IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 2 July 2022, accepted 11 July 2022, date of publication 19 July 2022, date of current version 26 July 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3192436

RESEARCH ARTICLE

Synthesis of a Controller Algorithm for Safety-Critical Systems

FELLIPE GUILHERME REY DE SOUZA[1], CELSO MASSAKI HIRATA[1], AND SIMIN NADJM-TEHRANI[2]

[1]Department of Computer Science, Instituto Tecnológico de Aeronáutica, São José dos Campos 12228-900, Brazil
[2]Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden

Corresponding author: Fellipe Guilherme Rey de Souza (fellipeguilhermerey@gmail.com)

The work of Celso Massaki Hirata was supported in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico under Grant 309620/2021-0, and in part by the Fundação de Amparo à Pesquisa do Estado de São Paulo under Grant 2022/01051-7. The work of Simin Nadjm-Tehrani was supported by the Swedish Governmental Agency for Innovation Systems-Vinnova, as part of the National Projects on Aeronautics (NFFP7), Project CLASSICS, under Grant NFFP7 2017-04890.

ABSTRACT Systems of today are becoming more complex; they have many levels of the control hierarchy, are software-intensive, use different networks, have increasing processing power, use a diversity of devices, and require more integration. Systems-Theoretic Process Analysis (STPA) is a technique that is being used to analyze the safety of those systems at the concept stage. For the design phase, STPA can be combined with SysML modeling activities, including simulation and formal verification of systems models to produce the control software more efficiently. However, for the design phase, when starting from the STPA analysis there is no support to elaborate the control algorithm. Building the control algorithm is one of the most difficult tasks in the design phase. We propose a method to synthesize the control algorithm for safety-critical systems from the STPA analyses and the functional requirements. Our method maps the control structure (STPA) into a block diagram (SysML), and it uses the STPA results to generate an initial state machine diagram (SysML) for automated controllers, actuators, and sensors. We use our method to generate the control algorithms for an Adaptive Cruise Control system. We evaluate the synthesized algorithms by performing model simulation and formal verification. This illustrates that our method is a systematic way to synthesize control algorithms that satisfy both safety and functional requirements.

INDEX TERMS Safety, systems modeling language, model checking, control system synthesis, system analysis, and design.

I. INTRODUCTION

Systems that we are building today are becoming more complex. They have many levels of control hierarchy, including human operators, decision makers, automated controllers, and autonomous controllers. They use different networks, have increasing processing power, use a diversity of devices, and require more integration. They are also software-intensive, which makes the development more expensive and longer. Software plays an essential role in systems control performing safety-critical tasks, where an error can lead to a loss.

The associate editor coordinating the review of this manuscript and approving it for publication was Engang Tian.

Systems-Theoretic Accident Model and Processes (STAMP) [1] is an accident causality model based on system theory and system thinking. Unlike the traditional safety analysis techniques based on the reliability theory, System-Theoretic Process Analysis (STPA) is a top-down hazard analysis technique based on STAMP. STPA identifies all causal scenarios found by the traditional techniques and more causal scenarios (often software-related) [2]. Moreover, STPA identifies more hazard scenarios involving software and component interaction than Failure Mode and Effect Analysis (FMEA) [3].

Souza *et al.* [4] propose a method that combines STPA with SysML modeling activities. The method allows for simulation and formal verification of systems models. Their method

employs SysML use case and sequence diagrams to perform analysis and SysML block diagrams to specify the system behavior. Inside each block, there is a state machine diagram that establishes the behavior (algorithm) of the block. However, the method does not provide any systematic support to elaborate the state machine diagram or design the software logic of the controller. Building the software is one of the most difficult tasks in the design phase since it must consider all the inputs to produce a controller that meets all the requirements, including the functional and safety requirements.

Recently, there have been significant advances in assessing the information identified in the STPA analysis through the combination of STPA with NuSMV [5], UPPAAL [6], or Event-B [7]. Although these approaches provide ways to support the safety assessment of the systems, they rely on the expertise of the designer in UPPAAL, Event-B, or NuSMV. The main challenge is how to assess the results identified in the STPA analysis in a way that the designer does not need to have major experience in modeling diagrams and how to use model checkers.

We propose a method that combines the STPA analysis and the functional requirements to systematically synthesize the control algorithm for non-human controllers. The method synthesizes a block diagram based on the STPA's control structure and an initial state machine diagram for automated controllers, actuators, and sensors. The advantages of using the proposed method are threefold. First, it allows checking the behavior of controllers through simulation and formal verification. Second, the method permits checking the correctness and completeness of the design and the STPA analysis (considering that STPA is the basis of the generation of the control algorithm). Last, it allows detecting existing conflicts in the safety requirements and between the STPA requirements and the functional requirements.

We generate the block and state machine diagram using Systems Modeling Language (SysML) [8]. Besides SysML, other feasible options are Unified Model Language (UML) [9], Arcadia/Capella [10], Modeling and Analysis of Real-Time and Embedded System (MARTE) profile for UML [11], Architecture Analysis & Design Language (AADL) [12], and Simulink [13]. We choose SysML because it is a widely used language for the development in the Systems Engineering field. Moreover, the SysML profile Automated Verification of Real-Time Software (AVATAR) [14] allows designers to elaborate, simulate and formally verify their diagrams having minor modeling skills with SysML or with temporal languages such as Computation Tree Logic (CTL) or UPPAAL [15].

We organize the remainder of this paper as follows. Section II provides the background required to understand the proposed method. Section III presents the related works and the novelty of our method. Section IV introduces the Adaptive Cruise Control system (ACC), used as a running example to illustrate the method. Section V introduces the method and its activities. Section VI introduces a tool that supports our method. Section VII provides the results of performing simulation and formal verification in the ACC. Section VIII presents the concluding remarks, discussion, recommendations, and suggestions for future work.

II. BACKGROUND WORK

This section briefly introduces the four steps of STPA and the activities of Souza *et al.* [4] method to combine STPA analysis and SysML modeling. These works are fundamental for the understanding of our method.

A. SYSTEM-THEORETIC PROCESS ANALYSIS

Systems-Theoretic Process Analysis (STPA) is a hazard analysis technique based on Systems-Theoretic Accident Model and Processes (STAMP) [1]. STAMP has its theoretical foundation on systems theory, and it considers safety as an emergent property that arises from component interaction.

STPA has four steps [2]. The first step is ''Define the purpose of the analysis''. The safety experts define the system engineering foundation of the analysis, identifying the losses, hazards, system-level safety constraints, and the relations between them. The second step is ''Model the control structure''. The hierarchical control structure is a concept from systems theory, where components of each level impose constraints on components of the level beneath. The safety experts must define the controllers, actuators, sensors, and controlled processes that compose the system. Moreover, they establish the relationships between components, such as control actions and feedback links.

The third step is ''Identify unsafe control actions''. Each controller identified in the previous step has a set of control actions (commands that must be issued to provide the system's functionality, keeping the system safe). In this step, the safety experts must identify why providing a control action anytime, not providing it, providing it at the wrong time or order, and applying a continuous control action too long or stopping it too soon causes a hazard. The last step is ''Identify loss scenarios''. It requires brainstorming of the safety experts to identify the loss scenarios in which the controller issues an unsafe control action. Moreover, they must identify recommendations to eliminate or mitigate these loss scenarios.

B. COMBINING STPA ANALYSIS AND SysML MODELING

Souza *et al.* [4] propose a method that combines STPA with SysML modeling activities, including simulation and formal verification of system models. Figure 1 depicts the nine activities of the Souza *et al.* method. There are two sets of activities in the figure. The first set is SysML (represented by the upper and lower part in the figure), which shows the activities related to the SysML modeling, model simulation, and model verification. The second set (represented by the part in the middle) represents the activities of the STPA analysis.

The first activity is stating the *Model Assumptions*. The output of this activity is list of assumptions that the system engineers and safety experts make to proceed with the analysis. The second activity is *Capture Requirements*,

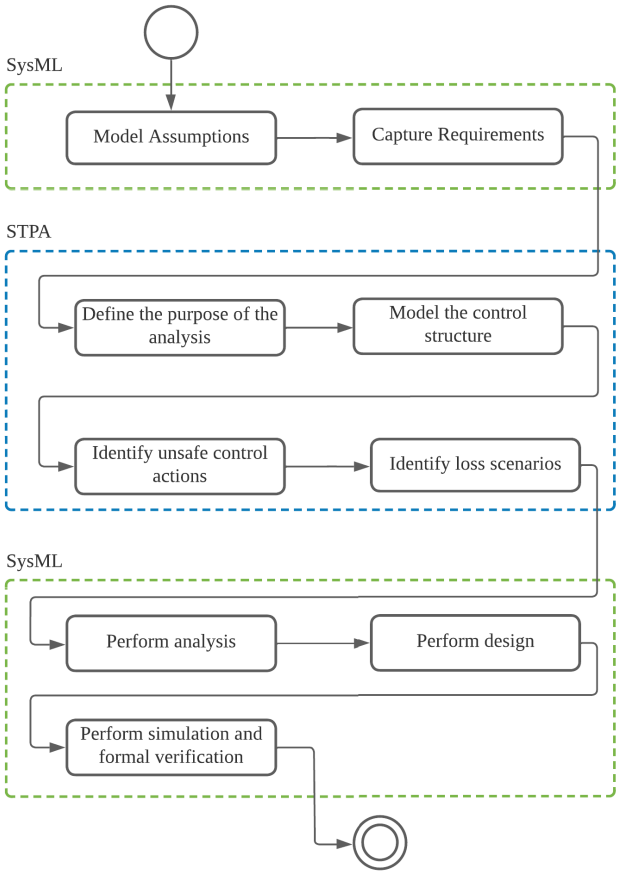

FIGURE 1. Method for combining STPA with SysML modeling [4].

corresponding to requirements elicitation from users, customers, and stakeholders.

The next four activities are the STPA steps (Section II-A). The activity *Perform Analysis* provides ways to use sequence and use case diagrams to confirm if the system's components and interactions were adequately identified in the activity *Model the control structure*. In the *Perform Design* activity, the system engineers must design the block diagram based on the control structure produced within STPA. For each block of the block diagram, there is a state machine diagram corresponding to the behavior of each component.

The last activity is *Perform simulation and formal verification*. This activity enables the system engineers to perform a simulation and use a model checker (such as UPPAAL) to check the safety properties. The simulation aids the system engineers in checking which paths of the model are being explored, including those related to safety and functional requirements.

III. RELATED WORK

Zhong *et al.* [5] propose an adaption of STPA based on SysML/MARTE and NuSMV (STPA-SN), which aims to improve the formalization of STPA. Their proposal includes (i) the use of the SysML internal block diagram in STPA step ''Model the control structure''; (ii) definition of the unsafe control action as a state and the unsafe control action as a command received by the controlled process (instead of issued by the controller) in STPA step ''Identify unsafe control actions''; and (iii) automatic definition of loss scenarios through the conversion between SysML state machine diagrams and NuSMV models. They use model checking to verify the safety constraints (converted into CTL/LTL properties) and the NuSMV model (constructed considering the MARTE's state machine diagrams after the addition of the causal factors from the ''Identify Loss Scenario'' step). Contrary to our method, their method does not provide any support to generate the internal block diagram and the system state machine diagrams (MARTE) automatically.

Zhao *et al.* [6] propose a method to perform a safety assessment of Reconfigurable Integrated Modular Avionics (IMA) using STPA and UPPAAL. Starting from the control structure of the STPA analysis, they provide the STPA-UPPAAL modeling rules to elaborate the timed automata manually for the IMA controller. They used the UPPAAL model checker to validate the controller, creating a statement for each unsafe control action and loss scenario identified using STPA. Different from our method, their work is tailored to the IMA systems. Although they introduce the STPA-UPPAAL modeling rules, they not describe how to use STPA to elaborate the timed automata automatically.

Howard *et al.* [16] present a methodology (named SE-STPA) to combine safety and security using unified concepts and processes based on STPA and Event-B. They construct the initial formal system model using Event-B and the tool Rodin, translating the artifacts of STPA into Event-B artifacts (such as control actions of STPA into events of Event-B). They also add Manipulation Points (MPs) in the control structure to determine how adversaries can access the system under analysis. They aim to generate critical requirements from the third step of STPA and the MPs to extend and refine their model. Their methodology aims to combine safety and security but they generate the contexts from Event-B. Meanwhile, we intend to generate automatically the behavior of the controller using safety and functional requirements.

Dghaym *et al.* [7] propose a framework for eliciting mission requirements for squads of autonomous missions. They use and refined the methodology proposed by Knorreck *et al.* [15] to formalize the critical requirements identified with Event-B models. They propose an approach with nine steps for building a correct-by-construction system. Their approach allows increasing the number of requirements identified through an iterative and continuous analysis. According to the authors, the main limitation of their approach is the expertise required in Event-B and formal methods. In their method, the system engineer must define the context of Event-B manually as opposed to generating the Event-B context through the SE-STPA.

Dakwat and Villani [17] propose a method that combines STPA and model checking. They used a robot flight simulator to demonstrate the effectiveness of their work. They provide a formal and unambiguous representation of the system

through a method that has six steps borrowed from STPA and UPPAAL. They use the two first steps of STPA to create an UPPAAL model and then use the list of unsafe control actions and safety constraints to review the STPA analysis and refine the UPPAAL model. Contrary to our method, they do not provide an algorithm to create the UPPAAL timed automata (corresponding to the state machine diagram, in our case). Moreover, they consider that STPA is sufficient to elaborate the model, disregarding the functional requirements that do not lead to a loss.

Wang *et al.* [18] propose an integrated model-checking scheme based on SysML and Model-Based Safety Analysis, conducting a model transformation between the SysML diagram and the NuSMV symbolic model checker. The block definition diagram (*bdd*) describes the structural composition of the system, and the state machine diagram (*stm*) represents the system behavior. They translate the *bdd* to the main module of NuSMV and its sub-modules, and define transformation rules to transform the *stm* into NuSMV based on the state machine's elements (e.g., status, event, and transition). The authors claim that their method can ensure model consistency, improve usability in engineering, and find defects in product development. Different from our work, they neither propose a way to produce the SysML diagrams automatically nor use STPA.

Krishnan and Bhada [19] propose an Integrated System Design and Safety (ISDS) framework that is capable of integrating Model-Based Systems Engineering (MBSE) with traditional safety analysis techniques, such as FMEA and Fault Tree Analysis (FTA). The framework follows the "V" system development lifecycle model and contains three phases. First, they use their own SysML profile to initiate the system design and safety analyses at the system level to produce the system-level FMEA and fault trees. Subsequently, they construct the system design and safety analyses at the subsystem or component level. Last, they verify the safety of the system design at the system and subsystem level using unit testing and methods such as fault injections. The authors only complete Phase 1, and the results of phases two and three are not elaborated. They claim that it is possible to use a model checker in their frameworks without further elaboration. It also leaves the choice of FMEA and FTA unclear.

