Incremental Contract-based Verification of Software Updates for Safety-Critical Cyber-Physical Systems

Yosab Bebawy*, Houssem Guissouma†, Sebastian Vander Maelen*, Janis Kröger‡, Georg Hake‡, Ingo Stierand*, Martin Fränzle‡, Eric Sax†, Axel Hahn‡
*R&D Transportation Division, OFFIS e.V., Oldenburg, Germany
†Institute for Information Processing Technologies, Karlsruhe Institute of Technology, Karlsruhe, Germany
‡Department of Computing Science, University of Oldenburg, Oldenburg, Germany
{yosab.bebawy, sebastian.vander.maelen, ingo.stierand}@offis.de, {houssem.guissouma, eric.sax}@kit.edu
{janis.kroeger, georg.hake, martin.fraenzle, axel.hahn}@uni-oldenburg.de

***Abstract*—Software updates are indispensable for the continuous development of Cyber Physical Systems (CPS): They allow for low-cost bug-fixing, fast adaptation to new or changing environments, or adding new functionality throughout the CPS's life-cycle. Due to the urgent need for some safety-critical updates, their verification and validation may need to happen as fast as possible without loss of quality. For this reason, incremental checks targeting specifically the introduced changes and their impact on the system are essential as they speed up the validation process. In this paper, we introduce a concept for such an incremental verification for different types of updates by using contract-based design and verifying the integration of the introduced changes by checking their compliance with the contractually agreed assumptions and guarantees. We demonstrate our approach by applying two update types to an Adaptive Cruise Control (ACC) system and verifying the impact of the changes within the environment of the changed module(s).**

***Index Terms*—Contract-based Design, Incremental Verification, Change Impact Analysis, Software Updates**

I. Introduction

The development of software modules in safety-critical Cyber-Physical Systems (CPS) is facing numerous challenges due to their complexity and deep integration of computing and communication parts. An example of the complexity of today's systems can be found in the automotive sector. Today's cars can either include a high number of ECUs [1] interacting with each other constantly or utilize common resources for different kinds of functionality [2]. Communication is either realized on-chip, i.e. via multiprocessor system-on-chip, or via physical communication buses. In order to handle the complexity of designing such systems, contract-based design has been introduced as an efficient formal verification method for CPS [3]. Contracts provide the developers of a system configuration with formal specifications for each module and their composition into a system. For this purpose, the requirements for each module and for the system as a whole are described by a set of assumptions about the respective environment of the module and by guarantees which have to be fulfilled by the module under the defined assumptions. Due to the safety-critical nature of most CPS, thorough verification and validation activities are an essential part of their development and maintenance processes. Unfortunately, when maintaining safety-relevant software modules during operation, regular software updates become necessary to fix errors, optimize processes or extend functionality. Moreover, updates can help to extend the lifespan of a system, making it more cost-efficient and adapting it to changing environments. In the automotive field for example, the frequency of software updates over the air is expected to rise significantly [4]. Hence, the test effort should be optimized towards more agility without negatively influencing the quality of the software.

With contract-based design as formal verification method specifying the correct interaction between modules and enabling continuous verification throughout the development process, we introduce an approach that utilizes contracts to verify the incremental changes applied by an update. As unit of incremental change, we use the notion of deltas, from which we derive an impact analysis process considering different update types (perfective, corrective, adaptive). We validate our approach with a case study from the automotive domain.

This work has been funded by the German Federal Ministry of Education and Research (BMBF) in the project Step-Up!CPS (Förderkennzeichen: 01IS18080-).

II. State-of-the-Art

A. Contract-Based Design

The term contract and contract-based design was used in many publications in the past [5] [6]. For further information, we refer to the *Contracts for System Design* book [7]. The general idea is to assign contracts to components which are designed to operate in a specified environment and generate a specified output. Assuming that the component is working in the specified environment, it is guaranteed that it fulfills the specified output. This leads to the notion of contract as a pair $\mathcal{C} = (A, G)$, where A are the assumptions regarding the component's environment (i.e. inputs) while G are the guarantees that are related to the output of the component.

In this work, we consider three categories of contracts assigned to different viewpoints of the system: functional, timing and safety contracts. We focus on contract-based Virtual Integration Testing (VIT), where contracts are used to check refinement steps between system models in a model-based design approach [3]. This allows the system to be virtually

978-1-7281-7624-6/20/$31.00 ©2020 IEEE

DOI 10.1109/CSCI51800.2020.00318

integrated with its subsystems equipped with multi-viewpoint contracts [3].

