INVITED: Specification, Verification and Design of Evolving Automotive Software

S. Ramesh
General Motors R&D
Warren, Michigan, USA
ramesh.s@gm.com

Birgit Vogel-Heuser
Technical University of Munich
Munich, Germany
vogel-heuser@tum.de

Wanli Chang
Singapore Institute of Technology
Singapore, Singapore
wanli.chang@singaporetech.edu.sg

Debayan Roy
Technical University of Munich
Munich, Germany
debayan.roy@tum.de

Licong Zhang
Technical University of Munich
Munich, Germany
licong.zhang@tum.de

Samarjit Chakraborty
Technical University of Munich
Munich, Germany
samarjit@tum.de

ABSTRACT

Modern automotive systems consist of hundreds of functionalities implemented in software. Moreover, these functionalities are constantly evolving with increasing demand for automation, industry competition and changing sensor and actuator capabilities. Correspondingly, it is important to adapt the engineering and software development processes for such systems to consider fast management of this evolution at minimum cost. Towards this, in this paper, we outline three different problems in the context of evolving automotive software and discuss potential solutions for each of them. First, we outline a framework that can accommodate variability in specifications while developing software for automotive product lines. Secondly, a technique is illustrated to addresses after-sales addition of new features in existing systems by studying corresponding acceptable performance degradation of existing functionalities. Finally, we discuss how an inconsistency management framework and regression verification can ensure consistent evolution of engineering processes for automotive mechatronic systems.

KEYWORDS

Formal Specification and Verification, Inconsistency management, Regression verification, Feedback control systems, Model predictive control, Evolving automotive systems

ACM Reference format:
S. Ramesh, Birgit Vogel-Heuser, Wanli Chang, Debayan Roy, Licong Zhang, and Samarjit Chakraborty. 2017. INVITED: Specification, Verification and Design of Evolving Automotive Software. In *Proceedings of DAC '17, Austin, TX, USA, June 18-22, 2017,* 6 pages.
DOI: http://dx.doi.org/10.1145/3061639.3072946

1 INTRODUCTION

A typical modern passenger vehicle has hundreds of features implemented in software using several components. The software realizes the control of several functions of the vehicle ranging from traditional engine control to active and passive safety systems, brake systems, and adaptive cruise control. Moreover, these features and functionalities are constantly evolving with increasing demand for automation, industry competition and changing sensor and actuator capabilities. Until recently, proprietary standards and processes were followed by vehicle manufacturers (OEMs) for the development of automotive software. The increased complexity and safety demands on these systems have led to new open standards such as AUTOSAR [7] and ISO 26262 [11]. AUTOSAR standard defines a layered component-based software architecture for automotive software development to tackle the complexity issue while ISO 26262 is a functional safety standard defining different safety integrity levels and recommends rigorous verification and validation methods for higher integrity levels (ASIL C and D). In addition, it is also important to adapt the engineering and software development processes for such systems to consider fast management of evolution at minimum cost. Towards this, one can have different perspectives of evolution and approach the problem accordingly.

For example, automotive software systems are always developed as product lines in order to realize a whole variety of functionalities and platforms. Correspondingly, it may be said that design specification evolves across different generation of products and also across different models in the same generation. In such an evolution, the features and subsystems reuse common core sets of functionality but have variability which are configured late in design cycle to produce a specific software for a particular vehicle class. The specification, design and verification of these features contain variability which make the problem of their development and analysis very complex.

Furthermore, towards evolving automotive software, one can also consider the case where new software modules are added onto an existing system. This scenario stems from the fact that new automotive applications, like in the domains of Advanced Driver Assistance Systems (ADAS) and autonomous driving, are becoming available at shorter time spans as compared to the lifetime of a car. Correspondingly, there is an increasing emphasis on after-sales integration of new software modules onto existing systems. However, addition of new modules is not trivial as they may impact the performance of existing functionalities and may also jeopardize the safety of the system.

Another important aspect in the context of evolution is the inconsistency management across different disciplines involved in the development of automotive systems. As automotive systems can be regarded as complex mechatronic systems with tight integration of mechanical, electrical/electronic and software systems, change in design specification of one influence the others. Correspondingly, it may be said that addition of new features into the system not only requires addition of software modules but may also need to install new electrical/electronic sensing devices and/or mechanical actuators. Therefore, it is important to consider the problem of evolution management from a more holistic perspective where

DAC '17, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/3061639.3072946

consistency among different heterogeneous models across different engineering disciplines and design phases must be verified.