Ding *et al.* [20] present a method that combines Finite State Machine (FSM) and model checking. They aim to avoid system abnormalities in the safety analysis method caused by errors in the control signals. The method creates an FSM model by combining the state transition diagram and the state transition table. They extract the state of the system and examine the input and output of each state to analyze the transition between states to build the state transition diagram. Finally, they convert the FSM model to the Symbolic Model Verifier (SMV) program through a mapping between FSM and the NuSMV program. Once the conversion is complete, it is possible to use NuSMV to check the SMV program against a Linear Temporal Logic (LTL) specification. They demonstrate their method using an aircraft Engine control software. They do not relate a hazard analysis technique to their work, arguing that their primary concern is to explain how multiple causes can lead to an accident.

Pétin *et al.* [21] present a combination of the SysML semi-formal modeling approach to identify and refine safety requirements, design the control system using formal models and verify its dynamic properties. Their system modeling approach combines non-formal methods based on SysML requirements, block diagrams, and model checking using UPPAAL to prove that the local behavior of each system component contributes to satisfying the system requirements. Although they produce the block diagram and the behavioral diagram (through UPPAAL timed automata), they focus only on safety properties, not the functional requirements. How to translate the state machines from SysML to timed automata from UPPAAL is not elaborated. However, it is clear that their aim is not to elaborate the behavioral diagram automatically.

Abdulkhaleq *et al.* [22] propose an approach that generates test cases from the safety requirements of STPA to design the model of the system. They formalize the safety requirements of STPA in Linear Temporal Logic (LTL) and construct the safe Software Behavioral Model (SBM) using the Stateflow diagram notation. Then, they transform the SBM into Symbolic Model Verifier (SMV) to check the model against the LTL statements and generate an Extended Finite State Machine (EFSM) from the SBM. Finally, they produce the test cases from the ESFM using tree search-based algorithms (such as depth-first search, breadth-first search, and both combined) that can be tested both on the ESFM or in an generated source code. While their work generates test cases for a source code, our work aims to model the controller at the design phase. At the same stage of system development, they did not provide any aid to create the SBM – according to the authors, the elaboration of the SBM depends on the designer's skills.

IV. STPA APPLIED TO AN ADAPTIVE CRUISE CONTROL

We present the Adaptive Cruise Control system (ACC) as a running example to describe our synthesizer in Section V. ACC is an extension of the standard Cruise Control. ACC controls the speed of the vehicle, but it cannot be considered an autopilot. The driver of the vehicle referred to as *Driver* must intervene in situations such as maintaining the vehicle in the lane and diverting from obstacles.

ACC has one or more sensors (such as radar and computer-connected cameras) installed in front of the vehicle. The sensor is responsible for measuring two data: the speed of the forward vehicle (FV) and the distance between the ego vehicle and FV. If ACC is engaged and there is no FV, ACC behaves like the standard cruise control, maintaining the speed set by *Driver*. When ACC is on, and there is a FV, ACC shifts from the speed control mode (standard cruise control) to spacing control mode (or following mode). In this last mode, ACC controls the vehicle's speed to maintain the preset distance from FV. *Driver* is responsible for setting the

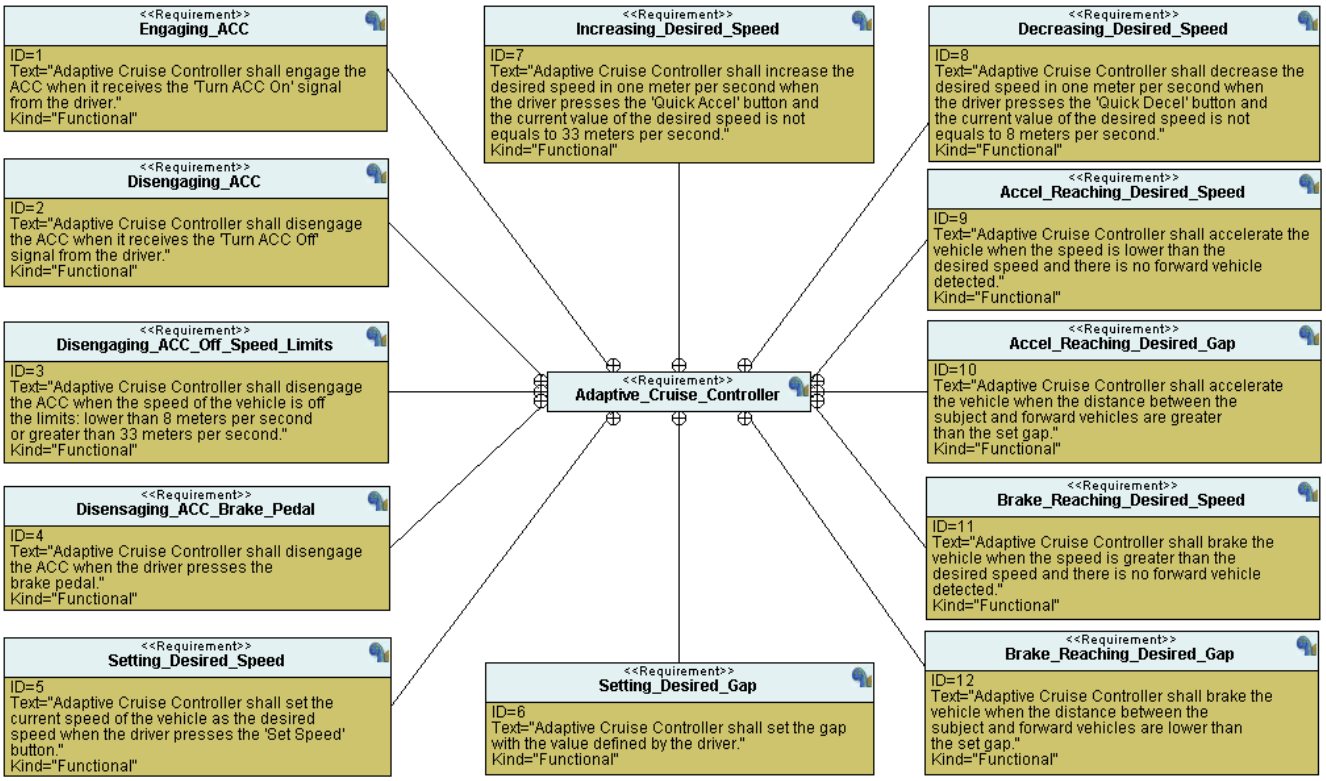

FIGURE 2. Functional requirements for ACC system.

safe distance (gap) between both vehicles. In this section, we provide the assumptions, functional requirements, use cases specifications, and the STPA analysis for ACC.

A. MODEL ASSUMPTIONS

When analyzing a system, the system engineers must document the assumptions about the system and environment that the development team must consider to conduct their analysis. We strive to perform the analysis presented herein, aligning it with the system specifications described in the ISO 15622:2018 [23]. To this end, we made six assumptions to serve as a guide to conduct the STPA analysis for ACC. The assumptions are:

- **A-1**: According to Moon *et al.* [24], ACC has two controllers: An upper-level controller, responsible for calculating the desired acceleration using the longitudinal velocity, relative distance, and relative velocity; and a lower-level controller, responsible for physically manipulating the throttle and brake actuator to reach the desired acceleration. Our focus is the upper-level controller. Therefore, we will not consider the mathematical equations of throttle angle and torques.
- **A-2**: The focus of the analysis is the ACC. Despite other classes of Advanced Driver-Assistance Systems (such as anti-lock braking system, lane-keeping system, traffic sign recognition, and others) improve the driver's experience and work jointly with ACC, they will be not considered in this analysis.
- **A-3**: ACC must not be used when the weather is poor. Therefore, we will not consider rain, fog, or snow. Moreover, we will not consider the lack of the tire grip due to ice-covered roads or hydroplaning.
- **A-4**: The focus of the analysis is the Limited Speed Range ACC (LSRA) [23] equipped in a vehicle that is traveling on a highway (a road where pedestrians and non-motorized vehicles are prohibited to use). Since LSRA does not operate at low velocities, the analysis does not consider traffic and the stop-and-go function.
- **A-5**: The vehicle has ACC type LSRA 2, where no manual clutch operation is required (vehicle equipped with automatic transmission).
- **A-6**: We will not consider rear-end collisions. Even if ACC brakes abruptly, the driver of the vehicle behind has the responsibility of maintaining the safe distance between the vehicle ahead (in this case, the vehicle with ACC).

B. REQUIREMENTS

After the assumptions, we elicit the requirements. Based on our knowledge of the Adaptive cruise controller system, we capture twelve requirements. Figure 2 illustrates them. The links with the box in the middle indicate that the surrounding boxes (requirements) are related to the Adaptive cruise controller. Each requirement has an identifier (ID), the text of the requirement, and its kind (Functional or Non-Functional).

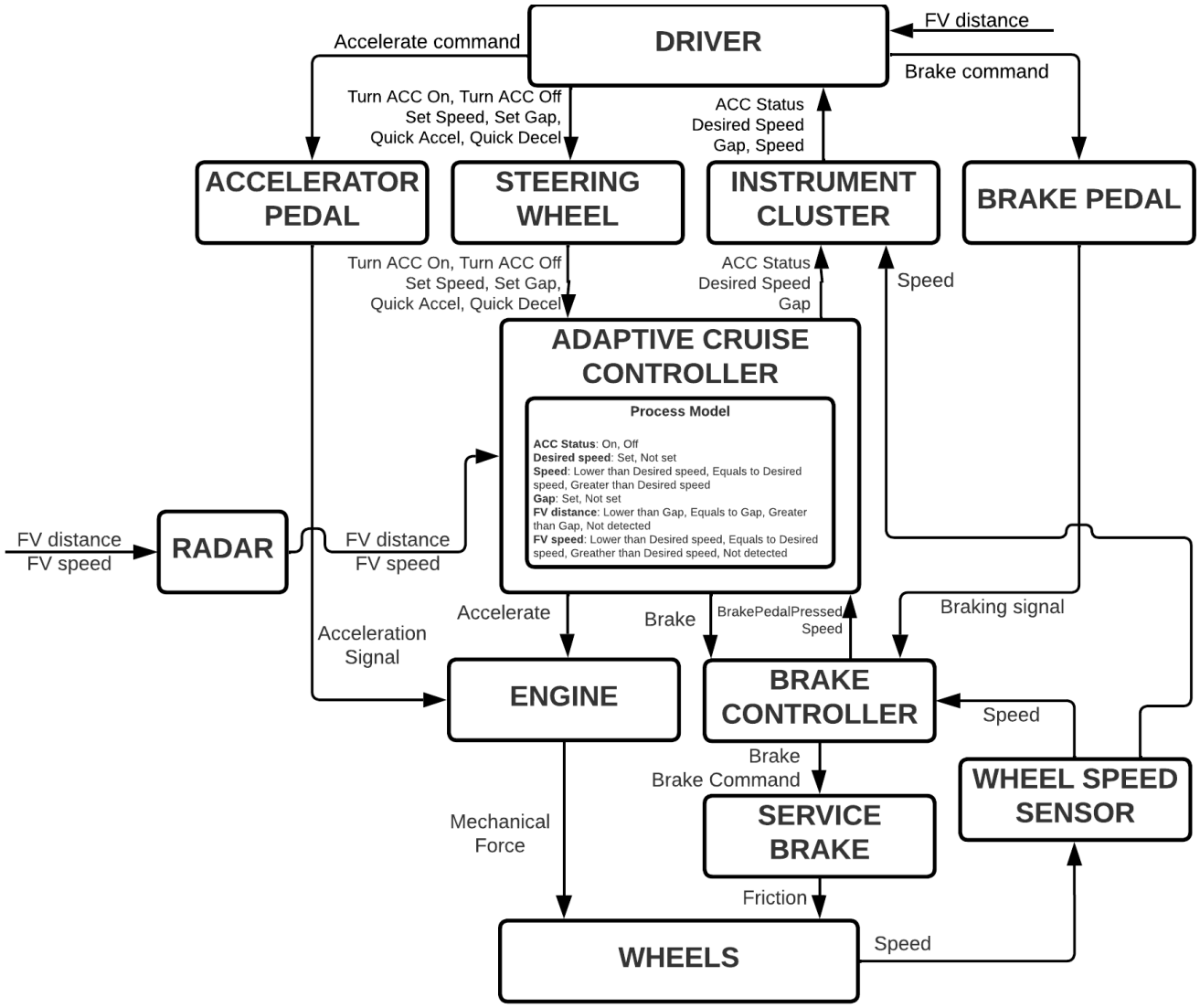

FIGURE 3. Control structure for ACC system. Adapted from Leveson and Thomas [2].

C. THE SYSTEM-THEORETIC PROCESS ANALYSIS (STPA) FOR ACC

1) DEFINE THE PURPOSE OF THE ANALYSIS

Here, we start conducting the STPA analysis. The first step is to define the purpose of the analysis. Table 1 shows the losses, hazards, and system-level safety constraints identified for the ACC system. Each hazard is associated with one or more losses, and each system-level safety constraint is associated with one or more hazards. The association is the identifier between brackets at the end of the hazard or system-level safety constraint.

TABLE 1. Losses, hazards, and system-safety constraints for ACC system.

Losses	Hazards	System-level safety constraint
L-1: Vehicle occupants are injured.	**H-1**: Safe distance to the forward vehicle is not respected [L-1] [L-2] [L-3]	SSC-1: The safe distance to the forward vehicle must be respected. [H-1]
L-2: Vehicle damage.	**H-2**: Disengaged ACC controls the vehicle [L-1] [L-2] [L-3].	**SSC-2**: The ACC must control the vehicle only when it is engaged [H-2].
L-3: The occupants of the forward vehicle are injured.	**H-3**: The ACC does not follow the parameters defined by the Driver [L-1] [L-2] [L-3].	**SSC-3**: The ACC must follow the parameters defined by the Driver [H-3]

2) MODEL THE CONTROL STRUCTURE

The second step is to model the control structure. The control structure is a representation of the model of the system composed of feedback control loops. The boxes represent the components (such as controllers, actuators, sensors, and controlled processes). The downward arrows are the control actions or control action enforcements. The upward arrows are the feedback. The horizontal arrows without a source component are the external information (that comes from the environment or external systems).

Figure 3 depicts the control structure for ACC. We customize the ACC's control structure of Leveson and

Thomas [2] to be compliant with our system under analysis. The system has a human controller, named *Driver*. *Driver* is responsible for accelerating and braking the vehicle when necessary.

Moreover, *Driver* controls ACC through four buttons in *Steering wheel*, which are the following control actions: *Set Speed*, *Set Gap*, *QuickAccel*, and *QuickDecel*. *Set Speed* sets the current speed of the vehicle as to the *Desired speed*. *Set Gap* sets the current distance between the vehicle with ACC and Forward Vehicle (FV) as the *Gap* (desired distance). *QuickAccel* increases the *Desired speed* by one unit, and *QuickDecel* decreases the *Desired speed* by one unit.

The process model for the process that the *Driver* is controlling has five variables, which are: *ACC Status*, with values *On* and *Off*; *Desired speed*, with values *Set* and *Not set*; *Speed*, with values *Lower than Desired speed*, *Equals to Desired speed*, and *Greater than Desired speed*; *Gap*, with values *Set* and *Not Set*; and *FV distance*, with values *Lower than Gap*, *Equals to Gap*, *Greater than Gap*, and *Not detected*.

