International Journal of Software Engineering
and Knowledge Engineering
Vol. 29, No. 1 (2019) 1–21
© World Scientific Publishing Company
DOI: 10.1142/S0218194019500013

A Method for Measuring the Constraint Complexity of Components in Automotive Embedded Software Systems

Mohit Garg* and Richard Lai†

Department of Computer Science and Information Technology
La Trobe University, Victoria 3086, Australia
**m.garg@latrobe.edu.au*
†r.lai@latrobe.edu.au

Received 8 June 2016
Revised 6 July 2016
Accepted 25 July 2017
Published 27 June 2018

The rapid growth of software-based functionalities has made automotive Electronic Control Units (ECUs) significantly complex. Factors affecting the software complexity of components embedded in an ECU depend not only on their interface and interaction properties, but also on the structural constraints characterized by a component's functional semantics and timing constraints described by AUTomotive Open System ARchitecture (AUTOSAR) languages. Traditional constraint complexity measures are not adequate for the components in embedded software systems as they do not yet sufficiently provide a measure of the complexity due to timing constraints which are important for quantifying the dynamic behavior of components at run-time. This paper presents a method for measuring the constraint complexity of components in automotive embedded software systems at the specification level. It first enables system designers to define non-deterministic constraints on the event chains associated with components using the AUTOSAR-based Modeling and Analysis of Real-Time and Embedded systems (MARTE)-UML and Timing Augmented Description Language (TADL). Then, system analysts use Unified Modeling Language (UML)-compliant Object Constraint Language (OCL) and its Real-time extension (RT-OCL) to specify the structural and timing constraints on events and event chains and estimate the constraint complexity of components using a measure we have developed. A preliminary version of the method was presented in M. Garg and R. Lai, Measuring the constraint complexity of automotive embedded software systems, in *Proc. Int. Conf. Data and Software Engineering*, 2014, pp. 1–6. To demonstrate the usefulness of our method, we have applied it to an automotive Anti-lock Brake System (ABS).

Keywords: Automotive embedded software system; timing dependency; constraint complexity; software measure; component-based systems.

1. Introduction

Automotive embedded software systems are integration-centric with a focus on assembling existing, third-party developed, black-box type components [4, 14].

Notably, dynamic interaction among these components and their reuse has increased complexity significantly [9, 29] and requires managing end-to-end timing requirements in terms of performance, scheduling, communication, and synchronization [12, 16, 31]. Software complexity management is thus becoming a limiting factor to ensure interoperability and the successful composition of heterogeneous components onto the network's Electronic Control Units (ECUs) [4, 11, 36]. Despite the availability of the event-driven AUTomotive Open System ARchitecture (AUTOSAR) methodology [1], which provides guidelines to automotive manufacturers on TIMing MOdeling (TIMMO) and the analysis of end-to-end timing constraints in order to maintain complexity at a manageable level [8, 22], the complexity measurement of timing constraints remains a challenging task [26]. The more we are able to measure the complexity of a system, the more we can understand its associated reliability [17, 35].

Figure 1 shows that similar to business systems, individual component descriptions of embedded software systems can be specified using Unified Modeling Language (UML) design tools [8, 11, 13, 22]. From a Component-Based System (CBS) development perspective, the complexity of a system depends on the interaction among its components [9, 29] and the interface properties of components [5, 6]. While interaction complexity is a measure of the degree of interdependence between components, interface complexity measures the internal properties of components characterized by their syntactic specifications and added constraints. In the literature, a number of techniques are proposed for measuring interface complexity [5–7, 24, 34]. At an early development phase, measures of the internal properties and interaction properties of components aid in (a) assessing overall complexity which has direct impact on quality factors of the system such as reliability, testability, reusability, maintainability, performance, and error-rates [29, 34]; (b) ranking various environments that place greater stress on the system due to complex component executions; and (c) minimizing the delays in time to market of an overly complex system where costs of non-conformance are generally high [34, 39].

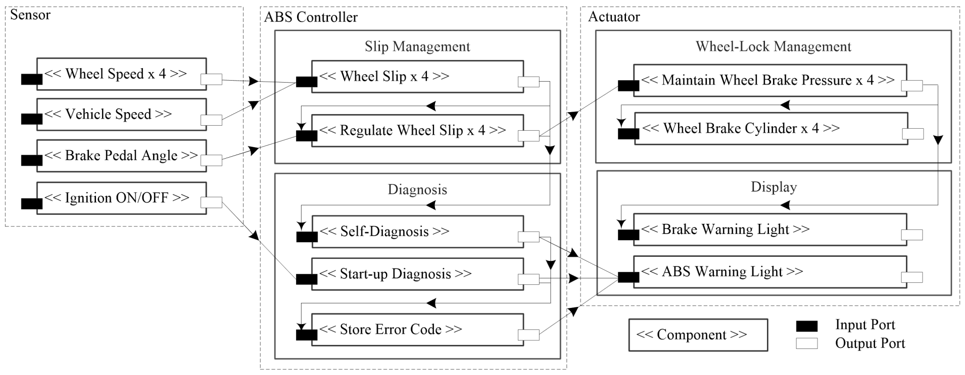

Fig. 1. Functional profile of an automotive ABS.