In this paper, we consider each of the three aforementioned problems and discuss potential solutions for them. First, we outline a framework that can accommodate variability in specifications and accordingly design and verify systems from their state transition models (Sec. 2). Second, we describe a technique to add new applications onto existing systems that allows changes in certain parameters of existing applications to the point such that the resultant systems still satisfy requirement specification (Sec. 3). Next, we discuss how an inconsistency management framework and regression verification can ensure consistent evolution in the engineering processes for mechatronic products (Sec. 4). Finally, we conclude in Sec. 5 by emphasizing the need for more work in this area.

2 FORMAL MODELING AND VERIFICATION OF AUTOMOTIVE PRODUCT LINES

In this section, we propose a rigorous approach to the specification and analysis of requirements and design of evolving automotive software product lines. This approach employs rigorous state transition models for precisely capturing the system behavior which can be subjected to automated analysis. Our modeling and analysis framework is designed to take into account the evolutionary nature and product line structure of automotive systems. The requirement specification framework enables incremental development which can be interleaved with analysis steps. The underlying state transition systems are equipped with a variability specification mechanism that helps in evolving and refining the specification over the development cycle.

In the rest of this section, we will summarize our framework highlighting the different features of incremental development of requirements and the product line verification scheme. For more details, the readers are referred to [16, 18]. Our focus in this paper will be on the discrete control aspects, though the proposed approach has been extended to continuous and hybrid frameworks [14].

2.1 Evolutionary Requirements Framework

The proposed framework, as the name suggests, caters to evolving systems or features for which the goals are fully understood but the system is in its early stages of development. Here, many aspects of the underlying technology not fully understood and are still evolving, as in the case of autonomous vehicles and ADAS features. Three main features associated with the development of evolving systems are: loose semantics, compositionality and integrated simulation and/or analysis capability. The loose semantics allows underspecification which can be "filled in" as the (understanding of) the system evolves. The analysis and simulation capability enables identification of new behaviors or refinement/removal of existing behaviors. It should be easy to add/remove existing behaviors to the requirements for an evolving system.

2.1.1 Structured Transition Systems. The framework employs a formalism called Structured Transitions Systems (STS) which is based upon the well-known Mealy Machines; mealy machines are finite state machines with transitions labeled with input and output events. STS is a declarative formalism specifying a set of mealy state machines described in terms of properties of states and transitions. A simple example of a STS specification is given below:

```
INPUT fail,reset;
OUTPUT notify,eps;
TYPE Mode:{MANUAL,AUTOMATIC};
TYPE Status:{DISABLES,OFF,ENGAGED,FAILED};
VAR mode: Mode; VAR status: Status;
INIT (status==OFF);
TRANSITION true<fail/notify> (status==FAILED);
TRANSITION true<reset/eps>(status==OFF);
```

The specification defines the inputs, outputs, state variables defining the system states and the transitions between the states. `mode` and `status` are state variables which take different values in different states. INIT clause defines the initial states of the system and the two TRANSITION clauses constrain the possible transitions between the states. For instance, the second TRANSITION clause states that any state transition to a state in which the status variable assumes the value OFF under a reset input event.

Note that the above description defines not a single concrete state machine but a whole family of machines constrained by the various clauses in the specification. New clauses can be added to further constrain or modify the set of machines implied by the specifications. For instance, one can introduce the following additional input clause to the above description:

```
INPUT set_auto,set_manual
```

which introduces a whole lot of state machines that have new transitions labeled with the new input events. Similarly, one can add new clauses of other kinds, e.g., transitions which can constrain the set of allowed transitions. This capability of STS is very useful and fundamental to building evolutionary requirements.

There are two main operations, `zoom` and `focus` provided to construct complex STS specifications from simpler ones. As the name suggests, `zoom` defines a superstate that identifies a subset of states in which additional constraints may be placed. For instance, the following clause

```
ZOOM (status == FAILED)
     {
       CONSTANT mode ON set_auto,set_manual WITH eps
     }
```

specifies a superstate that consists of all those states in which the state variable `status` is `FAILED` and that constrains the transitions labeled with input events `set_auto` or `set_manual` among these states to keep the state variable `mode` unchanged, as indicated by the keyword `CONSTANT`.

