, Aug 2014, pp. 361−368.
[9] A. Murugesan, M.W. Whalen, S. Rayadurgam, M.P.E. Heimdahl, In: Compositional verification of a medical device system, ACM SIGAda Ada Letters, vol. 33, ACM, 2013, pp. 51−64.
[10] A. Gacek, J. Backes, D. Cofer, K. Slind, M. Whalen, Resolute: An assurance case language for architecture models, ACM SIGAda Ada Letters, vol. 34, ACM, 2014, pp. 19−28.
[11] Department of Transportation. Connected vehicle reference implementation architecture. <http://www.iteris.com/cvria/>.
[12] http://www.iteris.com/cvria/html/applications/app16.html.
[13] https://www.iteris.com/cvria/html/applications/app16.html#tab-3.
[14] http://www.iteris.com/cvria/html/applications/app16.html#tab-5.