A system has the possibility to be in different operational modes. Hence, a system may have different functionality or timing behavior in each mode. To model this behavior by using contract-based design, we need mode-dependent contracts, which have different assumptions or guarantees depending on the selected mode. Hence, a regular contract $\mathcal{C}$ is not sufficient and we need to use the concept of extended contracts inspired from [8], [9]. In addition to the normal assumptions and guarantee pairs (A, G), we extend $\mathcal{C}$ with one or more conditional assumption guarantee pairs (B, H). In the following, we call conditional contract pairs as mode pairs. Unlike the normal pairs which must always be satisfied, this is not mandatory for the mode pairs. However, there is a condition that if an assumption from a mode contract holds, the corresponding guarantee must also hold. Formally, the extended contract $\mathcal{C}'$ results as following:

$$\mathcal{C}' = (A, G, \{(B_1, H_1), \ldots, (B_n, H_n)\}) \tag{1}$$

With regard to the modes of a system, the concept of extended contracts allows the normal assumption-guarantee pairs to express the pre- and post-conditions that must hold for all modes and the conditional pairs to express mode-dependent functionalities or timing specifications. This gives us a better structure of contract for modes and allows us to contractify the allowed transitions within a specific mode. Adding or updating a mode should also made be easier.

B. Update Types

In general, software and hardware updates or upgrades can be differentiated. However, we are focusing only on software updates in this work. We define an update U of one CPS as a change of its embedded software through modifying one or more of its software modules. Based on the change categorization proposed in [10], we differentiate three update types. These are the corrective update for fixing bugs, the perfective one for improving the functionality according to different criteria such as performance, and the adaptive update.

TABLE I
UPDATE TYPES: CORRECTIVE, PERFECTIVE, ADAPTIVE

Update type	Corrective	Perfective	Adaptive
Trigger	design inconsistency	safety issue, comfort	new features, environment
Implementation Change	yes	yes	yes
Contract Change	no	only guarantees	yes
Example	wrong output value	improving execution time	adding object detection

The latter type adds new functionality to the system to adapt to changing working conditions or to new customer demands. Depending on the update type, changes on different levels need to be introduced to the Module Under Update (MUU), which can influence the rest of the system architecture. This may require additional changes to other modules directly or indirectly related to the MUU. Table I summarizes the difference between the three update types.

III. Related Work

In the area of incremental verification, Johnson et al. and Bu et al. presented processes that allow the re-verification of module-based software systems after a change, such as additions, modifications or the removal of a module [11], [12]. Rothenberg, Dietsch and Heizmann use trace abstraction, an approach built on automata, in their work [13]. Moreover, Juhasz presents a verification approach that uses local theorem evidence for each point in the program instead of for the entire program and shares the results of the local results on request [14]. Cheng and Tisi present an approach based on a modeling language that divides contracts into sub-targets and provides verification results for the sub-problems [15].

Focusing more on the formal verification methodology, the authors of [16] introduce practical requirements when defining the scope of formal verification. In [17], Heitmeyer et al. utilize a Software Cost Reduction (SCR) tabular method to verifiy the system specifications.

Finally, Change Impact Analysis (CIA) is an activity that aims at reducing that effort by identifying the necessary re-verification effort. Oertel et al. present in [18] an approach which guarantees to keep system integrity while performing changes. The authors of [19] present a case study which analyses how much time is spent on CIA. Ultimately, in [20], a layered framework is introduced, that allows tracking the impact of changes at all levels of abstraction.

Our approach differs from the mentioned works in that we i) map three different dimensions using multi-viewpoint contracts (functional, time and safety) within the model-based design, ii) virtually test their integrity, and iii) do this incrementally for each type of update.

IV. Approach for safe updates

A. Model-based Design and Contracts Elicitation

Any system under design must be defined via a set of requirements. All defined requirements are transferred into system contracts that will be refined during the next phases of the design process. The function architecture definition takes place by a functional decomposition of the functions at subsequent granularity levels. All (sub)functions are annotated with (sub)contracts based on the top-level requirements and top-level contracts. In order to check in this phase whether the refinement and correctness of the composition of the functions holds, we perform a VIT using the tools MULTIC [21] for timing and OCRA [22] for functional aspects.

Based on the function architecture, the safety assessment of the system takes place. Therefore, the first step is to perform a Hazard and Risk Analysis (HARA) in order to identify all possible hazards resulting from possible failures. The second step is to perform a risk assessment for the identified hazards specifying the likelihood or frequency of a hazard occurring, as well as the severity of its consequences. The third step is

to identify the safety requirements that are needed to avoid or to mitigate the identified hazards. In the end of the design process, safety requirements must be satisfied by the technical realization of the system. They hence put constraints on how to decompose the function architecture, and on possible partitioning schemes in the logical and software design phase.