In contrast, `focus` operation performs an abstraction of the state space by projecting it onto a subset of state variables and specifying additional (constraints on the) behaviors. Consider the example

```
FOCUS mode[fail,reset/notify,eps]
      {
      MACHINE
              {
              STATE s1[mode==MANUAL];
              STATE s2[mode==AUTOMATIC];
              s1<fail,reset/?> s1
              s2<fail,reset/?>s2
              }
      }
```

The above specification essentially restricts the transitions under the input events `fail` and `reset` not to change the mode component of the state.This is done by focussing the attention on the 'mode' component of the states and specifying the type of transitions between them. The above example also illustrates another construct `MACHINE` which allows explicit specification of a mealy machine.

Another construct available in the STS formalisms is `Simulate`. The following clause that can be added to an STS specification is an interesting example of the usage of the construct.

```
!SIMULATE (status==DISABLED)<fail/?><reset/?> (status ==OFF);
```

This when included as a clause in our example, excludes a run that takes the state from DISABLED to OFF via a sequence of

inputs `fail;reset`. The negation symbol ! is used for specifying the absence of the indicated behavior.

For a more detailed discussion on the syntax and semantics of STS, the reader is referred to [18].

2.1.2 Analysis. The declarative framework of the formalism enables rigorous analysis of the specifications. Besides the standard consistency and ambiguity analysis, three kinds of analysis can be performed on specifications using the operators provided in the framework:

- Abstraction of a transition system focusing on a subset of state variables and construct an abstract state transition system based upon this subset. This analysis can be directly realized by using the `FOCUS` operation.
- State Space Restriction derives a transition system whose states satisfy a list of predicates which can be achieved by using the `zoom` operator.
- Finally, the simulation operator is used to generate scenarios from the specification which can be analysed for inclusion or removal.

The formalism was applied to develop requirements for many in-house systems in GM. A novel active safety feature was specified using the formalism by an engineer not trained in formal methods. The original specification of this feature is a 50-page document in natural language which was converted into the STS formalism which consists of 9 state variables, 20 input/output signals and around 120 clauses. During the analysis of this specification, a few new corner case scenarios were found and this led to the improvement of the requirements specification.

2.2 Product Line Modeling and Verification

As discussed in Sec. 1, automotive features and subsystems are developed as a product line that share a common core set of functionality. A central feature of product lines is variability, which requires careful attention in all the stages of development, including requirements. One challenging aspect of automotive systems that we encountered is their evolutionary nature, which extends even to variability. As the features and systems evolve, new variation points and variability may be introduced. We have extended the STS formalism to include the specification of variability.

2.2.1 Variable State Transition Systems. A variable STS is an STS enhanced with a set of special configuration variables which assume finitely many values. Each valuation of these variables define a variant. The difference between a state variable and a configuration variable is that the transitions do not change the values of the configuration variables. Another difference is that the values of configuration variables decide whether a transition is to be included or not in the specification of a variant.

As a simple example, consider the transition system described graphically in Figure 1. This captures the behavior of a simple *Door lock* feature. There are three configuration variables in this state machine which are given along with their possible values in the upper box in the figure. The lower box defines a constraint over the possible valuations of these variables; this constraint states that whenever the variable $Transmission_{dl}$ assumes the value *Manual*, DL_User_Pref takes the value *Speed*. There are six allowed configurations and hence variants defined by this machine. Each transition label is prefixed with a constraint involving the configuration variable that determines whether a transition is to be included in a variant or not. For instance, the transition label prefix *Speed* indicates that this transition is included in the description of the behavior of only those variants for which the variable DL_User_Pref assumes the value *Speed*. A transition with no prefix is allowed in all the variants.

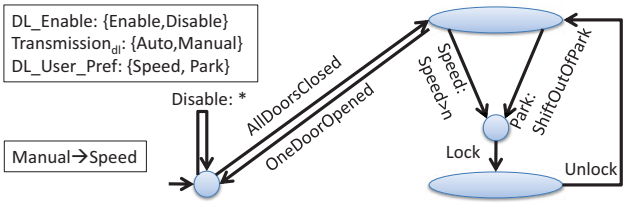

Figure 1: Variable STS for *Door lock* Feature

All the analysis operations defined in the previous section extend to variable STS as well and this helps in consistency and ambiguity analysis to be carried out across an entire product line. Also, the formalism supports evolution of product line requirements by way of introduction or revision of configuration variables. Further, the operations `focus` and `zoom` can be used to 'focus' and 'zoom' into a specific product variant or a subset of variants to specify variant specific behaviors.