The actuator *Steering wheel* receives the control actions from *Driver* and forwards them to *Adaptive cruise controller*. The sensor *Instrument cluster* receives *ACC Status*, *Desired speed*, and *Gap* from *Adaptive cruise controller* and *Speed* from *Wheel speed sensor* and forwards them to *Driver*.

The *Adaptive cruise controller* is responsible for controlling the vehicle when *Driver* engages ACC. *Adaptive cruise controller* provides two control actions to accelerate and decelerate (or *Brake*) the vehicle. Its process model has six variables, where five of them are equals to the variables of *Driver*: *ACC Status*, *Desired speed*, *Speed*, *Gap*, and *FV distance*. The additional variable is *FV speed*, with values *Lower than Desired speed*, *Equals to Desired speed*, *Greater than Desired speed*, and *Not detected*. The radar senses the speed (*FV speed*) and distance (*FV distance*) of the forward vehicle. The ACC receives *Speed*, *FV distance*, and *FV speed* as input. The *ACC Status*, *Gap*, and *Desired speed* are process model variables of ACC. Moreover, ACC receives the *Brake Pedal Pressed* signal from the *Brake Controller*.

When ACC commands the acceleration or *Driver* presses *Accelerator pedal*, *Engine* is responsible for accelerating the vehicle. In this analysis, we consider *Engine* as an actuator. However, the engine is a complex component that encompasses other components (such as the Engine control module and the Electronic throttle body).

Brake controller receives the signal to brake the vehicle from *Brake pedal* and *Adaptive cruise controller*. *Brake controller* activates *Service brake* to slow down the vehicle. *Brake controller* has the variable *Speed*, with the same value as defined in *Driver*. *Wheel speed sensor* measures the speed of *Wheels* (the vehicle speed) and sends it to *Brake controller* and *Instrument cluster*. Since the focus is the *Adaptive cruise controller*, we suppress the Process Model from *Driver* and *Brake controller* in Figure 3.

TABLE 2. Unsafe control actions for the *Adaptive cruise controller*.

CONTROL ACTION: ACCELERATE	
Providing causes hazard	Adaptive cruise controller provides Accelerate when ACC Status is Off [H-2]. Adaptive cruise controller provides Accelerate when ACC Status is On, and Desired speed is Not set [H-3]. Adaptive cruise controller provides Accelerate when ACC Status is On, and Desired speed is Set, and Speed is Equals to Desired speed [H-3]. Adaptive cruise controller provides Accelerate when ACC Status is On, and Desired speed is Set, and Speed is Greater than Desired speed [H-3]. Adaptive cruise controller provides Accelerate when ACC Status is On, Desired speed is Set, Gap is Set, and FV distance is Equals to Gap [H-3]. Adaptive cruise controller provides Accelerate when ACC Status is On, Desired speed is Set, Gap is Set, and FV distance is Lower than Gap [H-1].
Not providing causes hazard	No unsafe control actions
Providing in wrong time or order causes hazard No unsafe control actions	No unsafe control actions
Stopping too soon or applying too long causes hazard	Adaptive cruise controller applies Accelerate too long when ACC Status is On, Desired speed is Set, and Speed is Equals to Desired speed [H-3]. Adaptive cruise controller applies Accelerate too long when ACC Status is On, Desired speed is Set, Gap is Set, and FV distance is Equals to Gap [H-1].
CONTROL ACTION: BRAKE	
Providing causes hazard	Adaptive cruise controller provides Brake when ACC Status is Off [H-2].
Not providing causes hazard	Adaptive cruise controller not provide Brake when ACC Status is On, Desired speed is Set, and Speed is Greater than Desired speed [H-3]. Adaptive cruise controller not provide Brake when ACC Status is On, Desired speed is Set, Gap is Set, FV distance is Lower than Gap [H-1].
Providing in wrong time or order causes hazard No unsafe control actions	Adaptive cruise controller provides Brake too late when ACC Status is On, Desired speed is Set, and Speed is Greater than Desired speed [H-3]. Adaptive cruise controller provides Brake too late when ACC Status is On, Desired speed is Set, Gap is Set, and FV distance is Lower than Gap [H-1].
Stopping too soon or applying too long causes hazard	Adaptive cruise controller stops Brake too soon when ACC Status is On, Desired speed is Set, and Speed is Greater than Desired speed [H-3]. Adaptive cruise controller stops Brake too soon when ACC Status is On, Desired speed is Set, Gap is Set, and FV distance is Lower than Gap [H-1].

3) IDENTIFY UNSAFE CONTROL ACTIONS

Each controller has a set of commands (named as control actions in STPA) that it must provide to achieve the system's goals and maintain the system safe. In the third step of STPA, the safety experts must identify which contexts are unsafe to provide or not provide a control action. The context describes the system's state at any given time (combination of the pairs variable and value).

The list of UCAs (unsafe control actions) is necessary for generating the guard transitions of the state machine diagram in our method. For the generation, the system engineer must

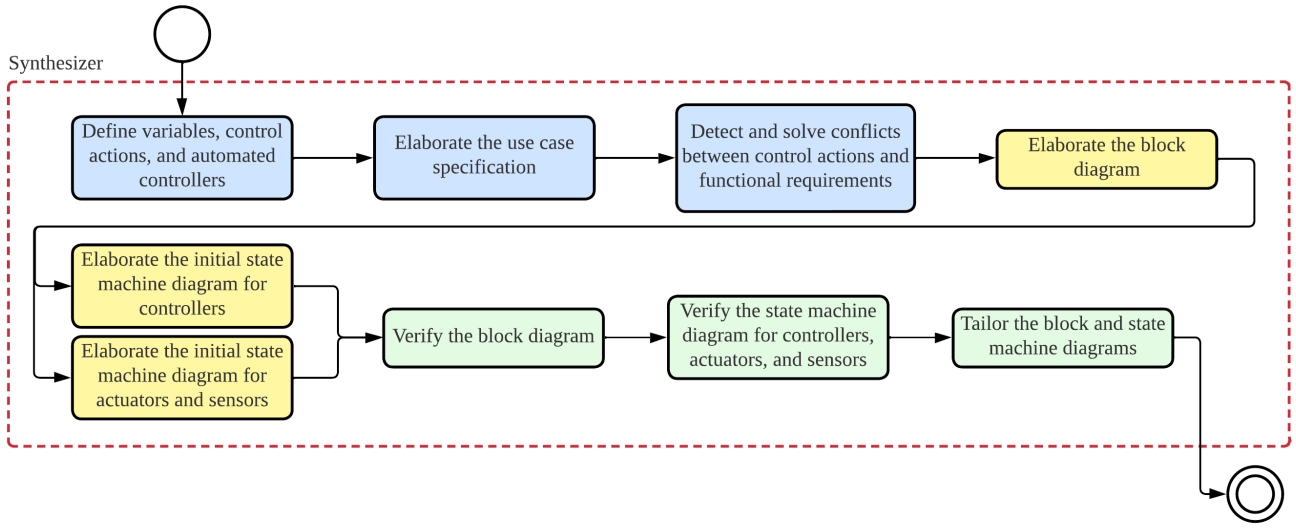

FIGURE 4. Method to synthesize the controller algorithm for safety-critical systems.

write the UCA following the structure proposed by Leveson and Thomas [25]. They suggest that the UCA has four components. The first component is the operator (controller that provides the control action). The second is the type (control action provided or not, provided in wrong time or order, and stopped too soon or applied too long). The third is the control action itself. The last is the context (pair of variables and value), such as "ACC Status is Off".

Table 2 shows the UCAs identified for the *Adaptive cruise controller*. Each UCA is associated with a hazard from Table 1. Therefore, we consider the provision of the control action as hazardous in the following contexts: When the minimum space between the ego and the forward vehicles is not respected (Hazard 1); When the *Driver* thinks the ACC is disengaged (*ACC Status* equals to *Off*) but the ACC controls the vehicle and causes unintended actions (Hazard 2); When an engaged ACC does not follow the parameters provided by the *Driver* when controlling the vehicle (Hazard 3).

Since our focus is on ACC, we will only show the UCAs for ACC. We write all UCAs using the structure of Leveson and Thomas [25], and the name of components, variables, and signals are consistent with the control structure. We consider that both *Accelerate* and *Brake* commands always change the *Speed*. Therefore, we disregard the constant acceleration (to maintain the *Speed*) in the unsafe control action list.

4) IDENTIFY LOSS SCENARIOS

The last activity of the STPA set is to "Identify loss scenarios". The goal is to identify the scenarios and the associated causal factors that can lead to one (or more) losses - in our case, the losses defined in Table 1. We found fifty-four loss scenarios. The quantity of identified scenarios relies on the system engineers' expertise in the system under analysis. The loss scenarios help identify the safety recommendations. The recommendations must be considered in the logic of the controller. However, they are too domain-specific to be systematically generated in the method.

An example of a loss scenario is "Collision between vehicles due to radar sensor obstruction". This scenario is related to the UCA "Adaptive cruise controller not provide Brake when ACC Status is On, Desired speed is Set, Gap is Set, FV distance is Lower than Gap [H-1]". For instance, a safety recommendation is to duplicate the number of radars, links, and input interfaces of ACC. The duplication affects the design, *i.e.*, the block diagrams. Therefore, we decide not to consider the safety recommendations of this step in the synthesis. We still consider the safety restrictions of the STPA step Identify "Unsafe Control Actions". We omit the description of the results of the activity Identify loss scenarios here.

V. A CONTROL ALGORITHM SYNTHESIZER

In this section, we present the method for synthesizing the control algorithm for safety-critical systems. The proposal of Souza *et al.* [4] is a method that combines STPA analysis and SysML modeling. Our synthesizer aims to systematize the *Perform Design* activity from Figure 1, automatically generating the block diagram and initial state machine diagrams. We call initial because the system engineer can tailor the state machine diagram later to consider, for instance, the safety recommendations of the STPA step "Identify Loss scenarios".

From STPA's control structure, our method generates the block diagram and the initial state machine diagrams for actuators and sensors. Moreover, it synthesizes the initial state machine diagram for the automated controllers using the functional requirements and the safety constraints associated with each unsafe control action, as exemplified in the rows "Not providing causes hazard" of Table 2. We do not intend to generate the state machine diagram for human controllers.

We argue that it is difficult to model human behavior because a human controller may not behave exactly in the same way under the same conditions (unlike automated controllers). Moreover, we do not intend to generate the state machine diagram for external information (such as process input and output on the controlled process). It contains information outside the system under analysis, and therefore must be adequately modeled by system engineers with the aid of stakeholders and other system's experts.

Figure 4 illustrates our method. The single circle represents the initial state, the double circle is the final state, and the boxes represent the activities. The arrows indicate the control flow (order) of the activities. The method contains nine activities, where the *Define variables, control actions, and automated controllers*, *Elaborate the use case specification*, and *Detect and solve conflicts between control actions and functional requirements* (boxes with the blue background) serve as preparation for the remaining activities of the method. The activities *Elaborate the block diagram*, *Elaborate the initial state machine diagram for controllers*, and *Elaborate the initial state machine diagram for actuators and sensors* (boxes with the yellow background) are the method's core. The activities *Verify the block diagram*, *Verify the state machine diagram for controllers, actuators, and sensors*, and *Tailor the block and state machine diagrams* (boxes with the green background) are complementary (are not synthesized by our method and must be conducted manually). They include manual verification and tailoring of the diagrams synthesized by the method. The activities to elaborate the initial state machine diagram for controllers, actuators and sensors are shown as concurrent because there is no specific order to synthesize them.

Recall that our method is an automation of the activity *Perform Design* proposed by Souza *et al.* [4], we illustrate our method using the case study in Section IV. In Section IV, we reviewed and applied the first activities of the Souza *et al.* method: *Model Assumptions* (Section IV-A), *Capture requirements* (Section IV-B), *Define the Purpose of the analysis* (Section IV-C1), *Model the control structure* (Section IV-C2), *Identify unsafe control actions* (Section IV-C3), and *Identify loss scenarios* (Section IV-C4).

In the *Perform Analysis* activity, Souza *et al.* [4] suggest using sequence or use case diagrams to check the completeness of the control structure (the components and their interactions). To conduct this activity, we opt for the use case diagrams. However, we suppress the diagram here because (i) we have already confirmed the control structure, and (ii) the use case diagrams are already specified for the Adaptive cruise controller in Section V-B. In the following subsections, we describe the activities of our method (as illustrated in Figure 4).

A. DEFINE VARIABLES, CONTROL ACTIONS, AND AUTOMATED CONTROLLERS

This activity aims to enhance the STPA's control structure information to synthesize a control algorithm compliant with an executable model. Therefore, the input and the output of this activity are the control structure. The activity has three sub-steps. The first sub-step aims to assign types to the variables and define their initial value. Assign types to variables means assessing if it is required to change the variable type (discretized in STPA) to a continuous form (such as numeric) for simulation purposes. The goal of the second sub-step is to classify the control actions as continuous or discrete. The last sub-step aims to categorize the controllers as human or automated. Next, we describe each sub-step and show its application using ACC.

TABLE 3. Transforming discrete variables into continuous.

Variable	Discretized form	Numeric form (numeric interval)
Desired speed	Set	> 0
	Not set	= 0
Speed	Lower than Desired speed	< Desired speed
	Equals to Desired speed	= Desired speed
	Greater than Desired speed	> Desired speed
Gap	Set	> 0
	Not set	= 0
FV distance	Lower than Gap	< Gap
	Equals to Gap	= Gap
	Greater than Gap	> Gap
	Not detected	= 0
FV speed	Lower than Desired speed	< Desired speed
	Equals to Desired speed	= Desired speed
	Greater than Desired speed	> Desired speed
	Not detected	= 0

It is common to discretize the variables of the process model when conducting an STPA analysis. The discretization aids in delimiting the range of values that the variables can assume, easing the analysis. When designing the model of the system, however, it may be better to work with the variables assuming values from the real numbers. The system engineers must complete the definition of variables manually before the synthesis of the control algorithm begins.

To assign types to the variables, the system engineers define the type of a variable as Boolean or numeric. To assign a variable to Boolean, the system engineer must define the variable that will have values true and false. Assigning the variable to numeric brings about inconsistency because of their discretization in STPA – it is impossible to equate a number with a string. To solve this problem, the system engineer must modify each value of the variables abstracted in the STPA analysis, using mathematical operators (such as equals, greater than, and others) to maintain the semantics. Moreover, the system engineers can choose not to change the variable. In this case, the method generates a data type by default (similar to the abstract data type *enum* of programming languages).

For ACC, we maintain the *ACC Status* variable in the discretized form. Therefore, the method will generate a new data type in the block diagram. We transform the other variables into numeric, as shown in Table 3. Moreover, we do not assign any variable to the Boolean type.

The next step is to define the initial value of each variable of the process model. It is a required step in our method

TABLE 4. Excerpt from use case specifications for ACC.