The function architecture then is extended by integrating safety mechanisms in the logical decomposition step based on the findings from the safety analysis. The inclusion of a safety mechanism to the function architecture leads to the addition of components and/or to extensions in the component functionality. These changes must also be reflected by the contracts and the resulting system has to be re-verified.

The logical decomposition also includes the mapping of the individual functions to *modules*. Our module definition is similar to the definition of a software component (SWC) in AUTOSAR [23], where a SWC exclusively uses interfaces that are provided by the operating system. This makes the SWC ready for deployment on different platforms. We use the notion of a module to identify the boundaries of an updatable component. The module combined with contracts can help us to reduce the verification and validation efforts when updating a component.

B. Incremental Verification for different Update Types

If a system is put together from individual components, care must be taken to ensure that the individual parts are compatible and there are no conflicts in their interaction as a whole. Therefore, attention must be paid to the influence that the replacement of a module has on the interaction of the overall system. This includes the timing of each component (timing failure), functional dependencies (e.g. deadlocks) or the load and sequence of memory accesses. In addition, there are indirect interactions between the modules that are not obvious w.r.t energy consumption or heat dissipation. For this reason, safety standards, such as IEC 61508, require that the individual modules are sufficiently independent of each other so that a change due to an update does not have a negative effect on safety-relevant functions of the overall system [24], [25]. Therefore, our approach is based on two major concepts. On the one hand, the concept of modularizing the individual components, which can be combined into a complete system according to a set of integration rules. On the other hand, we annotate the modules with contracts and thus formalize the composition so that the change introduced by an update can be controlled formally and tracked. The resulting modular system design with attached contracts provides properties such as high cohesion, low coupling, well-defined interfaces and information hiding [26], which we can use to assess the impact of each type of update (cf. section II-B). Assuming a contract-based system description with well-defined boundaries, the introduced change for an update can be one or a combination of the following delta types:

Interface change $\boldsymbol{\Delta I}$: An interface (input or output port) is modified, deleted or added to the component. This can be necessary to extend the functionality of the MUU for feeding in additional information containing sensor or actuator data or communication messages from other modules. This change is typically made for adaptive updates.

Contract change $\boldsymbol{\Delta \mathcal{C}}$: Changing one or more contracts, deleting, or adding new ones. This change is typically made for adaptive and perfective updates.

Implementation change $\boldsymbol{\Delta Impl}$: The implementation in the form of a behavioral model or a structure of sub-components is adjusted. This change is typically made for corrective updates, i.e. bug-fixes.

Figure 1 represents the three described delta types for an exemplary component with one input, one output, and a list of three contracts $\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$.

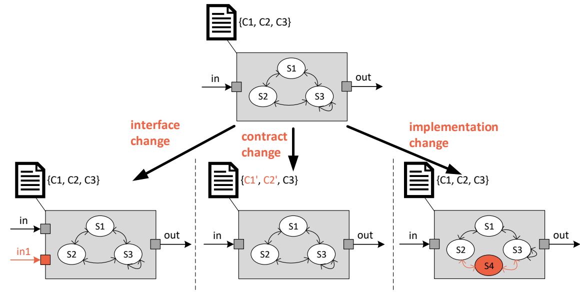

Fig. 1. *Delta* types in a contract-based design approach for CPS updates

We define an update U as the sum of deltas $\boldsymbol{\Delta}$ at a certain abstraction level of the k affected components. Each $\boldsymbol{\Delta}$ has three dimensions according to the three types of change described above (cf. Fig. 1). This is expressed by the formula in Eq. 2. The motivation of using deltas as a unit of change is their ability to represent spatial and temporal variability at the same time such as used for software product lines [27]. In addition, the modularity aspect of contract-based design facilitates the traceability and quantification of changes.

$$U = \sum_{k} \boldsymbol{\Delta} = \sum_{k} \begin{pmatrix} \Delta I \\ \Delta \mathcal{C} \\ \Delta Impl \end{pmatrix} \tag{2}$$

In a first step, the introduced delta $\boldsymbol{\Delta}$ for the MUU is used to determine the impact on dependent components at various granularity levels within the system hierarchy. Since, in the worst case, all sub-components of an overall system can be directly or indirectly dependent on each other, the dependency on the affected components is first identified in order to reduce the search space. Since an update can change the number of affected modules, the search space can only be identified after the final description of the update. Then, it is possible to carry out the impact analysis by following the process steps in Figure 2.