A special operation called *instantiation* is defined for variable STS which is useful for product line requirements. Given a variable STS, the instantiation helps in extracting the requirements applicable to a specific variant or a subset of variants. This helps in isolating variant specific requirements and then analyzing them or revising them. This also helps in inserting variant specific requirements to the existing requirements. One specific use case is that new product variants can be introduced that leads to the evolution of the product line structure.

2.2.2 Design Verification. Formal specification of requirements helps not only in removing ambiguities and inconsistencies early in the development cycle, but helps in validating the designs and their implementations. The design step involves refining the high level behaviors specified in the requirements into lower abstraction levels that are realized on a computational platform. For example, the state variables and the operations on these variables are realized using appropriate data structures available in the implementation machine. In the automotive domain, the implementation often contains multiple variants which are configured late at the deployment time to a specific variant. To facilitate late configuration, the implementation uses calibration parameters to specify variability and they can be easily set to specific values to get a specific variant.

In order to verify that an implementation specifies an STS requirement, we propose modeling the implementation also as a variable STS and define a formal notion of refinement of variable STS specifications. Given a variable STS A, we define $Config(A)$ to be the set of all variant STS defined by A, by instantiating the configuration variables of A to valid values. Further, we associate a behavior for a standard STS B (the one without any configuration variable), denoted by $Beh(B)$, to be the set of all possible input-output sequences allowed by B. Then we can define for a pair of variable STS A, A' a notion of refinement, denoted by ref, as follows: $A\ ref\ A'$ if for every variant B in $Config(A)$, there exists a variant B' in $Config(A')$ such that $Beh(B) \subseteq Beh(B')$.

We have developed efficient methods for checking refinement using standard model checking techniques and the readers are referred to [16] for more details. To illustrate the notion of refinement, consider the variable STS given in Figure 2. This STS can be shown to be an implementation of the STS introduced earlier in Figure 1. Let us denote these two STSs by Des_{dl} and Req_{dl} respectively. The structure of Des_{dl} is similar to Req_{dl} except that the top elliptical

shaped state in Figure 1 is split into two states (the top and the bottom elliptical shaped states) in Figure 2. The top state is for auto-transmission whereas the bottom one is for manual transmission as can be seen from the configuration labels of the two transitions going from the initial state. Two configuration variables $Cp1$ and $Cp2$ are used in Des_{dl}. The box in Figure 2 depicts the set of possible values of these. $Cp1 = Auto$ corresponds to the configuration in which the transmission is $Auto$ whereas $Cp1 = Moff$ corresponds to either the manual transmission or the case when $Cp1$ is disabled; similarly, $Cp2 = Speed$ means that the user preference is set on $Speed$, while $Cp2 = Poff$ means either $Park$ or the case when $Cp2$ is disabled. It can be shown that Des_{dl} refines Req_{dl}.

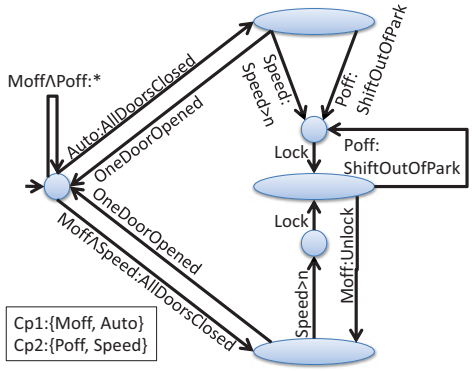

Figure 2: Des_{dl}: *Door lock* Implementation STS

3 EVOLVING AUTOMOTIVE CONTROL SOFTWARE DESIGN

When new control software modules are implemented on automotive platforms, it is desirable that the existing pre-tested and verified code is not changed. Taking this into consideration, we discuss a design method for evolving automotive control software systems design. Here, as new control tasks are added, only the sampling periods of the existing tasks need to be reconfigured. As will be explained later in this section, the sampling period is an input parameter to the control software, and correspondingly, the code can be kept unchanged. We illustrate the technique with feedback control applications.

3.1 Basics of Feedback Control Applications

Plant dynamics: The dynamic behavior of a linear time-invariant (LTI) single-input-single-output (SISO) plant can be modeled by a set of differential equations,

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t), \\ y(t) &= Cx(t), \end{aligned} \tag{1}$$

where $x(t) \in \mathbb{R}^l$ is the system state, $y(t)$ is the system output, and $u(t)$ is the control input. The matrices A, B and C represent physical properties of the plant. Here, the control input $u(t)$ is computed and applied to the plant to achieve certain desired control performance.