Id	Precondition	Trigger	Postcondition	Include
UC01	Receive signal Turn ACC On (Steering wheel)	-	ACC Status = On	UC05
UC02	Receive signal Turn ACC Off (Steering wheel)	-	ACC Status = Off	UC05
UC03	Speed <8 OR Speed >33	-	ACC Status = Off	UC05
UC04	Receive signal Brake pedal Pressed (Brake controller)	-	ACC Status = Off	UC05
UC05	-	ACC Status (Instrument cluster)	-	-
UC06	Receive signal Set Speed (Steering wheel)	-	Desired speed = Speed	UC07
UC07	-	Desired speed (Instrument cluster)	-	-
UC08	Receive signal Set Gap (Steering wheel)	-	Gap = FV distance	UC09
UC09	-	Gap (Instrument cluster)	-	-
UC10	Receive signal QuickAccel (Steering wheel) AND ACC Status = On AND Speed <33	-	Desired speed = Desired speed + 1	UC07
UC11	Receive signal QuickDecel (Steering wheel) AND ACC Status = On AND Speed >8	-	Desired speed = Desired speed - 1	UC07
UC12	ACC Status = On AND Speed <DesiredSpeed AND (FVDistance = 0 OR Gap <FVDistance)	Accelerate (Engine)	ACC Status = On AND (Speed = DesiredSpeed AND (FVDistance = 0 OR Gap <FVDistance)	-
UC13	ACC Status = On AND Speed <DesiredSpeed AND Gap >FVDistance	Accelerate (Engine)	ACC Status = On AND Speed <= DesiredSpeed AND Gap = FVDistance	-
UC14	ACC Status = On AND Desired speed >0 AND Speed >Desired speed	Brake (Brake controller)	ACC Status = On AND Speed = Desired speed	-
UC15	ACC Status = On AND Gap >0 AND FV distance <Gap	Brake (Brake Controller)	ACC Status = On AND Gap >0 AND FV distance = Gap	-

because when performing the simulation of the model, the controller issues the control actions based on the process model. Therefore, the process model of the controller when the simulation begins must equal to the system's initial state. For the *Adaptive cruise controller*, we define the value of *ACC Status* as *Off* and *Desired speed*, *Speed*, *Gap*, *FV distance*, and *FV speed* with value zero.

We classify the control actions similarly to the variables. Some control actions are a discrete event signal issued by the controller. However, some control actions are applied continuously and have a duration time. Therefore, the system engineer must define if the control actions are discrete or not. In the synthesized model, we intend to associate a variable to each continuous control action to determine the duration of the command, *i.e.*, how long the control action is applied. For ACC, we define the *Accelerate* and *Brake command* (issued by *Driver*) and the *Brake command* (issued by the *Brake controller*) as continuous. The other control actions remain discrete. We assume the *Adaptive cruise controller* always sends a control action to accelerate or break that lasts one unit of time (if necessary, the controller will issue the control action another time). Therefore, the *Accelerate* and *Brake* commands are modeled as a sequence of one-unit time control actions.

In the controller's categorization, the system engineer must define the controllers as human or automated. As stated before, the method does not synthesize the control algorithm for human controllers because they may not behave exactly in the same way when exposed to the same input/set of values (differently from automated controllers). For ACC, we define the *Adaptive cruise controller* and *Brake controller* as automated controllers and *Driver* as human controller.

B. ELABORATE THE USE CASE SPECIFICATION

System engineers can identify some functional requirements using STPA. The ''Identify unsafe control actions'' step from STPA provides a list of control actions that maintain the system safe (type ''Not providing causes hazard''). However, STPA only identifies the functional requirements that impact safety (*i.e.*, that can lead to a hazard or loss). Therefore, STPA does not consider the behavior of commands that are not hazardous. The input of this activity is the requirements diagram, and the output is the use case specification.

The requirements diagram (Figure 2) shows the functional requirements for ACC system in a textual form. To formalize the requirement, we employ the use cases (focusing on the use case specification). The use case specification must have the following elements: Use case id (a unique identification), the primary actor (the component responsible for initiate the use case), the precondition (conditions that must be satisfied to begin the use case), the trigger (the event that initiates the use case, *e.g.*, the control action itself), the basic flow steps (that gives the sequence of steps for the main success scenario), and the postcondition (the conditions that must be satisfied at the end of the use case).

Table 4 shows the excerpt of the use case specifications, describing the id, precondition, trigger, postcondition, and if the use case includes others or not. The primary actor of the use cases is the *Adaptive cruise controller*. We considered the following use cases: Engaging ACC (UC01); Disengaging ACC at Driver's command (UC02), when speed is out of boundaries (UC03), and after Driver brakes the vehicle (UC04); Feedbacking ACC Status to Driver (UC05); Setting Desired speed (UC06); Feedbacking Desired speed to Driver (UC07); Setting Gap (UC08); Feedbacking Gap to Driver (UC09); Increasing Desired speed

(UC10); Decreasing Desired speed (UC11); Accelerating to reach Desired speed (UC12) or desired Gap (UC13); Braking to reach Desired speed (UC14) or desired Gap (UC15).

The precondition of UC01 is the receiving of *Turn ACC On* (*Steering wheel*). It means the *Adaptive cruise controller* receives the *Turn ACC On* command from *Steering wheel* component. UC05 does not have a precondition, but it is included in other use cases. Therefore, it only contains a trigger (representing the signal sending).

C. DETECT AND SOLVE CONFLICTS BETWEEN CONTROL ACTIONS AND FUNCTIONAL REQUIREMENTS

Conflicts may arise between a control action and a functional requirement or between two (or more) control actions or functional requirements [26]. In our method, two control actions that (or a control action and a functional requirement) have the same context and opposite effects, conflict with each other. They may be the result of wrong or incomplete analysis. The conflicts may result in an unfeasible logic of the controller. This activity aims to identify conflicts arising from the identified UCAs. We use the identified UCAs as input to identify conflicts. After identifying the conflicts, we solve them by refining the context so that conflicts do not exist. Solving conflict is manual. We only address the conflicts arising from the opposing unsafe control actions.

Algorithm 1 shows the pseudocode to detect conflicts between commands. The method uses the context table [25] to verify in which contexts a conflict can arise. The context table contains all possible combinations of the pair variable-value of the process model. Moreover, the method adds a variable in the context table: the controller's last signal received. Considering the *Adaptive cruise controller*, it receives ten signals and has seven variables (*ACC Status*, *Desired speed*, and *Gap* with two values; *Speed* with three values; *FV distance* and *FV speed* with four values). The context table for this controller has 3840 rows (or possible combinations). We obtain the number of rows by multiplying the number of possible values from each variable with the number of signals received (e.g., $10\times2\times2\times2\times3\times4\times4 = 3840$).

Table 5 provides information to synthesize the *Adaptive cruise controller* based on functional requirements (use case specifications from Section V-B) and STPA (''Not providing causes hazard'' UCAs from Section IV-C3). We do not consider the other UCAs types because they are hazardous when provided anytime or at the wrong time/order. The *Guard Condition* is a Boolean expression representing the context (STPA) or the precondition for a transition when it is true. The *State Id* is the name of the control action (prefix CA) or the use case (prefix UC) followed by its identifier. The *Trigger* is the event (*e.g.*, Accelerate, Brake, or none ''−'') that the controller issues. For states that do not have a trigger event, the *Effect* is an internal behavior performed during the transition (the postcondition). Table 5 is based on Table 4, but we remove the use cases that do not have a precondition (because they are included by other use cases).

The method iterates over the 3840 contexts from the context table, checking if the current context satisfies the Guard Condition from Table 5 (*i.e.*, if the *Guard Condition* evaluates true for the current context). None or one *Guard Condition* evaluating true means there is no conflict in the current context. When two (or more) *Guard Conditions* evaluate true, there is a conflict or reinforcement between the commands. We call as reinforcement when the same context satisfies a safety property and a functional requirement, issuing the same command. For the *Adaptive cruise controller*, we observed conflicts between the UC04, UC06, UC08, UC10, and UC11 with the CA01, CA02, UC12, UC13, UC14, and UC15. Since the *Guard Condition* of the first set of commands is mostly feedback receiving, there is a conflict between both sets. For instance, the following context satisfies both UC08 and UC13: *ACC Status = On AND Desired speed > 0 AND Speed < Desired speed AND Gap > 0 AND FV distance < Gap AND FV Speed <= Desired speed AND Feedback = Set Gap (Steering wheel)*.

Therefore, we need to change the context (control action) or precondition (use case) of the second set to avoid these conflicts. We solve it by adding the following conditions to all commands into the second set (CA01, CA02, UC12, UC13, UC14, and UC15): *LastSignalReceived != Brake pedal Pressed (Brake controller) AND LastSignalReceived != Set Speed (Steering wheel) AND LastSignalReceived != Set Gap (Steering wheel) AND LastSignalReceived != QuickAccel (Steering wheel) AND LastSignalReceived != QuickDecel (Steering wheel)*. Moreover, we notice redundancy between (i) CA01 and UC15 and (ii) CA02 and UC14. Therefore, we opt to maintain both control actions (CA01 and CA02) and discard the use cases (UC14 and UC15).

In the previous example, the issuances of commands at the same time are undesirable. However, in some cases, the controller must provide two or more control actions in a specific order. For these cases, system engineers can define the order of two (or more) commands in this activity. Once we identify the possible conflicts between commands (control actions and use cases), our method can synthesize the block diagram.

D. ELABORATE THE BLOCK DIAGRAM

This activity aims to elaborate the system's architecture in the form of a block diagram. Inside each block, there is a state machine diagram that determines the behavior of the block. The input of this activity is the control structure from the STPA, and the output is the block diagram. The block diagram describes the blocks, their connections, variables, signals, and data flows. The ''block diagram'' we use is from AVATAR [14] and encompasses a block definition diagram and internal block diagram.

Algorithm 2 describes the straightforward transformation between the control structure of STPA into the block diagrams of SysML. The method transforms STPA components (actuator, controlled process, controller, and sensor) into the SysML blocks. The method also transforms the connections

Algorithm 1 Detecting Conflicts Between Commands

Require: Controller's process model, control actions, and use case specifications
Ensure: Conflicting commands and respective context when they conflict

```
Generate the context table
for each context in the context table do
    conflictingCommands ← [ ]
    for each command issued by the controller do        ▷ Command is a control action or a functional requirements
        if a Guard Condition from the command evaluates to true within the current context from the context table then
            conflictingCommands.push(currentContext)
        end if
    end for
    if conflictingCommands.length ≥ 2 then
        Exhibit the conflicting commands and the current context
    end if
end for
```

TABLE 5. Controller's command (trigger) and their respective context (guard condition).

State Id	Guard Condition	Trigger	Effect
CA01	ACC Status = On AND Gap >0 AND FV distance <Gap	Brake (Brake Controller)	-
CA02	ACC Status = On AND Desired speed >0 AND Speed >Desired speed	Brake (Brake Controller)	-
UC01	Receive signal Turn ACC On (Steering wheel)	-	ACCStatus = On
UC02	Receive signal Turn ACC Off (Steering wheel)	-	ACCStatus = Off
UC03	Speed <8 OR Speed >33	-	ACCStatus = Off
UC04	Receive signal Brake pedal Pressed (Brake controller)	-	ACCStatus = Off
UC06	Receive signal Set Speed (Steering wheel)	-	DesiredSpeed = Speed
UC08	Receive signal Set Gap (Steering wheel)	-	Gap = FVDistance
UC10	Receive signal QuickAccel (Steering wheel) AND ACC Status = On AND Speed <33	-	DesiredSpeed = DesiredSpeed + 1
UC11	Receive signal QuickDecel (Steering wheel) AND ACC Status = On AND Speed >8	-	DesiredSpeed = DesiredSpeed - 1
UC12	ACC Status = On AND Speed <DesiredSpeed AND (FVDistance = 0 OR Gap <FVDistance)	Accelerate (Engine)	-
UC13	ACC Status = On AND Speed <DesiredSpeed AND Gap >FVDistance	Accelerate (Engine)	-
UC14	ACC Status = On AND Desired speed >0 AND Speed >Desired speed	Brake (Brake controller)	-
UC15	ACC Status = On AND Gap >0 AND FV distance <Gap	Brake (Brake Controller)	-

between STPA components into connections and the standard ports in the SysML blocks. Each variable in the control structure model is also a variable in the SysML block. The method uses the information from the Define variables, control actions, and automated controllers activity to assign the type and the initial state of the variables in the block diagram. The method creates the corresponding data type if a variable has no type assigned. Moreover, the method creates a variable corresponding to the duration time for continuous control actions. Each control action and feedback are signals in the SysML block. A step-by-step description of the Algorithm is provided using the ACC system.

Figure 5 depicts the block diagram generated from the control structure (Figure 3) of ACC. Due to space limitations, we resize the blocks. Therefore, some signals may be missing in Figure 5.

Some tools enable code generation. Although code generation is not our focus, the method suits the name of components, variables, control actions, and feedback defined in STPA to satisfy the rules of variable names in a programming language (removing spaces or replacing them with an underscore, for instance).

Following the algorithm, the first step is to create a block for each component of the control structure. Therefore, the method synthesizes the following blocks: *Driver*, *Accelerator_Pedal*, *Steering_Wheel*, *Instrument_Cluster*, *Brake_Pedal*, *Radar*, *Adaptive_Cruise_Controller*, *Engine*, *Brake_Controller*, *Service_Brake*, *Wheel_Speed_Sensor*, and *Wheels*. We use the Snake_Case convention to create the name of the blocks (we replace the spaces for underscores, and each word has the first letter capitalized). Additionally, *Driver* receives one external information (*FV distance*), and the *Radar* receives two external information (*FV distance* and *FV speed*). Therefore, the method creates an additional block named *EXT*, responsible for the external information.

Algorithm 2 Synthesizing the Block Diagram

Require: Functional control structure (STPA)
Ensure: Block diagram (SysML/AVATAR)

for each component in the control structure **do**
    Create a block with the name in the Snake_Case convention
    for external information received by a controller **do**
        Create a block with the name *EXT*
    end for
end for
for each connection in the control structure **do**
    if a channel between two components does not exist **then**
        Create an asynchronous channel between both blocks
    end if
end for
for each component in the control structure **do**
    if the component is a controller or controlled process **then**
        Create a variable in the current block using the PascalCase convention
        Define the variable type as set in the first activity of the method
        if no type was assigned **then**
            Create a new type with the name *Variable_DataType* ▷ *Variable* is the name of the variable in PascalCase
        end if
    end if
    if the component is an actuator **then**
        if the received control action is continuous **then**
            Create a numeric variable in the block with the name *DUR_SignalName*
        end if
    end if
    if the component is a sensor **then**
        for each feedback sensed **do**
            Create a variable equal to the defined in the process model of the controlled process or controller
        end for
    end if
end for
for each component in the control structure **do**
    for each control action or control action enforcement **do**
        Create a signal (output) named *sendSignal* using the camelCase convention ▷ *Signal* is the name of the signal
    end for
    for each feedback **do**
        Create a signal (input) named *receiveSignal* using the camelCase convention ▷ *Signal* is the name of the signal
    end for
end for
for each channel in the block diagram **do**
    Link the signal in of one component (*receiveSignal*) with the signal out of the other component (*sendSignal*)
end for

Next, the method defines the connections between the blocks. The control structure has directional connections (for instance, the downward arrows are control actions and the upward arrows are feedback). In the block diagram, the connections are bidirectional. Therefore, the method creates a connection in the block diagram following the connections in the control structure without duplicity (e.g., there are two one-way connections between the *Adaptive cruise controller* and *Brake controller* in the control structure. In the block diagram, there is one bidirectional connection). The hollow squares in the edge of the connections mean that the connector type is asynchronous.