An update is usually triggered by a request from customers, stakeholders or developers to either fix, optimize, extend or delete a specific system functionality. The update team must then check whether the system is designed to be updated and equipped with the required infrastructure that allows the

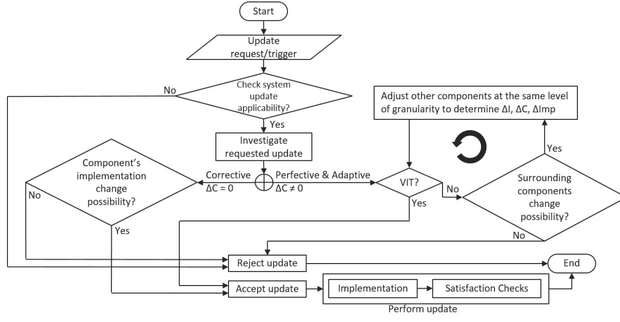

Fig. 2. Impact analysis and incremental verification for software updates

system to receive an update (e.g. update interface, over-the-air-update module) or not. In case of an updatable system, the update team must investigate the description of the requested update, so that they can identify the set of changes (e.g interface, contract, implementation) that are needed to update a certain component within the system. Based on the update type, the impact analysis process is branched into two tracks. The corrective update track is pursued without contracts change, where the team must check whether there is a possibility to change the implementation of the component or not. If yes, the update is accepted and the process of updating the component starts in the "perform update" process, else the update is rejected and the impact analysis is ended. On the other hand, the perfective/adaptive update track starts with performing a VIT in which the compatibility and consistency of the overall system can be determined on a formal level, without implementation and solely on the basis of the contracts. The value ranges of the respective contracts (in our case only the delta to the previous state) are run through and influences on dependent modules within the system are measured. Based on the measured influences, the affected components must be updated too and their set of changes, i.e. deltas, must be identified and adjusted so that the dependent components become compatible and consistent again. Those steps are iterated starting from the component level up to the system level or vice versa if necessary. However, the iteration can be interrupted at any granularity level in case that (i) there is a dependent component that can not be updated due to unreachability or copyrights reasons (ii) a VIT is successfully terminated when the composition of a set of components contracts within the same granularity level refine their top-level contract.

Since different levels of abstractions are usually used in a top-down refinement process, we focus on the level of the smallest updatable components, which we define as MUUs. The conducted incremental verification checks are mainly dependent on the type of the update; this is explained below.

1) Violated contracts as input for corrective update: We assume that a corrective update of one component M introduces one or more deltas with changes only in the implementation of the corresponding contract(s). This means that the interfaces and contracts of the MUU stay the same. In this work, we introduce the implementation changes based on violated contracts only. Those contract violations are part of the feedback data within the update life-cycle model. For N affected contracts $\{\mathcal{C}_1, \cdot, \mathcal{C}_N\}$, only those contracts are checked for the updated module using unit testing techniques. Depending on the degree of modularity of the used contract-based design environment, regression tests may be required to make sure that the introduced changes didn't lead to other unforeseen errors. For this purpose, state of the art regression testing techniques, such as described in [28], can be used. Also, and to make sure that no inconsistencies happened between the stages of unit testing and components integration, incremental satisfaction checks using simulation or a digital-twin representation are conducted as further validation step.

2) Incremental virtual integration check: Static and dynamic verification techniques are used for the realization of VIT whenever there is a change of contracts $\Delta\mathcal{C} \neq 0$. Static checks use formal verification techniques based on a static representation of the system, however dynamic ones rather use a simulation to monitor the corresponding contracts at run-time. We assume an update of a component M changing or adding a contract $\mathcal{C}'$. The composibility of M based on $\mathcal{C}'$ within the system is verified incrementally at design time by checking whether the changes in the composition of the component with its environment (horizontal contracts) as well as the its refinement relations to the higher components (vertical contracts) hold or not.

V. Case Study

A. Use Case Description

We consider an Adaptive Cruise Control (ACC) with a collision avoidance extension as described in [29]. The ACC is designed to provide two operational modes regarding its functionality *Cruise* and *Follow*. In the *Cruise* mode, a constant velocity set by the driver is maintained by the car. This is the case when there is no slower lead car in front of the ego vehicle. If this condition does not hold, the system switches to the *Follow* mode and controls the distance in order to match a desired safe distance. To ensure that the system always keeps the safe distance, we extend our mode concept by an additional *Safety Critical* mode. In this mode, the system performs an emergency brake when the critical distance is violated in a way that makes an emergency action necessary. For the reaction time, we assume that the whole ACC system should not need more than 400 ms to calculate the inputs for the powertrain actuators after receiving inputs from the sensors.