In automotive systems, the controller is implemented in a digital fashion. Therefore, the system states must be sampled when measured by the sensors. Assuming the sampling period to be h, the sampled system state and output can be represented as

$$x[k] = x(t_k), \quad y[k] = y(t_k), \quad t_k = kh, \quad k = 0, 1, 2, 3, \cdots. \tag{2}$$

The control input is denoted as $u[k]$, and is given by

$$u[k] = u(t), \quad t_k \le t < t_{k+1}. \tag{3}$$

Correspondingly, the discretized dynamics of (1) is

$$\begin{aligned} x[k+1] &= A_d x[k] + B_d u[k], \\ y[k] &= Cx[k], \quad \text{where} \\ A_d &= e^{Ah}, \ B_d = \int_0^h (e^{A\tau'} d\tau')B. \end{aligned} \tag{4}$$

Control performance: A popular performance metric in the control context is the quadratic cost function, which is given by

$$J = \sum_{k=0}^{N-1} (x^T[k]Qx[k] + u^T[k]Ru[k]) + x^T[N]Sx[N], \tag{5}$$

assuming that the system state $x[k]$ is expected to stabilize at 0. Q is a positive semi-definite weight matrix, R is a positive definite weight matrix, and S is a positive semi-definite matrix. Since SISO applications are considered, $u^T[k] = u[k]$. To optimize the control performance, J is minimized. Among the three terms, $x^T[k]Qx[k]$ penalizes the transient state deviation, $u^T[k]Ru[k]$ penalizes the control effort, and $x^T[N]Sx[N]$ penalizes the finite state deviation.

Model Predictive Control: The goal is to find a sequence of control inputs $u[0], u[1], \ldots, u[N-1]$ that minimizes the quadratic cost function J, where N is the horizon. Assuming that $S = Q$ and bringing (4) into (5), we have

$$J = \frac{1}{2} U^T H U + x^T[0] F U + x^T[0] Y x[0], \tag{6}$$

where $U = [u[0], u[1], \ldots, u[N-1]]^T$. The matrices H, F, and Y are dependent on A_d, B_d, Q, and R. Details can be found in [6]. Correspondingly, there is a large number of algorithms reported to solve this optimization problem in model predictive control (MPC) [17]. Clearly, the sampling period h is an input parameter to the software, and therefore, the code remains unchanged, if we reconfigure the sampling periods.

3.2 Automotive Operating Systems and Sampling Periods

Automotive control software is typically implemented on operating systems such as OSEK/VDX that support a limited set of predefined sampling periods [8]. An example set is

$$\phi = \{1ms, 2ms, 5ms, 10ms, 20ms, 50ms, 100ms\}. \tag{7}$$

Denoting e_i and h to be the worst-case execution time (WCET) and sampling period of a control application, C_i, respectively, the processor load for C_i is

$$L_i = \frac{e_i}{h}. \tag{8}$$

The upper bound on the load of a processor is denoted as B_u. Considering a single processor p,

$$\sum_{\{i | C_i \text{ runs on } p\}} L_i \le B_u. \tag{9}$$

Here, the value of B_u depends on the scheduling policy. For uniform sampling, B_u is equal to 1. A variety of tools, such as Inchron [1], Timing Architects [3], and SymtaVision [2], are used in the industry for more general schedulability analysis.

3.3 An Illustrative Example

In order to illustrate the evolving automotive control software design technique, we consider a five-state electronic wedge brake (EWB), which is a brake-by-wire solution developed by Siemens [12]. MPC is used as the control algorithm and the horizon N is set to be

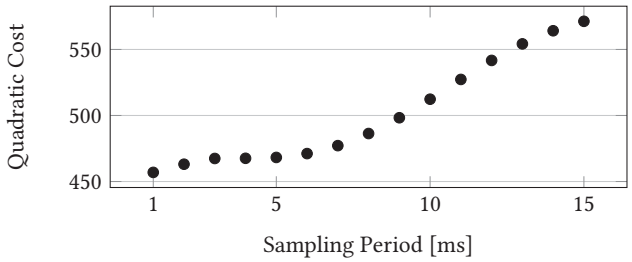

Figure 3: The relationship between the control performance and the sampling period for the EWB

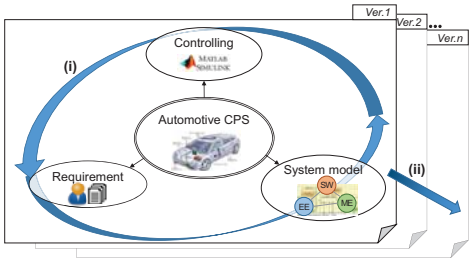

Figure 4: Inconsistency management over the models within a version (i) and over the versions (ii)

4. The quadratic cost function in (5) is the control performance metric and the requirement is 500. The relationship between the control performance and the sampling period is reported in Figure 3.