Once the blocks are connected, the method creates the variables. The variable name follows the PascalCase convention (*i.e.*, it capitalizes the first letter of each compound word in the variable). The variables receive the same data type and initial value as defined in the *Define variables, control actions, and automated controllers* activity. For the

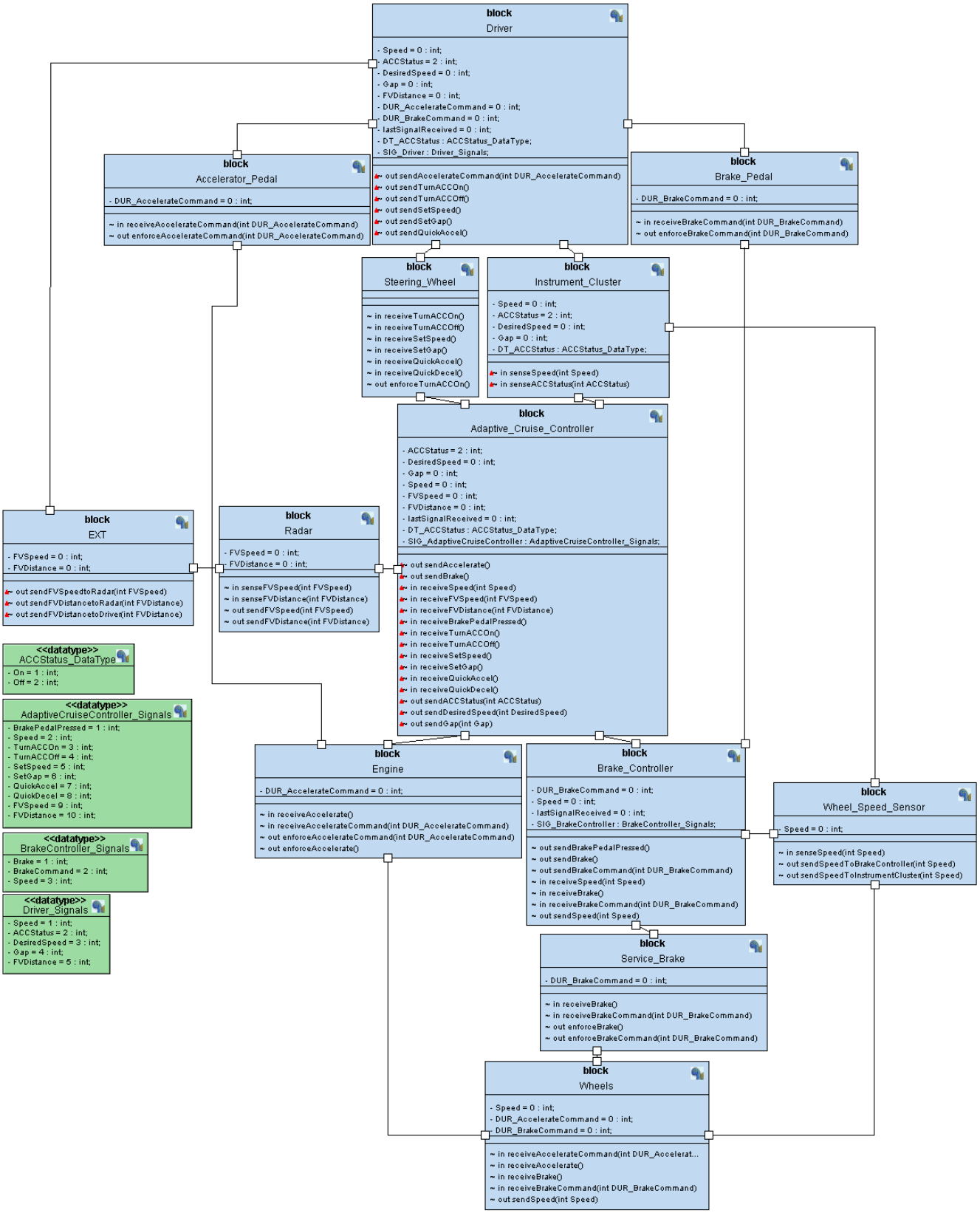

FIGURE 5. Block diagram for ACC synthesized by our method.

ACC Status variable (not defined as a numeric or Boolean variable), the method creates a new data type named *ACC-Status_DataType*. The name of the data type is composed of the variable name (in PascalCase) and the suffix *_DataType*. The method creates the variable *ACC Status* as a numeric (because it is similar to the *enum* type). However, the variable only assumes the values of the data type. Additionally, the method creates a variable in the block named *DT_ACCStatus*, with the prefix *DT_* representing the data type.

Next, the method creates the signals. Since the component can receive and send a signal with the same name, we differentiate the feedback and control actions by adding the prefix *receive* (*sense*, for sensors) and *send* (*enforce*, for actuators) in the signal name. Using the *Instrument cluster* as an example, it receives three signals: *ACC Status* (*Adaptive cruise controller*), *Gap* (*Adaptive cruise controller*), and *Speed* (*Wheel speed sensor*). Therefore, the method creates the following signals (type: input): *senseACCStatus(int ACCStatus)*, *senseGap(int Gap)*, and *senseSpeed(int Speed)*. Similarly, it sends three signals: *ACC Status* (*Driver*), *Gap* (*Driver*), and *Speed* (*Driver*). Therefore, the method creates the following signals (type: output): *sendACCStatus(int ACCStatus)*, *sendGap(int Gap)*, and *sendSpeed(int Speed)*. The name of the signals follows the camelCase convention (similar to PascalCase, except for the first letter of the name that is not capitalized).

Control actions usually do not have a parameter. However, continuous control actions (as defined in Section V-A) must have the duration as the parameter. For instance, *Driver* sends the *Brake Command* to the *Brake pedal*. Therefore, we create an associated variable named *DUR_BrakeCommand*, where the prefix *DUR_* means duration. Additionally, the method creates a variable with the same name in the block.

The last step is to connect the signals in the connections. We need to define that when *Driver* issues the signal *sendTurnACCOn()*, the *Steering wheel* receives this signal through the *receiveTurnACCOn()* signal. The signal's data type contains the list of the signals that the controller can receive. Using *Driver* as an example, it can receive the signals *ACCStatus*, *Speed*, *Desired speed*, *Gap*, and *FVDistance*. Additionally, the method creates a numeric variable named *LastSignalReceived*. To that end, the method also creates a data type for each controller, where the name of the data type is the controller's name in PascalCase followed by the suffix *_Signals*. This data type is associated with the *LastSignalReceived* variable.

Once the method synthesizes the block diagram, it is possible to elaborate the state machine diagram. We separate this into two activities, where the first synthesize the initial state machine diagram for controllers and the second for actuators and sensors. The method performs both activities concurrently.

E. ELABORATE THE INITIAL STATE MACHINE DIAGRAM FOR CONTROLLERS

Each block from the block diagram has a state machine diagram associated with it. The method aims to generate a state machine diagram to controller's blocks. We also synthesize the state machine diagrams for actuators and sensors, but they are generally simple. Algorithm 3 shows how the method synthesizes the state machine diagram for automated controllers.

We consider that controllers of real-time safety-critical systems have three states. The first state is *WaitingForInformation*, where the system is idle waiting to receive some information (control action or feedback from systems components or external system). In the second state, named *InformationReceived*, the system receives information from a higher-level component, sensor, or external system and must decide on a control action to issue. If the controller does not have a control action to issue, it returns to the *WaitingForInformation* state. The last state is *CommandIssued*, where the system issued a control action to maintain the system safe. The only path possible is to return to the *WaitingForInformation* state.

Figure 6 depicts the initial state machine diagram for the *Adaptive cruise controller* synthesized by our method. Due to space limitations, we suppress some elements from the Figure 6, such as the following signals (*receiveTurnACCOff*, *receiveSetSpeed*, *receiveSetGap*, *receiveQuickAccel*, *receiveQuickDecel*, *receiveSpeed*, and *receiveFVDistance*) and the following commands (CA02, UC02, UC02_UC05, UC03, UC03_UC05, UC04, UC04_UC05, UC06, UC06_UC07, UC08, UC08_UC09, UC10, UC10_UC07, UC11, UC11_UC07, UC12, and UC13).

The initial state is *WaitingForInformation*. Therefore, the Adaptive cruise controller waits for receiving feedback, control actions from higher-level components, or external communication. The concave pentagon represents the receiving signal. An example of information received is *receiveTurnACCOn()*, sent by the *Steering wheel*. The remaining signals (type: input) defined in the block diagram follow the same idea. After receiving a signal, the control flows to the state *InformationReceived*.

The next step is to determine which commands the controller must provide and in which context. The method resorts to the third step of STPA (Identify unsafe control actions) and the SysML diagrams (use case specifications) to determine the commands to be issued (see Table 5). The third step of STPA provides a list of UCAs of "not providing causes hazard" type (*i.e.*, the controller must issue these control actions to maintain the system safe). The use case specification contains the functional requirements that complement the controller. Therefore, the *Adaptive cruise controller* may provide twelve different commands – as stated in Table 5 (we excluded UC14 and UC15 due to reinforcement reasons). The value of the column *State Id* is the name of the state.

Algorithm 3 Synthesizing the Initial State Machine Diagram for Controllers

Require: Block Diagram (SysML/AVATAR), Controller's control actions and functional requirements
Ensure: State Machine Diagram (SysML/AVATAR)

1: Create an initial state
2: Create the *WaitingForInformation* state
3: Add a transition between the states initial and *WaitingForInformation*
4: Create the *InformationReceived* state
5: Create the *CommandIssued* state
6: **for** each received signal from the controller's block **do**
7: Create the receive signal state
8: Add a transition between the states *WaitingForInformation* (source) and receive signal (target)
9: Add a transition between the states receive signal (source) and *InformationReceived* (target)
10: **end for**
11: Add a transition between the states *InformationReceived* (source) with *WaitingForInformation* (target) with the guard *else*
12: **for** each unsafe control action from ''not provided'' type and functional requirements **do**
13: Create the state *CAindex* ▷ *index* is the identifier of the control action
14: Create the send signal state ▷ *signal* is the control action
15: Add a transition between the states *InformationReceived* and *CAindex* with the guard equal to the context of the UCA
16: Add a transition between the CAindex and send signal state
17: **while** there are control actions to be issued in a specific order **do**
18: Create the state *CAindex_CA2index* ▷ *CA2index* is the identifier of the control action issued in order
19: Create the send signal state to issue the second control action
20: Add a transition between the send signal state (first CA) and *CAindex_CA2index*
21: Create a transition between the send signal state (second CA) and *CommandIssued*
22: **end while**
23: **if** there are not control actions to be issued in a specific order **then**
24: Create a transition between the send signal state and *CommandIssued*
25: **end if**
26: **end for**
27: **for** each use case specification **do**
28: Create the state *UCindex* ▷ *index* is the identifier of the use case
29: **if** there is a *precondition* **then**
30: Add a transition between the states *InformationReceived* and *UCindex* with the guard equals to *precondition*
31: **end if**
32: **if** there is a *trigger* **then**
33: Create the send signal state ▷ The signal is the trigger
34: Add a transition between the states *UCindex* and send signal
35: **end if**
36: **if** there is a *postcondition* and there is not a trigger **then**
37: Add a transition between the states send signal and *CommandIssued* with the effect equals to the *postcondition*
38: **end if**
39: **while** there are *included use cases* **do**
40: Create the state *UCindex_UC2index* ▷ *UCindex_UC2index* is the identifier of the second use case
41: Repeat the steps from lines 29 to 38, replacing *UCindex* for *UC2index*
42: **end while**
43: **if** there are not *included use cases* **then**
44: Add a transition between the send signal state and the *CommandIssued*
45: **end if**
46: **end for**
47: Add a transition between the *CommandIssued* and *WaitingForInformation*

The transition to these states has a guard condition, which is the value of the *Context (Preconditions)* column. The representation of a control action (signal type: output) is a convex pentagon. If no guard condition is satisfied, there is a path with a *[else]* where the system returns from *Information-Received* state to *WaitingForInformation* state. In ACC, each

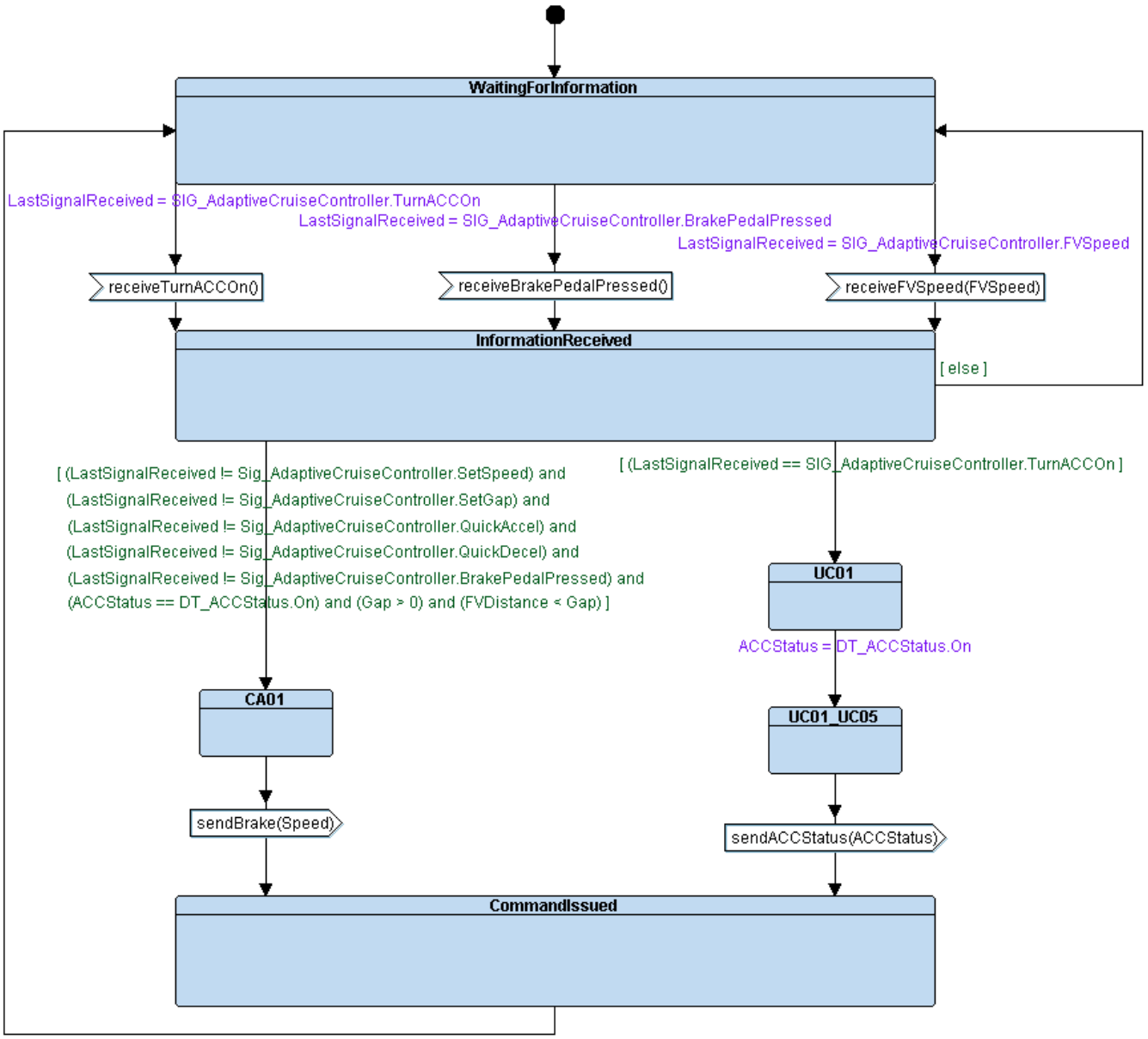

FIGURE 6. State machine diagram for the adaptive cruise controller.

context satisfies a single command. However, the controller must provide two (or more) commands in the same context.