B. System Architecture

1) Software components: The ACC system is composed of a set of interconnected components and sub-components each with a dedicated functionality (see Figure 3). This modular system architecture facilitates the realization of the intended use case description introduced in section V-A. It includes the following components on the system level:

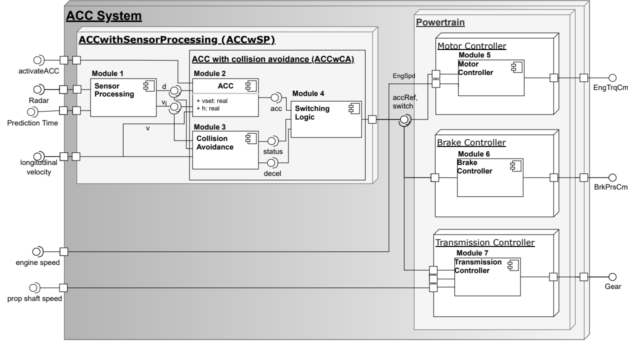

Fig. 3. Component diagram of the studied ACC system

ACCwSP (ACC with Sensor Processing): This is the main subsystem which contains the *Sensor Processing* (Module 1) and the *ACCwCA* (ACC with Collision Avoidance) components. *Sensor Processing* receives the radar detection points and their corresponding time stamps. Then, it processes and analyzes them so that they become ready to be read by the following modules. The **ACCwCA** component is realized by the composition of *ACC* (Module 2), which calculates the acceleration control value ($accRef$) based on the sensor inputs and the parameters h and $vset$, *Collision Avoidance* (Module 3), which outputs a status (on/off) and a deceleration value for the maximum applicable deceleration by full brake, and a *Switching Logic* (Module 4) component. *Switching Logic* implements a simple multiplexing by forwarding the *ACC* output when the system is in the *Cruise* or *Follow* mode, and the *Collision Avoidance* output when the system is in the *Safety Critical* mode.

Powertrain: a simplified model of the powertrain control of a vehicle including a *Motor Controller* (Module 5) calculating the torque command for the engine based on the value of the switch and accRef, a *Brake Controller* (Module 6) providing the braking system with a brake command and a *Transmission Controller* (Module 7), which provides the gear shifting system with a gear command.

For the system specification, we defined a list of requirements based on existing standards such as ISO 15622 and other published works on ACC systems and associated them to the categories *Timing*, *Functional*, and *Safety*.

2) Functional Requirements: We transformed the list of the functional requirements of the component *ACCwCA*, which we consider in the following as the top-level system of interest for the contract analysis, into four guarantees $G_1^T, \cdots, G_4^T$, while the environmental requirements are transformed into assumptions $A_1^T, \cdots, A_4^T$. Based on the decomposition of *ACCwCA* (cf. Figure 3) and the contracts at system level, we defined further contracts at the interfaces of the subcomponents in the same way as described in [30].

3) Timing Requirements: All components above described of the ACC system model require timing analysis. Hence, the individual components are annotated with timing contracts, i.e. specifications, which serves as a guiding strategy when designing and realizing them. In Figure 4, we show the timing scheme for the ACC components, following our main requirement from section V-A that the ACC system shall not induce more than 400 ms delay between receiving the inputs and generating the outputs. Refinement, consistency and compatibility checks for the contracts were successfully verified at this step by performing a VIT in MULTIC that uses a simulation-based verification method for contract-based timing checks.

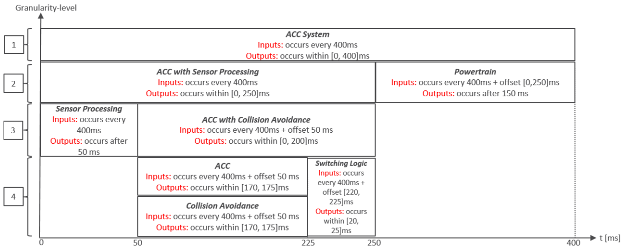

Fig. 4. Timing Requirements of the ACC Components

C. Perfective Update Scenario

A perfective update U_{perf}, as introduced in section II-B, is used to optimize the execution time of the ACC system and consequently improve the quality of the system response. Since we assume a change in the execution time of the system, our focus will lie on the analysis of the timing contracts and the components that realize them.