We assume that there are initially three applications running on the processor. For the sake of better and simple illustration, all the applications are taken as identical to the EWB. The WCET of each application is $0.6ms$. Considering the set ϕ, the shortest sampling period that can be assigned to all the three applications, without violating the processor load constraint, is $2ms$. The total processor load is then $0.6/2 \times 3 = 0.9 \leq 1$. The control performance is 463.

When a new control application, which is also taken as identical to the EWB, is to be added, it is not possible to schedule all the four applications on the processor with the sampling period of $2ms$, since the total processor load would be 1.2, larger than 1. Therefore, we reconfigure the sampling period to be $5ms$. The processor load then becomes $0.6/5 \times 4 = 0.48$. The control performance becomes 468, which is slightly deteriorated, yet still satisfies the control performance requirement. In fact, this change of sampling period enables the addition of more control applications. This evolving process of automotive control software only requires a change of the sampling period, which can be determined during the design phase, while keeping the program code unchanged. When there are different applications with different or even non-uniform sampling periods, a more sophisticated schedulability analysis is required.

4 CONSISTENT EVOLUTION IN DESIGN PROCESSES FOR MECHATRONICS

Automotive systems can be regarded as complex mechatronic systems or even cyber-physical systems (CPS) due to the tight integration of mechanical, electrical/electronic and software systems. Heterogeneous models are adopted in different design phases to show different abstraction levels and different viewpoints of the automotive system. The rapid changes in technology and customer needs results in evolution of these models, and therefore, it is essential to maintain consistency among interacting mechatronic models and software versions [22]. In this section, we study automotive systems from the point of view of cross-disciplinary engineering process and address the consistent evolution of such systems concerning different design layers: from high-level models to software.

4.1 Focusing on the engineering process

In the development of automotive systems, heterogeneous models are involved, e.g., user requirements, object-oriented mechatronic models, dynamic simulation models, manufacturing plans, and test cases. During the engineering process, an efficient and effective collaboration between individual research departments is a crucial factor for the overall success [13]. When some models in the system need to be changed (e.g., updating electrical interfaces) or refined (e.g., adding a dynamical property to a motor), inconsistencies may

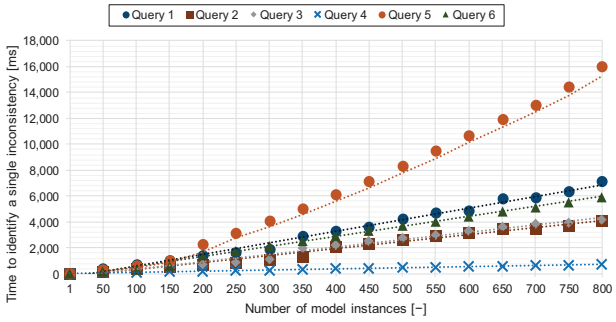

Figure 5: Time to identify a single structural inconsistency depending on the number of model instances. Each model instance represents a single model of a CPS with 100 inputs and outputs [10].

occur between domain-specific models that are associated with the changed/refined parts. The associated model may be deployed in a different phase of the engineering life-cycle. In order to capture the model dependencies and maintain the overall consistency, an inconsistency management framework is proposed, which consists of two parts: cross-disciplinary model representation and derivation of model interaction [9]. Semantic Web technologies prove to be promising to identify and manage inconsistencies in this framework. In addition, a central graphical modeling language is put forward as an essential part of the inconsistency management approach [10]. By means of such a modeling language, inconsistencies can be flexibly specified, diagnosed and handled. In this work, inconsistencies are defined in the level of the meta-model, where interconnections among models are specified by rules based on triple graph patterns [10]. They serve as an engineer-friendly graphical tool to reduce manual efforts.