The method uses the information provided in the activity *Detect and solve conflicts between control actions and functional requirements* (Section V-C) to determine the order of the commands in the state machine diagram (for instance, if a use case includes others or if a control action must be issued before other). The UC01 includes the UC05 (see Table 4 and Figure 6). The method creates the state UC01 (with its respective guard condition) and, after that, it creates the state UC01_UC05. The state indicates that the UC01 includes the UC05. The method places both use cases one after the other and follows the same rules defined previously (preconditions are the guard conditions, and if there is a trigger, it sends a command).

The method follows the same algorithm to synthesize the state machine diagram for the *Brake controller*. The method does not synthesize the state machine diagram for *Driver* controller because it is human. Therefore, the system engineer must design it.

F. ELABORATE THE INITIAL STATE MACHINE DIAGRAM FOR ACTUATORS AND SENSORS

Usually, actuators and sensors have a simple operation. The actuator stays in a standby state, waiting to receive signals. After receiving the signal, the actuator performs the mechanical force required to enforce the command. Similarly, the sensor observes the controlled process (or another component). When it senses a change, it sends feedback to the controller (i.e., the change of the observed variable).

The method synthesizes the state machine diagram for actuators and sensors since actuators receive commands from the controller and enforce them, and sensors observe the controlled process. It excludes external sensors and sensors coupled to the process input or process output from the controlled process. Algorithm 4 shows how the method creates the state machine diagram for actuators and sensors.

The method synthesizes the state machine diagram for actuators using the following model: There is a state named *Standby*, which represents the actuator in its idle state (*i.e.*, waiting for receive commands). There is a connection between the *Standby* state and each signal (type: input) of the actuator. The signal represents the control actions issued by the controller. When the actuator receives the signal, it enters a new state named *signalReceived* (replacing signal with the name of the signal received). Then, it immediately forwards the corresponding signal (type: output). It enters in a new state named *signalEnforced* (replacing signal with the name of the signal sent) and goes back to the *Standby* state.

The state machine diagram for sensors follows the same idea. However, instead of *Standby*, we use the name Sensing. In the intermediate state, instead of the name *signalReceived*, we use *signalSensed*. Figure 7 depicts the initial state machine diagram for the sensor *Instrument cluster* (left side of the Figure 7) and the actuator *Steering wheel* (right side of the Figure 7). Due to space limitation, we suppress the signals for sensing the *Desired speed* and *Gap* (*Instrument cluster*) and the signals to enforce the commands *SetSpeed*, *SetGap*, *QuickAccel*, and *QuickDecel* (*Steering wheel*).

G. VERIFY THE BLOCK DIAGRAM

The method generates the block diagram from the control structure. However, there may be inconsistencies in the block diagram, probably because the STPA analysis is incorrect or incomplete. The verification is manual, and it includes performing a manual comparison between the block diagram (STPA) and the initial control structure (STPA). Therefore, it is the responsibility of the system engineer to verify if the block diagram is similar to the control structure.

H. VERIFY THE STATE MACHINE DIAGRAMS FOR CONTROLLERS, ACTUATORS, AND SENSORS

Following the same idea of the previous section, the system engineer must verify the initial state machine diagram for controllers, actuators, and sensors. In this activity, the system engineer can manually check the initial state machine diagrams to discover if the STPA analysis and the use case specifications are complete and consistent.

I. TAILOR THE BLOCK AND STATE MACHINE DIAGRAMS

The last activity of the proposed method is to tailor the block diagram and the state machine diagrams. In this step, the system engineer must change the block diagram or state machine diagrams (if necessary) and create the state machine diagrams that are not synthesized by our method (human controllers, controlled process, and external information).

First, we modify the block diagram. We rename the block *EXT* to *EXT_Forward_Vehicle*. This change has no impact on the model, and we only did it to improve the readability of the block. Therefore, we need to design the state machine diagram for the following blocks: *EXT_Forward_Vehicle*, *Driver*, and *Wheels*.

We define the behavior of the *EXT_Forward_Vehicle* (including the initial value of its variables) based on (i) our expertise in the system under analysis, (ii) and the knowledge acquired from our previous models (manually designed, without the method support). For the *EXT_Forward_Vehicle*, we elaborate the state machine diagram considering the system does nothing in the first fifty units of time. We did it to guarantee that *Driver* accelerates the vehicle and engages ACC before detecting a vehicle ahead. After this time, the forward vehicle block sends the *FVDistance* equals to one hundred followed by sixty information about the forward vehicle's speed and distance (each message after one unit of time). The initial value of the *FVSpeed* is twenty-eight. We use two variables to change the speed at each iteration. The first variable is the *randomValue*, which assumes the values zero, one, or two. The second is *isIncrementing*, which receives the values zero (false) or one (true). Both variables receive the return of the TTool's [27] function *RANDOM0(minValue, maxValue)*. The function generates integer numbers between (and including) *minValue* and *maxValue* using a discrete uniform distribution. If *isIncrementing* is one, it means that we add the *randomValue* to the *FVSpeed*. Otherwise, we decrement the *randomValue* from the *FVSpeed*.

After sixty units of time, the block sets the value of *FVSpeed* and *FVDistance* to zero and sends them to the radar, indicating there is no forward vehicle detected. We did a minor change in the *Adaptive cruise controller*, adding the following instruction after the *FVSpeed* receiving: *FVDistance = FVDistance + FVSpeed - Speed*. Without this instruction, it is impossible to calculate the *FVDistance* since there is no way to simulate the current distance between vehicles without this mathematical expression.

We model the block *Wheels* with two main states: *Stopped* and *Moving*. The vehicle starts in the *Stopped* state and goes to *Moving* state when *Driver* presses the *Accelerator* pedal, *i.e.*, it receives the *receiveAccelerateCommand(DUR_AccelerateCommand)* from *Driver*. We also consider that the acceleration and deceleration rate is two. That means the vehicle accelerates incrementing the speed in two units per time and brakes decrementing the speed in two units per time. Moreover, *Wheels* also receives the commands to accelerate and brake through ACC. We implement the natural deceleration of the vehicle using timers. When the vehicle remains without receiving any command to accelerate or decelerate for more than two units of time, the vehicle will decrease the speed in one unit. When the speed is zero, it returns to the *Stopped* state.

To test all commands, we model *Driver* using the following behavior. First, *Driver* accelerates the vehicle until the velocity of thirty units. Then, *Driver* turns on ACC and sets the

Algorithm 4 Synthesizing the Initial State Machine Diagram for Actuators and Sensors

Require: Block Diagram (SysML/AVATAR)
Ensure: State Machine Diagram (SysML/AVATAR)

for each actuator in the block diagram **do**
    Create the *Standby* state
    for signal received from the actuator block **do**
        Create the receive signal state
        Add a transition between the states *Standby* and receive signal
        Create the state *signalReceived* ▷ signal is the name of the receive signal
        Add a transition between the states receive signal and *signalReceived*
        Create the send signal state
        Add a transition between the states *signalReceived* and the send signal state
        Create the state *signalEnforced* ▷ signal is the name of the send signal
        Add a transition between the states *signalEnforced* and Standby
        Create the receive signal state
    end for
end for
for each sensor in the block diagram **do**
    Create the *Sensing* state
    for signal received from the actuator block **do**
        Create the receive signal state
        Add a transition between the states *Sensing* and receive signal
        Create the state *signalSensed* ▷ signal is the name of the receive signal
        Add a transition between the states receive signal and *signalSensed*
        Create the send signal state
        Add a transition between the states *signalReceived* and the send signal state
        Create the state *signalSent* ▷ signal is the name of the send signal
        Add a transition between the states *signalSent* and Standby
        Create the receive signal state
    end for
end for

current *Speed* as the *Desired speed*. Then, *Driver* presses the brake of the vehicle for two units of time (decrementing the Speed in four units). It deactivates ACC (UC04). After that, *Driver* turns on ACC and presses the buttons *QuickAccel* and *QuickDecel*. *Driver* waits for the *Adaptive cruise controller* to control the vehicle's *Speed* for twenty units of time, and when a forward vehicle is detected, it sets the current distance (*FVDistance*) as the *Gap*. It waits for sixty units of time, observing the *Adaptive cruise controller* controlling the vehicle's *Speed*. When no forward vehicle is detected, *Driver* turns ACC off and ends the execution (final state).

VI. CHECKING THE IMPLEMENTATION OF THE ALGORITHMS

In the previous section, we elaborated the block and initial state machine diagrams (for automated controllers, actuators, and sensors) manually, following the Algorithms 1 to 4. To check if the algorithms are consistent, we implemented a prototype of the synthesizer using the algorithms to generate the diagrams automatically. We also integrated the prototype of the synthesizer's module in a STPA tool named WebSTAMP [28]. WebSTAMP is a web application that supports STPA analyses.

We later compared the diagrams generated by the tool and the diagrams generated manually. Disregarding the synthesizer's module, STPA supports five activities from the Souza *et al.* [4] method: *Model assumptions* (SysML); *Define the purpose of the analysis* (STPA); *Model the control structure* (STPA); *Identify unsafe control actions* (STPA); *Identify loss scenarios* (STPA). Currently, WebSTAMP does not support the activity *Capture requirements*.

The synthesizer's module adds four features to the extended WebSTAMP. The first three features correspond to the following activities of our method: *Define variables*, *Elaborate the use case specification*, and *Detect conflicts between control actions and functional requirements* (Algorithm 1). The last feature generates an XML file containing the results of the activities: *Elaborate the Block Diagram* (Algorithm 2), *Elaborate the initial state machine for controllers* (Algorithm 3), and *Elaborate the initial state machine for actuators and sensors* (Algorithm 4).

In the *Define variables* activity, the systems engineer can assign a type to the variables (numeric, Boolean, or a new data type), define the new values (numeric interval for numeric variables, and true/false for the Booleans), and determine the initial state. Moreover, we do not allow to define a variable as

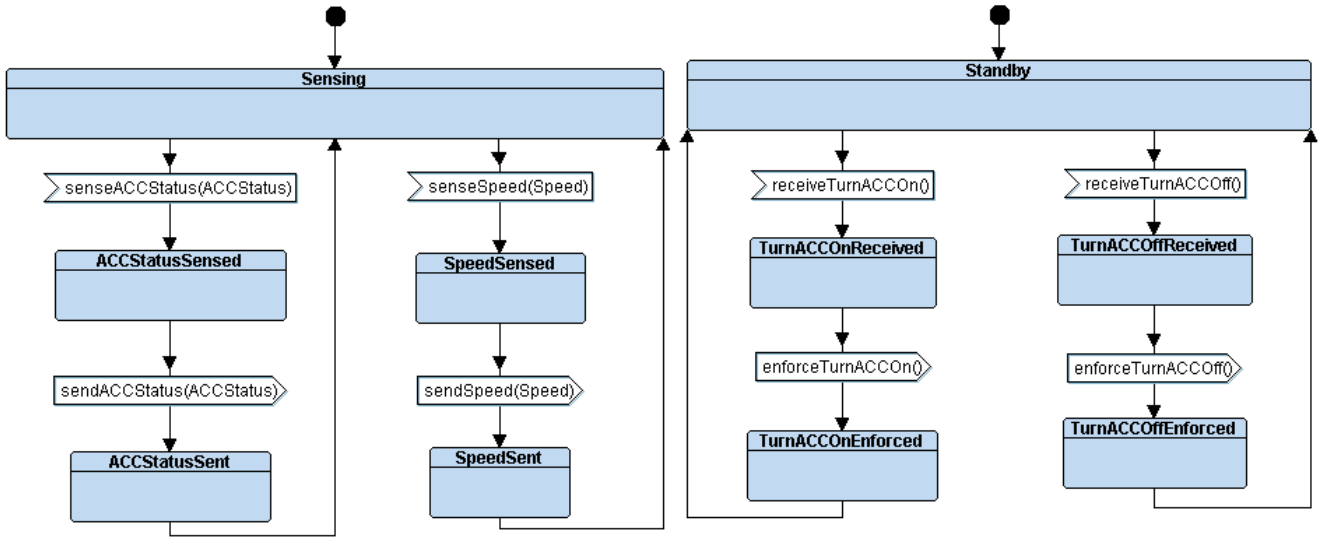

FIGURE 7. State machine diagram for the instrument cluster (left) and steering wheel (right).

Boolean in the tool if the number of states is not exactly two. The first activity of our method also requires the definition of type of control action (as continuous or discrete) and type of controller (as human or automated). However, WebSTAMP demands these definitions in the *Model the control structure* activity. With tool support, it was possible to define the variables, control actions, and controllers in the same way as manually.

In the *Elaborate the use case specification* activity, the tool provides an environment to store the following attributes for each use case: identifier, name, preconditions, trigger, effects, postconditions, and if there is an included use case. System engineers define the identifier, name, and effects using natural language. The preconditions, trigger, inclusion of another use case, and postconditions are defined systematically. The trigger has the controller's output defined in the *Model the control structure* activity, and the included use case has a list of the use cases already created in the system (and the option ''No'' if the current use case does not include other).

Figure 8 depicts the preconditions of the use case Engaging ACC (UC01). When the *ACC Status* is *Off*, and the controller receives the *Turn ACC On* from the *Steering Wheel*, the precondition is satisfied. The token ''−'' indicates that the variable is not relevant to the current use case specification. The addition of the postcondition follows the same design as the precondition. The tool creates the precondition using only the AND operator. Therefore, the system engineers must split preconditions with the OR operator (*e.g.*, UC03) in two use cases, one with precondition $Speed < 8$ (named UC03_A) and another with precondition $Speed > 33$ (named UC03_B). That was the only difference between the activity performed manually and with tool support.