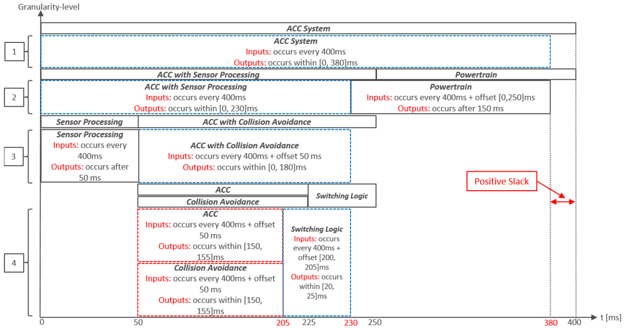

Fig. 5. ACC System Timing Diagram after the Perfective Update

1) Possible Update Reasons: A perfective update is usually performed when it is necessary to optimize the system timing so that a positive slack can be introduced. A positive slack can be used either to introduce a new system functionality within the slack or to perform an early execution of the subsequent components. For the ACC system, we assume that we want to perform a perfective update with which we can save 20 ms from the total system execution time. That means, we can derive the corresponding change in the system timing contract, i.e. a ΔC that leads to ΔImp, with which the execution time is reduced from 400 ms to 380 ms. Therefore, the impact of the change on the whole system and its components must be investigated in a top-down approach throughout the different granularity levels.

TABLE II
IMPACT ANALYSIS RESULTS AT DIFFERENT GRANULARITY LEVELS

			Δ Changes for component due to Perfective Update		
		Component	ΔI	ΔC	ΔImp
Granularity Level	1	ACC System	0	1	1
	2	ACCwSP	0	1	1
		Power Train	0	0	0
	3	ACCwCA	0	1	1
		Sensor Processing	0	0	0
	4	ACC	0	1	1
		Collision Avoidance	0	1	1
		Switching Logic	0	1	0

2) Impact Analysis and Incremental Verification: Starting from the request to perform the perfective update U_{perf} and the assumption that the update is applicable to the system, we have investigated the update prerequisites, i.e. reducing the execution time for the system by 20 ms. Accordingly, we have determined that the ACC system's timing contract must be changed as well as the implementation of the system component, i.e. $\Delta\mathcal{C}$ and ΔImp at granularity level 1, as shown in Figure 5 and Table II. Since $\Delta\mathcal{C} \neq 0$, we identified the need for performing a VIT between two consecutive granularity levels within the ACC system. Such a check will help us to reflect the top-level component's changes to its subcomponents as well as to determine the set of changes that the subcomponents may need in order to make the system consistent again. It is performed as follows:
VIT checks at granularity levels 2 and 1: At this step, the VIT checks whether the composition of *ACCwSP* and *Powertrain* timing contracts at granularity level 2 in Figure 4 are still refining the new timing contract of the ACC system at granularity level 1 in Figure 5. Obviously, the VIT reports a violation due to the inconsistency between the subcontracts and the top-level one. Therefore, there is a need to reflect the changes to the subcomponents and determine the deltas for them. We assume that the *Powertrain* component can not be updated due to unreachability or copyrights reasons. Hence, we can keep the changes for the ACCwSP component and determine the need to change its contract $\Delta\mathcal{C}$ as well as its implementation ΔImp (see Figure 5).
VIT checks at granularity levels 3 and 2: Following the same procedure, the VIT checks the refinement between *Sensor Processing*, *ACCwCA* and *ACCwSP*. It reports a violation due to inconsistencies between components. At this level, we assume that the *Sensor Processing* component can not be updated due to unreachability or copyrights reasons. Hence, we can keep the changes for *ACCwCA* and determine its needed changes: $\Delta\mathcal{C}$ and ΔImp (see Figure 5).
VIT checks at granularity levels 4 and 3: Similarly, the VIT reports a violation at this granularity. Hence, the changes of *ACCwCA* must be reflected by its subcomponents, namely the *ACC*, the *Collision Avoidance* and the *Switching Logic* components. We assume that the implementation of *Switching Logic* stays the same. Hence, the changes of the *ACCwCA* component will be reflected by *ACC* and *Collision Avoidance* only. That means both components require a change in their timing contracts and their implementation. On the other hand, the *Switching Logic* component will require only a change to its contract assumptions as its expectation regarding the behavior of the previous components has changed.
Table II summarizes the results of the impact analysis. When analysing the results, one can conclude that the behaviour of the ACC system can be updated in case that we update both the *ACC* and *Collision Avoidance* components as well as changing the *Switching Logic* contract at granularity level 4. This conclusion holds as long as we consider the ACCwCA, ACCwSP and ACC system do not have any other dedicated functionalities than the considered ones. Consequently, we can argue that the verification of the implemented update U_{perf} can be accomplished by verifying the composition and the behavior of the components at granularity level 4 only, while there is no need to verify the components at the higher granularites.