Concrete applications of the proposed approaches can be:

- Model checking, considering structural compatibility, e.g., representing models in Systems Modeling Language (SysML) and checking entity constraints with the Object Constraint Language (OCL).
- Global inconsistency checking and resolving among interdisciplinary engineering models (Fig.4-i), e.g., representing different models in a common knowledge base with the Web Ontology Language (OWL) and checking the same object in different models with query languages.
- Time-spanning consistency checking along the engineering life cycle (Fig.4-ii), e.g., continuously checking updated requirements and test cases by querying respective attributes when the whole system evolves.

The proposed inconsistency management framework shows good scalability in the first evaluation. As an use case, we can consider that an electrical part is replaced by a new version. Correspondingly, it is required to analyze for potential inconsistencies in mechanical or software parts, e.g., whether the newly integrated device provides the necessary interface or whether an upper bound is respected. For the evaluation, 800 SysML model instances are used with 100 inputs and outputs in each model (Fig. 5). Different queries are conducted for the inconsistency check, e.g., query 4 checks whether an analog output is provided and query 5 checks whether the current of all related components is limited. In this case, the processing time varies from 2 to 16 sec [1], which is acceptable for engineers.

However, the above mentioned inconsistency management is feasible only when commonly accepted vocabularies, i.e., semantics, are explicitly defined. In CPS design, one potential enabler to implement a cross-disciplinary applicable concept is to use a multitude of triple stores that support Semantic Web technologies, which can provide higher flexibility than traditional relational databases.

4.2 Focusing on the product model

Additionally, the detection of behavioral inconsistencies in the components' interface require analysis using rich interface specifications using [5, 15] or combined simulation (MATLAB/Simulink) and model checking approaches for specific constraints [23]. Using formal verification, correctness of a system can be proved rather than only identifying faults. Simulation shows the exact time behavior of a system. A key challenge to combine both approaches is to identify an appropriate level of abstraction of both models.

Moreover, modifications on the system may introduce unwanted behavior, so called 'regression'. Undetected regressions may cause severe consequences. Especially for automotive systems which are one of the most safe-critical systems, this might be the most important engineering consideration since a car is supposed to interact directly with humans, and correspondingly, unexpected regressions may cause catastrophic accidents.

Although formal verification is well known for its power to prove the behavior of the system by exhaustively exploring all reachable states, it is not realistic to derive functional or behavioral specification for industrial size programs, like in automotive system. This is a big barrier towards applying formal methods. Regression verification [19] is an approach that attempts to ensure absence of regression with formal verification techniques without the overall system specification and this was successfully integrated on mechanical systems in [4, 21]. Recently, a verification-supported evolution methodology was introduced in [20] which uses regression verification. Moreover, new behaviors are verified using delta verification.

5 CONCLUDING REMARKS

Due to increasing demand for automation, industry competition and changing sensor and actuator capabilities, automotive systems are evolving at different scales. Correspondingly, in this paper, we have considered three problems in the context of evolving automotive software and discussed potential solutions for them. First, we have outlined a framework which allows the designer to specify requirements corresponding to different vehicle variants to address evolution across vehicle product lines. Second, we have illustrated that in order to accommodate new applications, one can exploit the fact that change in certain parameters of existing control applications does not result in violation of acceptable performance bounds. Finally, we have emphasized on the importance of inconsistency management frameworks and regression verification to ensure consistent evolution in of engineering process for mechatronic products such as automotive systems. Although rapid evolution of automotive systems is well-acknowledged, there have not been enough efforts to address fast management of evolution cycles in a comprehensive way in this domain. Different aspects of evolving automotive systems have been studied separately; however, in the future, it may be necessary to integrate these aspects into a single holistic framework.

[1] For the implementation, Fuseki was used, which is part of the Apache Jena Framework. Evaluations were run on a standard office PC (Windows 7 x64 platform, 16 GB RAM, 4 cores, 3.6 GHz).