The tool generates the context table in the *Detect and solve conflicts between control actions and functional requirements* activity. If two or more control actions (or use cases) satisfy a context of the table, it means there is a possibility of conflict or reinforcement. The tool exhibits the conflicting commands and the context where the conflict occurs. The system engineers must solve the conflicts manually (changing the context of control actions or the preconditions of use case specifications). The tool found 6304 conflicts or reinforcements (considering the commands from Table 5).

Once the STPA analysis and the pre-steps of the model are complete, the tool generates the XML. We use TTool [27] as the SysML toolkit to run our models. The only difference between the diagrams generated manually in Section V and with tool support is the arrangement of the elements in the diagrams. Moreover, TTool has a syntax checker that finds errors in the model (such as inconsistency of names, signals not connected, malformed guard conditions, and others). The model generated by the tool does not have any syntax error.

Figure 9 depicts a fragment of the initial state machine diagram for the *Adaptive cruise controller* generated by the tool prototype. Due to space limitations, we suppress some elements of Figure 9, such as some signals: *receiveBrakePedalPressed()*, *receiveTurnACCOn()*, *receiveTurnACCOff()*, *receiveSetSpeed()*, *receiveQuickAccel()*, *receiveFVSpeed(FVSpeed)*, and *receiveFVDistance(FV Distance)*, and commands: *CA01*, *UC01*, *UC01_UC05*, *UC02*, *UC02_UC05*, *UC03_A*, *UC03_A_UC05*, *UC04*, *UC04_UC05*, *UC06*, *UC06_UC07*, *UC08*, *UC08_UC09*, *UC10*, *UC10_UC07*, *UC11*, *UC12_A*, *UC11_UC07*, *UC12_B*, *UC14*, and *UC15*.

The fragmented diagram shows the relevant parts of the diagram synthesized for the *Adaptive cruise controller*. The complete diagram has 64 transitions, 28 states, 26 signals, 19 effects, and 16 guard conditions, and some of the elements overlap in its visual presentation, making its readability difficult. The complete diagram contains all control actions and the use cases defined in the previous activities.

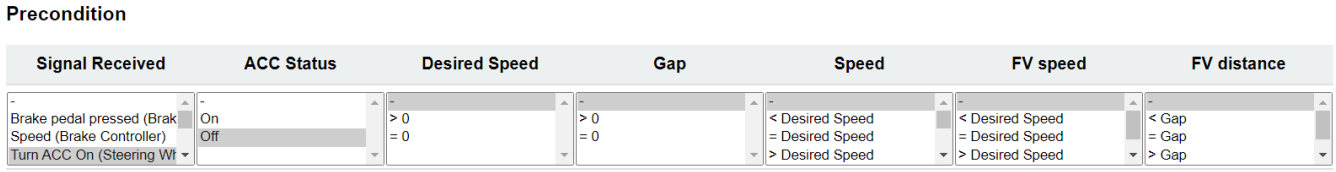

FIGURE 8. Adding a precondition in the elaborate use case specification activity.

An enhancement of the user interface to make the graphics more user-friendly is part of our future works.

VII. RESULTS AND DISCUSSIONS

The output of our method is a model with the state machine diagrams for each block from the block diagram. There are two ways to check if the synthesized model is correct. First, the system engineers check the model analyzing if the method synthesized the block and initial state machine diagrams correctly (sections V-G and V-H). Next, they can perform simulation and formal verification to check if the diagrams' behavior is the same as defined in the STPA and functional requirements. *Perform simulation and formal verification* is the last activity of Souza *et al.* [4] method. We discuss the results of the model generated in Section VI with the tailoring described in Section V-I.

To conduct this activity, we use TTool [27], a toolkit that supports the modeling of SysML diagrams and enables simulation of the model and formal verification using UPPAAL. We chose TTool because it is an open-source environment that supports the AVATAR profile [14]. Moreover, the authors claim that designers that use TTool can elaborate their models even with minor knowledge of UML/SysML and formal languages such as UPPAAL [15]. However, we believe the same results can be achieved by tools with the same features of TTool.

We performed several simulations of ACC using TTool. We considered that each simulation has two parts. The first part encompasses *Driver's* interactions with the vehicle (such as accelerating and braking) and with the *Adaptive cruise controller* (engaging ACC, setting Desired speed and Gap). The second part encompasses ACC controlling the vehicle's speed while *Driver* observes. The first part has the same results in all simulations. However, the second part has different results for each simulation. The differences in the results were due to the elaboration of the state machine diagram for the block *EXT_Forward_Vehicle*.

We considered the speed of the forward vehicle equal to twenty-eight at the beginning of the simulation. In the following sixty units of time, the variable *isIncrementing* assumed the values zero or one, and the variable *randomValue* received the values zero, one, or two. If the value of *isIncrementing* is zero, we decrement the speed according to the content of variable *randomValue*. The same is valid for *isIncrementing* equal to one, but instead of decrementing the speed, we increment it. At each unit of time, the variables *isIncrementing* and *randomValue* receives the return of the TTool's function *RANDOM0(minValue, maxValue)*, which uses a uniform distribution function to generate the values (where the *minValue* is zero and *maxValue* is one for *isIncrementing* and two for *randomValue*).

There are differences in the simulation when analyzing the second part because of the randomness of the *FV Speed*. The differences are in the number of times *Adaptive cruise controller* issues the commands that control the vehicle when there is a forward vehicle (CA01, CA02, UC12, and UC13) - In some scenarios, the *Adaptive cruise controller* will *Accelerate* more than *Brake* and vice-versa. Despite that, we did not detect an unsafe or unexpected behavior in any simulation.

Figure 10 depicts the interactive simulation window of TTool. For ACC, one of the simulations ended after 139 (one hundred thirty-nine) units of time and after 14034 (fourteen thousand thirty-four) transactions. The coverage indicates that, during the simulation, the model visited 97.3% (ninety-seven and three tenths) of all states, considering all state machine diagrams from all blocks. The exceptions were UC03_A and UC03_B (disengaging ACC when Speed is out of the boundaries). The simulation did not achieve these states because the *Adaptive cruise controller* maintained the *Speed* between eight and thirty-three when ACC was *On* (the minimum *FV Speed* is ten and the maximum is thirty, within the ACC's Speed limit). We executed the simulation using the "step-by-step" feature that runs one transaction at a time (defined in the "Nb of steps" input). We can see the simulation running in the state machine diagrams like a sequence diagram in the inferior part of Figure 10.

There is information on the right side of Figure 10 that aids the analysis of the simulation results. The figure shows the variables of *Adaptive cruise controller* at the end of the simulation, where the speed of the vehicle with ACC and the forward vehicle is zero (i.e., the vehicle is stopped and no forward vehicle is detected). By exploring the simulation feature, we also can analyze: (i) The number of transactions of each block in the "Blocks" panel; (ii) Information about each transaction (such as the block, state, time when it begins, duration, and time when it ends) in the "Transactions" panel; (iii) The number of times each state was visited in "Met states" panel.

We can use formal verification into our model using the UPPAAL model checker. TTool performs the formal verification verifying each safety property (or pragma) defined by the systems engineer. Figure 11 depicts some of the safety properties verified for ACC system. We wrote the properties using the "leads to" operator. The meaning is when the state on the

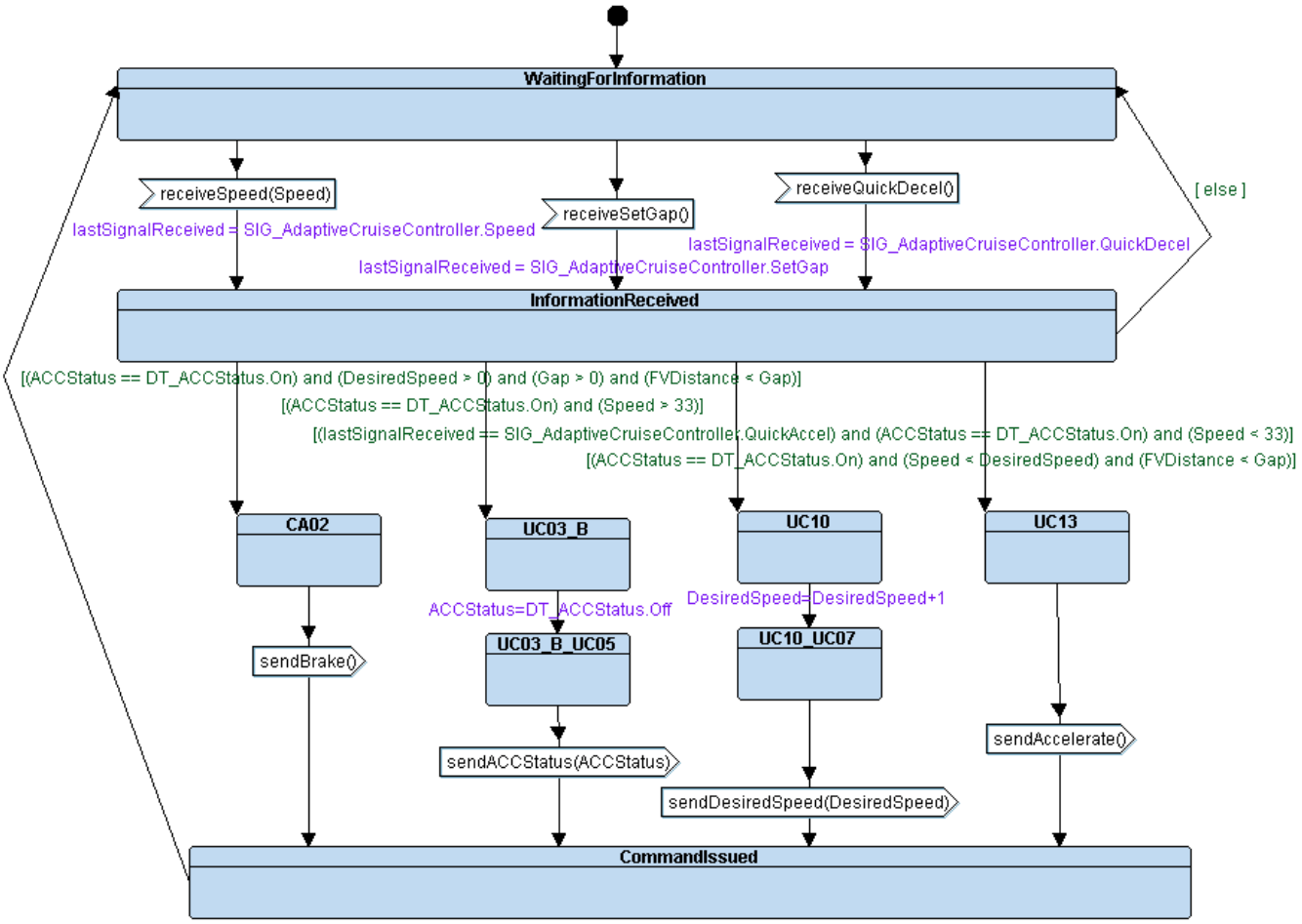

FIGURE 9. Initial state machine diagram for the adaptive cruise controller generated by the method.

left of the arrow (–>) becomes true, the state on the right of the operator will inevitably become true. For example, whenever *Driver* accelerates the vehicle (Driver.Accelerating), the *Wheels* must move (Wheels.Moving). The checkmark before each safety property indicates the safety pragma is satisfied. If a pragma is not satisfied, the tool displays the Crossmark instead. We created the safety pragmas based on STPA step 3 (Identify unsafe control actions) and the use cases. As stated in Figure 11, we were able to check that all functionalities of the ACC were satisfied.

According to Knorreck *et al.* [15], when the models achieve a certain level of detail, they are not amenable to formal verification. Unfortunately, this was the case with our model. To check the safety pragmas using UPPAAL in a tolerable time for us (less than thirty minutes per property) and to reach in Figure 11, we need to reduce the number of iterations of the forward vehicle, *e.g.*, from sixty to five.

Besides ACC, we also applied our method to the Train door system (TDS), a simple but widely discussed example in the STPA literature [1], [25]. The method adequately synthesized the block diagram and state machine diagram for the TDS system, including its four control actions from ''Not providing causes hazard'' and one functional requirement. Moreover, all the ten safety pragmas were satisfied.

We faced two main challenges when conceiving the method. The first challenge was to discover how to synthesize the control algorithm based on the STPA analysis and how to complement it with the functional requirements (dealing with the conflicts/reinforcements and maintaining the safety). The second challenge was to reach the model of the initial state machine diagram for controllers (generic enough to fit any non-human controller).

The main advantage of our method is to check if the automated controllers are behaving safely and as expected. Using our method, the system engineers can achieve it with little effort while they perform their usual safety analysis and elicit the requirements for their systems.

The proposed method has some limitations. The first limitation is not considering the safety recommendations of the ''Identify Loss Scenarios'' step of STPA to synthesize the control algorithm. Considering a safety recommendation as ''sensor replication'', the method should synthesize the block diagram with the replicated component (and, therefore, the state machine diagram for controllers should receive

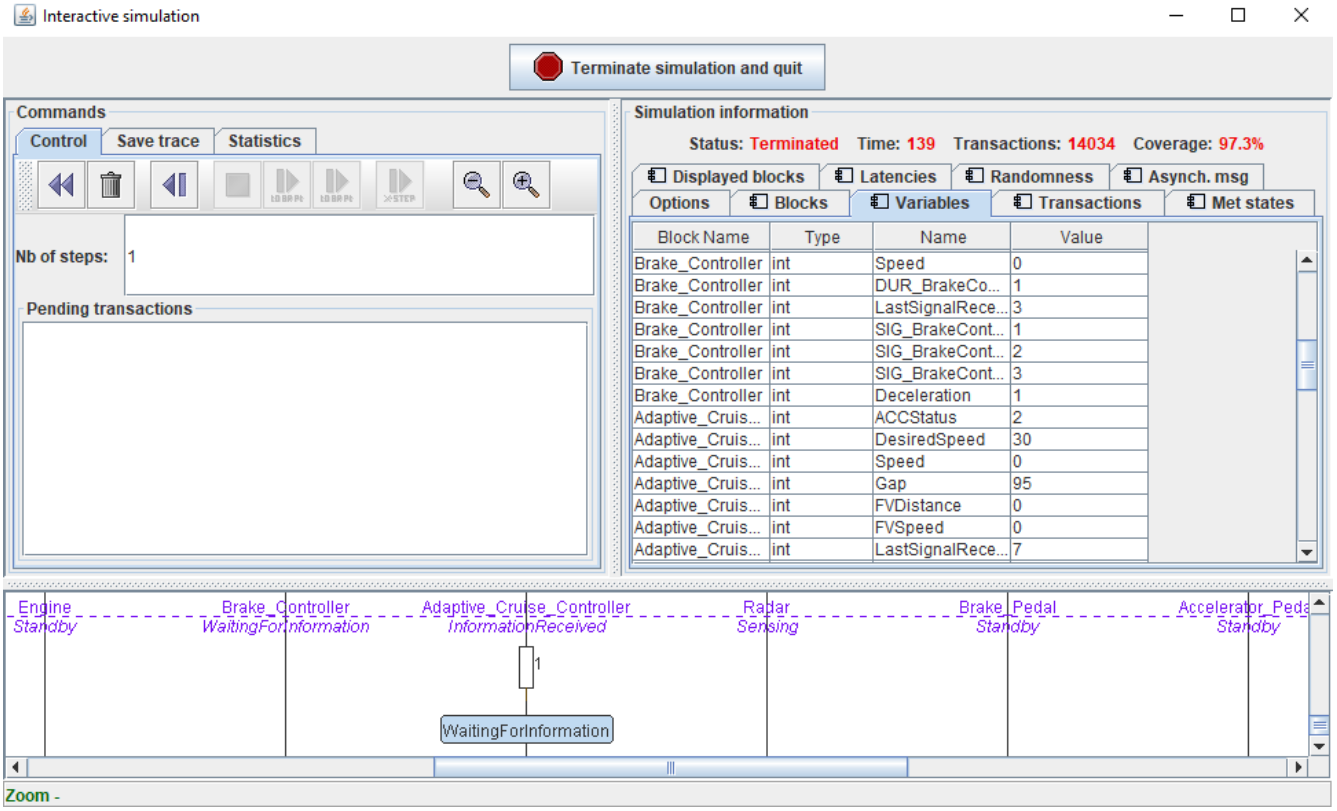

FIGURE 10. Interactive simulation window of TTool after execute the simulation of ACC.