D. Corrective Update Scenario

We consider a corrective update U_{corr} triggered by a violation of one functional contract specifying a threshold value for the acceleration command acc of the *ACC* component (cf. Figure 3). The violation leads to deceleration values which are higher than the specified threshold in the corresponding contract. In order to avoid the resulting strong braking, which may lead to passenger inconvenience, a change in the implementation of the *ACC* module is made.

1) Possible Update Reasons: The main reasons for corrective updates are as described in Table I an arising inconsistency during the design process or not verifying specific assumptions during testing. As shown in [30], the inconsistencies may occur during the refinement of the design towards the realization in code, or because of overlooking an assumption or a guarantee during development. In addition, as testing every possible situation in which the system can exist is impossible and very time-consuming, unexpected behavior of the system may occur during usage due to untested situations. Contract violations can be detected by run-time monitors run on a specific CPS middleware.

2) Impact Analysis and Incremental Verification: As the contracts should stay the same after the update ($\Delta\mathcal{C} = 0$), the only required change is to adapt the implementation of *ACC*. In order to fix the issue of too high deceleration values, we check the component's implementation change possibility according to Figure 2. Then, we derive a corresponding change $\Delta Impl$ by adding a saturation block to the used Simulink model directly after the PID controller. The introduced additional block bounds the input to the lower limit value $-3.0\,\mathrm{m\,s^{-2}}$ without changing the parameters of the PID controller. After applying the change to the implementation model, we conduct satisfaction checks considering the old module's bug by running a system simulation and checking the results.

VI. DISCUSSION AND CONCLUSION

In this work, we show an approach to incrementally check evolving CPS by applying contracts. Based on the ACC use

case, we demonstrate that the validation process for changed modules can be accelerated by limiting the extent of the respective update to only those system areas that are affected by the change and have to be re-verified. This saves the need to re-test the entire system. By using a contract-based integration method combined with the notion of deltas Δ, it is possible to examine the composition of the overall system, both top-down and bottom-up, at different levels of granularity, to detect integration problems, and to formally validate and verify them. Although several publications, e.g. [7] [16], investigated thoroughly the use of contracts and formal verification for verifying and validating embedded systems, they did not show how to use those approaches for efficient incremental checks, which we cover for the important case of safety-critical software updates. Other works suggesting approaches for incremental-verification such as [11] and [12] did not explicitly target the integration into an incremental development process. That is why we introduced the impact analysis process with focus on its incremental verification part.

The approach presented in this work provides improved incremental verification through the use of contracts and reduces the use of formal description methodology. Moreover, contracts provide an acceptable level that is applicable and understandable to practitioners and can easily be adapted to changing requirements. In future work, this approach shall be further pursued and expanded by monitoring the contracts also during operation and by delivering data from the field back to the system integrator. It can also be extended to product-lines by enabling the verification of variants.

References

[1] M. Staron, "Automotive software architectures," *Automot. Softw. Archit*, pp. 33–39, 2017.

[2] Y. Dajsuren and M. v. den Brand, "Automotive Software Engineering: Past, Present, and Future," in *Automotive Systems and Software Engineering: State of the Art and Future Trends*, Y. Dajsuren and M. van den Brand, Eds. Cham: Springer International Publishing, 2019, pp. 3–8.

[3] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming dr. frankenstein: Contract-based design for cyber-physical systems*," *European Journal of Control*, vol. 18, no. 3, pp. 217 – 238, 2012.

[4] H. Guissouma, H. Klare, E. Sax, and E. Burger, "An empirical study on the current and future challenges of automotive software release and configuration management," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug 2018, pp. 298–305.

[5] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[6] B. Meyer, "Touch of class: Learning to program well using object technology and design by contract," *Springer*, 2009.

[7] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. A. Henzinger, and K. G. Larsen, "Contracts for system design," *Foundations and Trends in Electronic Design Automation*, vol. 12, no. 2-3, pp. 124–400, 2018.

[8] I. Sljivo, B. Gallina, J. Carlson, and H. Hansson, "Strong and Weak Contract Formalism for Third-party Component Reuse," in *Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'13)*. IEEE Computer Society, 2013, pp. 359–364.