REFERENCES

[1] 2017. Inchron GmbH. https://www.inchron.de/. (2017).
[2] 2017. Symtavision GmbH. https://www.symtavision.com/. (2017).
[3] 2017. Timing Architects. http://www.timing-architects.com/. (2017).
[4] B. Beckert, M. Ulbrich, B. Vogel-Heuser, and A. Weigl. 2015. *Regression Verification for Programmable Logic Controller Software.* Springer International Publishing, Cham, 234–251. DOI: https://doi.org/10.1007/978-3-319-25423-4_15
[5] M. Broy and K. Stølen. 2001. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement.* Springer-Verlag New York, Inc., Secaucus, NJ, USA.
[6] W. Chang and S. Chakraborty. 2016. Resource-aware Automotive Control Systems Design: A Cyber-Physical Systems Approach. *Foundations and Trends® in Electronic Design Automation* 10, 4 (2016), 249–369.
[7] AUTOSAR Consortium. 2015. AUTOSAR Specification R4.2.2. (2015). http://www.autosar.org/standards/classic-platform/release-42/
[8] OSEK/VDX Consortium. 2005. OSEK/VDX operating system specification Version 2.2.3. (2005).
[9] S. Feldmann, S. J. I. Herzig, K. Kernschmidt, T. Wolfenstetter, D. Kammerl, A. Qamar, U. Lindemann, H. Krcmar, C. J. J. Paredis, and B. Vogel-Heuser. 2015. Towards Effective Management of Inconsistencies in Model-Based Engineering of Automated Production Systems. *IFAC-PapersOnLine* 48, 3 (2015), 916 – 923.
[10] S. Feldmann, M. Wimmer, K. Kernschmidt, and B. Vogel-Heuser. 2016. A comprehensive approach for managing inter-model inconsistencies in automated production systems engineering. In *Proc. of 2016 IEEE International Conference on Automation Science and Engineering (CASE).* Fort Worth, TX.
[11] International Organization for Standardization (ISO). 2011. ISO 26262 Standard Specification. (2011). http://www.iso.org/iso/catalogue_detail?csnumber=43464
[12] J. Fox, R. Roberts, C. Baier-Welt, L. Ho, L. Lacraru, and B. Gombert. 2007. *Modeling and Control of a Single Motor Electronic Wedge Brake.* Technical Report. SAE.
[13] A. Kohn, J. Reif, T. Wolfenstetter, K. Kernschmidt, S. Goswami, H. Krcmar, F. Brodbeck, B. Vogel-Heuser, U. Lindemann, and M. Maurer. 2013. *Improving Common Model Understanding Within Collaborative Engineering Design Research Projects.* Springer India, India, 643–654.
[14] S. N. Krishna, G. K. Narwane, S. Ramesh, and A. Trivedi. 2015. Compositional Modeling and Analysis of Automotive Feature Product Lines. In *Proc. of 2015 52nd ACM/EDAC/IEEE Design Automation Conference.* San Francisco, CA.
[15] C. Legat, J. Mund, A. Campetelli, G. Hackenberg, J. Folmer, D. Schütz, M. Broy, and B. Vogel-Heuser. 2014. Interface behavior modeling for automatic verification of industrial automation systems' functional conformance. *Automatisierungstechnik* 62, 11 (2014), 815–825.
[16] J.-V. Millo, S. Ramesh, S. N. Krishna, and G. K. Narwane. 2013. Compositional Verification of Software Product Lines. In *Proc. of 10th Inernational Conf. Integrated Formal Methods.* Turku, Finland.
[17] J. Rawlings and D. Mayne. 2009. *Model Predictive Control: Theory and Design.* Nob Hill Publishing.
[18] P. Sampath, S. Arora, and S. Ramesh. 2011. Evolving Specifications Formally. In *Proc. of 2011 IEEE 19th Conf. on Requirement Engineering.* Trento, Italy.
[19] O. Strichman and B. Godlin. 2008. *Regression Verification - A Practical Way to Verify Programs.* Springer Berlin Heidelberg, Berlin, Heidelberg, 496–501.
[20] S. Ulewicz, M. Ulbrich, A. Weigl, M. Kirsten, F. Wiebe, B. Beckert, and B. Vogel-Heuser. 2016. A verification-supported evolution approach to assist software application engineers in industrial factory automation. In *Proc. of 2016 IEEE International Symposium on Assembly and Manufacturing (ISAM).* Fort Worth, TX.
[21] S. Ulewicz, B. Vogel-Heuser, M. Ulbrich, A. Weigl, and B. Beckert. 2015. Proving equivalence between control software variants for Programmable Logic Controllers. In *Proc. of 2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA).* Luxembourg.
[22] B. Vogel-Heuser, A. Fay, I. Schaefer, and M. Tichy. 2015. Evolution of software in automated production systems: Challenges and research directions. *Journal of Systems and Software* 110 (2015), 54 – 84.
[23] B. Vogel-Heuser, J. Folmer, T. Aicher, J. Mund, and S. Rehberger. 2015. Coupling simulation and model checking to examine selected mechanical constraints of automated production systems. In *Proc. of 2015 IEEE International Conference on Industrial Informatics (INDIN).* Cambridge.