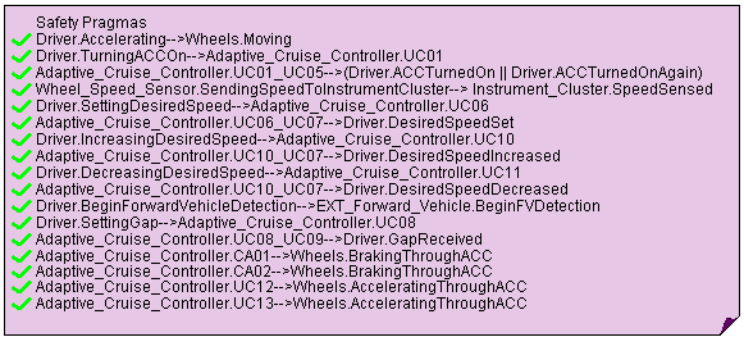

FIGURE 11. Satisfied safety properties for ACC system.

data from both sensors). We did not address this limitation because these recommendations are specific and require *ad hoc* changes in the block diagrams and control algorithm. Since the identification of loss scenarios is a cognitive activity and is currently based on expert knowledge, Artificial Intelligence methods may be useful for identifying loss scenarios in existing systems where operational data can be collected. However, the loss scenarios in existing systems and systems that are subject to our concept design stage activities may or may not be related and finding methods to study the relationship is an open area of research.

The second limitation is that the method considers that all components (mainly automated controllers, sensors, and actuators) are always available (On). Since our focus is the controller's behavior (checking if the controller issues all commands adequately), we did not concern about creating an Off state. Due to this fact, we believe the behavioral diagram synthesized by the method is insufficient for code generation (although we do not intend to produce source code).

Another limitation of the method is the conflict resolution. Conflict resolution is a manual task and depending on the number of unsafe control actions identified (added to the functional requirements), it can be a laborious task. Currently, the method only shows the conflicts, and the system engineers must resolve them. There is no additional aid from the method to solve the conflicts automatically.

The tool prototype has a limitation when defining the preconditions and postconditions. Currently, it only allows

the AND operator, not supporting the OR operator between variables or states of the same variable. Implementing the OR operator in the tool is a trade-off because it adds a feature to the system; however, it increases the complexity, making the tool harder to use. Moreover, the current way to define the precondition is more laborious than defining them textually using natural language. However, using natural language is more difficult to generate the guard condition (due to the author's writing style) and prone to error.

Another limitation is related to the use of TTool. We considered including the stop-and-go function in ACC. We intended to use the mathematical expressions defined by Kesting *et al.* [29] to calculate the vehicle's acceleration. However, TTool does not support floating-point numbers (required to calculate the acceleration). Despite TTool's limitation, we decided to keep using it due to its features.

The synthesizer generates the initial state machine diagram for controllers, actuators, and sensors. It does not synthesize the state machine for the controlled process and external components. However, we do not consider these restrictions as limitations of the method because there is no such information in the STPA analysis or requirements diagram (it is out of the scope of the analysis).

In the ACC logic synthesis, we did not consider performance requirements. Performance requirements are critical to some safety-critical systems such as flight control systems. In these systems, the controller has deadlines to process all the inputs and issue commands. To address performance requirements for controllers, time-out events can be easily added in the state machine diagrams. A time-out event is scheduled when a certain condition holds and the occurrence of a time-out event may trigger commands to other components in the state machine diagrams. If the processing unit of the controller is not reliably capable to process all the events and generate the commands in due time, more processing units may be required. The addition may affect the design of the blocks and their state machine diagrams.

VIII. CONCLUDING REMARKS

We proposed a method to synthesize the control algorithm for automated controllers (state machine diagram) that meets the safety and functional requirements. We provide an automated method to the *Perform Design* activity that combines the STPA analysis and SysML modeling [4]. The proposed method is iterative and may be used to evaluate the automated controllers, detect conflicts and redundancies between commands (control actions and functional requirements), check if the control algorithm fulfills both safety and functional requirements, and help to evaluate the impact of the changes of the STPA analysis. We evaluated our method using the Adaptive Cruise Control as a running example. By performing model simulation and formal verification to assess the Adaptive cruise controller's behavior, we checked that the method synthesized a controller that satisfied the safety properties and followed the functional and safety requirements.

Based on the current limitations of the method, we recommend the improvement of the initial state machine diagrams (approaching them to components' original state) and the inclusion of the information from the "Identify loss scenarios" step in the block diagram. Other future works include exploring other functions of TTool to improve simulation results, such as defining a channel between blocks as lossy.

A systematic method for conflict resolution, identifying the redundant commands automatically, and grouping them conveniently to perform the analyses was not the goal of this work, but is needed for the whole method to be useful in practice. Moreover, we suggest improving the method to consider the security requirements together with safety and functional requirements.

It is not easy to claim cost and time reductions due to using this method without several controlled experiments in realistic industrial environments. Nonetheless, we believe that the systematic generation of the control algorithm considering the results of the first three steps of STPA and the functional requirements gives a useful aid in the development of complex systems we are building today.

REFERENCES

[1] N. G. Leveson, *Engineering a Safer World: Systems Thinking Applied to Safety*, 1st ed. Cambridge, MA, USA: MIT Press, 2011. Accessed: May 7, 2022. [Online]. Available: http://sunnyday.mit.edu/safer-world.pdf

[2] N. G. Leveson and J. P. Thomas. (Mar. 2018). *STPA Handbook*. Accessed: May 7, 2022. [Online]. Available: https://psas.scripts.mit.edu/home/get_file.php?name=STPA_handbook.pdf

[3] S. M. Sulaman, A. Beer, M. Felderer, and M. Höst, "Comparison of the FMEA and STPA safety analysis methods—A case study," *Softw. Quality J.*, vol. 27, no. 1, pp. 349–387, Mar. 2019, doi: 10.1007/s11219-017-9396-0.

[4] F. G. R. de Souza, J. de Melo Bezerra, C. M. Hirata, P. de Saqui-Sannes, and L. Apvrille, "Combining STPA with SysML modeling," in *Proc. IEEE Int. Syst. Conf. (SysCon)*, Aug. 2020, pp. 1–8, doi: 10.1109/SysCon47679.2020.9275867.

[5] D. Zhong, R. Sun, H. Gong, and T. Wang, "System-theoretic process analysis based on SysML/MARTE and NuSMV," *Appl. Sci.*, vol. 12, no. 3, p. 1671, Feb. 2022, doi: 10.3390/app12031671.

[6] C. Zhao, L. Dong, H. Li, and P. Wang, "Safety assessment of the reconfigurable integrated modular avionics based on STPA," *Int. J. Aerosp. Eng.*, vol. 2021, pp. 1–14, Jan. 2021, doi: 10.1155/2021/8875872.

[7] D. Dghaym, T. S. Hoang, S. R. Turnock, M. Butler, J. Downes, and B. Pritchard, "An STPA-based formal composition framework for trustworthy autonomous maritime systems," *Saf. Sci.*, vol. 136, Apr. 2021, Art. no. 105139, doi: 10.1016/j.ssci.2020.105139.

[8] *OMG Systems Modeling Language (OMG SysML)*, document Version 1.6, Dec. 2019.

[9] *OMG Unified Modeling Language (OMG UML), Superstructure*, document Version 2.4.1, Jul. 2011.

[10] P. Roques, "MBSE with the Arcadia method and the capella tool," in *Proc. 8th Eur. Congr. Embedded Real Time Softw. Syst. (ERTS)*, Jan. 2016, pp. 2–11.

[11] *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems)*, document Version 1.2, Apr. 2019.

[12] P. H. Feiler and D. P. Gluch, *Model-Based Engineering With AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st ed. Reading, MA, USA: Addison-Wesley, 2012.

[13] J. B. Dabney and T. L. Harman, *Mastering Simulink*, 1st ed. Upper Saddle River, NJ, USA: Prentice-Hall, 2004.

[14] G. Pedroza, L. Apvrille, and D. Knorreck, "AVATAR: A SysML environment for the formal verification of safety and security properties," in *Proc. 11th Annu. Int. Conf. New Technol. Distrib. Syst.*, May 2011, pp. 1–10, doi: 10.1109/NOTERE.2011.5957992.

[15] D. Knorreck, L. Apvrille, and P. de Saqui-Sannes, "TEPE: A SysML language for time-constrained property modeling and formal verification," *ACM SIGSOFT Softw. Eng. Notes*, vol. 36, no. 1, pp. 1–8, Jan. 2011, doi: 10.1145/1921532.1921556.

[16] G. Howard, M. Butler, J. Colley, and V. Sassone, "A methodology for assuring the safety and security of critical infrastructure based on STPA and Event-B," *Int. J. Crit. Comput.-Based Syst.*, vol. 9, pp. 56–75, Mar. 2019. Accessed: May 5, 2022, doi: 10.1504/IJCCBS.2019.098815.

[17] A. L. Dakwat and E. Villani, "System safety assessment based on STPA and model checking," *Saf. Sci.*, vol. 109, pp. 130–143, Nov. 2018, doi: 10.1016/j.ssci.2018.05.009.

[18] H. Wang, D. Zhong, T. Zhao, and F. Ren, "Integrating model checking with SysML in complex system safety analysis," *IEEE Access*, vol. 7, pp. 16561–16571, 2019, doi: 10.1109/ACCESS.2019.2892745.

[19] R. Krishnan and S. V. Bhada, "An integrated system design and safety framework for model-based safety analysis," *IEEE Access*, vol. 8, pp. 146483–146497, 2020, doi: 10.1109/ACCESS.2020.3015151.

[20] Y. Ding, W. Li, D. Zhong, H. Huang, Y. Zhao, and Z. Xu, "System states transition safety analysis method based on FSM and NuSMV," in *Proc. 2nd Int. Conf. Manage. Eng., Softw. Eng. Service Sci. (ICMSS)*, 2018, pp. 107–112, doi: 10.1145/3180374.3181346.

[21] J. F. Pétin, D. Evrot, G. Morel, and P. Lamy, "Combining SysML and formal methods for safety requirements verification," in *Proc. 22nd Int. Conf. Softw. Syst. Eng. Their Appl.*, Nov. 2010, pp. 1–11.

[22] A. Abdulkhaleq, S. Wagner, and N. Leveson, "A comprehensive safety engineering approach for software-intensive systems based on STPA," *Proc. Eng.*, vol. 128, pp. 2–11, Jan. 2015, doi: 10.1016/j.proeng.2015.11.498.

[23] *Intelligent Transport Systems Adaptive Cruise Control Systems Performance Requirements and Test Procedures*, document ISO 15622:2018, Sep. 2019.

[24] S. Moon, I. Moon, and K. Yi, "Design, tuning, and evaluation of a full-range adaptive cruise control system with collision avoidance," *Control Eng. Pract.*, vol. 17, no. 4, pp. 442–455, 2009, doi: 10.1016/j.conengprac.2008.09.006.

[25] N. G. Leveson and J. P. Thomas, *An STPA Primer*. Cambridge, MA, USA: Massachusetts Institute of Technology, 2015.

[26] D. P. Pereira, C. M. Hirata, R. M. Pagliares, and S. Nadjm-Tehrani, "Towards combined safety and security constraints analysis," in *Proc. Int. Conf. Comput. Saf., Rel., Secur. (SAFECOMP)*, 2017, pp. 70–80, doi: 10.1007/978-3-319-66284-8_7.

[27] L. Apvrille, "TTool for DIPLODOCUS: An environment for design space exploration," in *Proc. 8th Int. Conf. New Technol. Distrib. Syst. (NOTERE)*, 2008, pp. 1–3, doi: 10.1145/1416729.1416764.

[28] F. G. R. de Souza, D. P. Pereira, R. M. Pagliares, and C. M. Hirata, "WebSTAMP: A web application for STPA & STPA-sec," in *Proc. Int. Cross-Ind. Saf. Conf. (ICSC) Eur. STAMP Workshop Conf. (ESWC) (ICSC-ESWC)*, Feb. 2019, Art. no. 02010, doi: 10.1051/matecconf/201927302010.

[29] A. Kesting, M. Treiber, M. Schönhof, and D. Helbing, "Adaptive cruise control design for active congestion avoidance," *Transp. Res. C, Emerg. Technol.*, vol. 16, no. 6, pp. 668–683, Dec. 2008, doi: 10.1016/j.trc.2007.12.004.

FELLIPE GUILHERME REY DE SOUZA received the B.Sc. degree in computer science from the Universidade Federal de Alfenas (UNIFAL), and the M.Sc. degree in electronics and computer engineering from the Instituto Tecnológico de Aeronáutica (ITA), where he is currently pursuing the Ph.D. degree in electronics and computer engineering.

CELSO MASSAKI HIRATA received the B.S. and M.S. degrees in mechanical-aeronautical engineering and operations research from the Instituto Tecnológico de Aeronáutica (ITA), in 1982 and 1987, respectively, and the Ph.D. degree in computer science from the Imperial College London, in 1995.

From 1986 to 2014, he was an Assistant Professor with the Department of Computer Science, ITA. Since 2014, he has been a Full Professor with the Department of Computer Science, ITA. His research interests include distributed systems, cyber-physical systems, machine learning, and big data.

Dr. Hirata was a recipient of the ACM SAC Best Symposium Paper Award, in 2011.

SIMIN NADJM-TEHRANI received the B.Sc. degree from Manchester University, U.K., and the Ph.D. degree in computer science from Linköping University, Sweden, in 1994.

During 2006 to 2008, she was a Full Professor at the University of Luxembourg. She is currently a Professor of dependable distributed systems at the Department of Computer and Information Science, Linköping University, where she has been led the Real-Time Systems Laboratory, since 2000. Since 2015, she has been leads the National Research Centre on Resilient Information and Control Systems (RICS) financed by Swedish Civil Contingencies Agency which focuses on security for industrial control systems. Her research interests include networks and systems with dependability requirements and resource constraints, with applications in aerospace, edge networking, and intelligent systems.