[9] I. Sljivo, J. Carlson, B. Gallina, and H. Hansson, "Fostering Reuse within Safety-critical Component-based Systems through Fine-grained Contracts," in *Proceedings of the International Workshop on Critical Software Component Reusability and Certification across Domains (CSCR'13)*, 2013.

[10] E. Geisberger and M. Broy, *agendaCPS: Integrierte Forschungsagenda Cyber-Physical Systems*. Springer-Verlag, 2012, vol. 1.

[11] K. Johnson, R. Calinescu, and S. Kikuchi, "An incremental verification framework for component-based software systems," in *Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering - CBSE '13*. Vancouver, British Columbia, Canada: ACM Press, 2013, p. 33.

[12] L. Bu, S. Xing, X. Ren, Y. Yang, Q. Wang, and X. Li, "Incremental Online Verification of Dynamic Cyber-Physical Systems," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2019, pp. 782–787.

[13] B.-C. Rothenberg, D. Dietsch, and M. Heizmann, "Incremental Verification Using Trace Abstraction," in *Static Analysis*, A. Podelski, Ed. Cham: Springer International Publishing, 2018, vol. 11002, pp. 364–382.

[14] U. Juhasz, "Incremental Verification," Doctoral Thesis, ETH Zurich, 2016.

[15] Z. Cheng and M. Tisi, "Incremental Deductive Verification for Relational Model Transformations," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Tokyo, Japan: IEEE, Mar. 2017, pp. 379–389.

[16] Y. Umezawa and T. Shimizu, "A formal verification methodology for checking data integrity," in *Design, Automation and Test in Europe*, 2005, pp. 284–289 Vol. 3.

[17] C. L. Heitmeyer, "Formal methods for specifying, validating, and verifying requirements." *J. UCS*, vol. 13, no. 5, pp. 607–618, 2007.

[18] M. Oertel and A. Rettberg, "Reducing re-verification effort by requirement-based change management," in *Embedded Systems: Design, Analysis and Verification*, G. Schirner, M. Götz, A. Rettberg, M. C. Zanella, and F. J. Rammig, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 104–115.

[19] M. Borg, J. L. de la Vara, and K. Wnuk, "Practitioners' perspectives on change impact analysis for safety-critical software–a preliminary analysis," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2016, pp. 346–358.

[20] M. Ring, J. Stoppe, C. Luth, and R. Drechsler, "Change impact analysis for hardware designs from natural language to system level," in *2016 Forum on Specification and Design Languages (FDL)*. IEEE, 2016, pp. 1–7.

[21] W. Damm, G. Ehmen, K. Grüttner, P. Ittershagen, B. Koopmann, F. Poppen, and I. Stierand, "Multi-layer time coherency in the development of adas/ad systems: Design approach and tooling," in *Proceedings of the Workshop on Design Automation for CPS and IoT (DESTION'19)*. ACM New York, NY, USA, 04 2019, pp. 20–30.

[22] A. Cimatti, M. Dorigatti, and S. Tonetta, "Ocra: A tool for checking the refinement of temporal contracts." Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, 11 2013.

[23] "Autosar Adaptive Platform," https://www.autosar.org/standards/adaptive-platform, accessed: 2019-09-04.

[24] M. Moestl and R. Ernst, "Cross-Layer Dependency Analysis for Safety-Critical Systems Design," in *ARCS 2015 - The 28th International Conference on Architecture of Computing Systems. Proceedings*, 2015, pp. 1–7.

[25] IEC, "IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems," 1998.

[26] J. Fenn, R. Hawkins, P. Williams, T. Kelly, M. Banner, and Y. Oakshott, "The who, where, how, why and when of modular and incremental certification," in *2nd IET International Conference on System Safety 2007*, vol. 2007. London, UK: IEE, 2007, pp. 135–140.

[27] A. Haber, H. Rendel, B. Rumpe, and I. Schaefer, "Evolving delta-oriented software product line architectures," in *Large-Scale Complex IT Systems. Development, Operation and Management*, R. Calinescu and D. Garlan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 183–208.

[28] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, "Regression testing in an industrial environment," *Commun. ACM*, vol. 41, no. 5, p. 81–86, May 1998.

[29] S. Vakili, "Design and formal verification of an adaptive cruise control plus (acc+) system," Ph.D. dissertation, 2015.

[30] H. Guissouma, S. Leiner, and E. Sax, "Towards design and verification of evolving cyber physical systems using contract-based methodology," in *2019 International Symposium on Systems Engineering (ISSE)*, 2019, pp. 1–8.