The constraints associated with a component's interface are divided into two categories: (a) resource constraints, and (b) dependability constraints [26]. The resource constraints are specific to CPU and memory requirements, whereas dependability constraints are a special form of non-functional constraints, associated with usability, timeliness, performance, etc. In this paper, we concentrate on two types of dependability constraints, namely structural [24, 34] and timing [14–16]. While structural constraints focus on I/O signal types and simulation properties which directly affect the feasibility of a component's role in a particular context and its usage in assorted contexts [6, 28], timing constraints constrain a time-dependent attribute of a real-time function by means of an end-to-end relation, abstracting from all implementation details and their timing properties [30]. The constraint complexity values when used in combination with interface complexity values help in obtaining quantitative measure of the internal properties of components. Approaches dealing with constraint complexity measurement [5–7] are however very scarce, and typically only cover the structural constraints, by means of static inspection of the software artifacts [35], rarely taking timing constraints into account, which restrict the behavior of components at run-time [10, 21]. Hence, the development of robust constraint complexity measurement techniques becomes infeasible for obtaining realistic estimates of internal properties at the specification level and thereby quantifying the dynamic nature of components in an embedded software system [34, 35], and requires the employment of sophisticated methods and heuristics.

In [37], we reported a preliminary version of our constraint complexity measurement method for automotive embedded software system. It consisted of two phases: namely (i) functional specification, and (ii) software component specification. As the method was preliminary, the 2014 paper did not include a case study. Since its publication, we have refined our method. In this paper, we focus only on those essential steps through which the constraint complexity of components in an automotive embedded software system can be measured easily and systematically while preserving the essence of the preliminary method.

First, the functional profile of a system is developed using a UML design tool. Second, different events and event chains responsible for logical signal communication in a system are depicted using the AUTOSAR-based UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE-UML) [2]. Third, nondeterministic constraints on the event chains of components are defined using the Timing Augmented Description Language (TADL) [3, 15, 16]. In the following steps, structural and timing constraints on events and event chains are specified using the Object Constraint Language (OCL) [25] and its real-time extension (RT-OCL) [19, 20], and then a measure is applied to estimate the constraint complexity of components. We have extended the constraint complexity measurement technique described in [5] to compute the complexity of a UML Component-Based System Specification (CBSS) which is intended for use in a real-time environment. Unlike our conference paper [37] which did not describe how the preliminary method could be used in a real-life application, in this paper, we discuss the application of our method to an Anti-lock Brake System (ABS) in order to demonstrate the usefulness of our method when used in an automotive embedded software system.

2. Related Work

Among earlier work, the issue of measuring the complexity of component-based software is discussed by Sadigh-Ali *et al.* [34], where complexity of interfaces is interpreted as quality metrics. Cho *et al.* [38] identified that interface and middleware code, needed for integrating different components, are the two main factors affecting the complexity of a CBS. They proposed metrics for computing component static complexity and component dynamic complexity using the combination of the number of classes, interfaces, and the relationships among classes. However, these metrics also require the analysis of component's source code which may not be available. Narasimhan and Hendradjaya [9] stated that higher density of interaction among components in a CBS, characterized by the use of component's interfaces, causes higher complexity level.

To estimate interface complexity with respect to CBS development, very few approaches proclaiming static complexity measurements with the use of a component's syntactic and semantic specifications have been published [5–7].

2.1. *Interface constraints measure*

Gill and Grover [6] have defined different interface complexity measures based on an interface characterization model of a component. According to them, the interface signature, interface constraints, interface packaging and configuration are the factors contributing to the interface complexity of components in a CBS. Interface constraints are defined as constraints associated with a component's interface in terms of its operation semantics. Based on the definition and categorization of interface constraints into either those on individual elements or those concerning relationships among the elements, we interpret that the authors are focusing only on the structural type of constraints. Moreover, there is no information given on how to compute these interface constraints.

2.2. *UML CBSS constraint complexity measure*

Mahmood and Lai [5] extended Gill and Grover's [6] work by proposing a UML CBSS constraint complexity measure where, using OCL clauses, the invariants of constraints associated with each interface are shown as sets of pre- and post-conditions. Their description of constraints reinstates the fact that constraint concerns the use of components in interface configurations. However, their work lacks a discussion on whether there is any complexity arising due to the timing dependencies associated with a component's interfaces. In addition, the application result shows that the proposed measure is suitable for only non-real-time CBS.

2.3. *Temporal constraints validation tool*

Han and Ker [7] have specified temporal constraints as run-time service semantics associated with a component's interface signatures and proposed a tool to validate component interactions against these temporal constraints. The usability of the proposed tool is limited as it only helps in validating the temporal constraints against component interactions in a CBS and there is no argument on the suitability of the proposed tool in measuring these constraints.

In Table 1, we have summarized the strengths and weaknesses of the constraint complexity measurement methods/validation tool described above. While [5, 6] shows that it is essential for a component user to understand and estimate the constraint complexity to enable its proper and precise use, authors in [7] have commented on the usefulness of explicit specification and management of timing constraints which are important for quantifying the dynamic behavior of components at run-time. The measures proposed in [5, 6] are inspiring in terms of measuring the complexity due to structural constraints only and not timing constraints. The encouraging insights gained from the analysis of the strengths and weaknesses of related work and the successful application of the UML CBSS constraint complexity measurement method on CBS motivate us to recast the existing UML CBSS constraint complexity measure in order to obtain a realistic estimate of the constraint complexity of components in embedded software systems.

Table 1. Strengths and weaknesses of the existing constraint complexity measurement methods/validation tool.

Method/Tool	Strengths	Weaknesses
Interface constraints measure [6]	Described interface constraints as constraints associated with a component's interface in terms of its operation semantics	Interface constraints are of structural-type and no information is given on how to compute these constraints
UML CBSS constraint complexity measure [5]	Proposed a UML CBSS constraint complexity measure where, using OCL clauses, the invariants of constraints associated with each interface are shown as sets of pre- and post-conditions	There is no discussion on whether the constraints complexity can be due to added timing dependencies. The proposed measure helps only in computing the structural constraint complexity of components in a non-real-time CBS
Temporal constraints validation tool [7]	Specified temporal constraints as run-time service semantics associated with a component's interface signatures	The usability of the proposed tool is limited as it only validates the temporal constraints against component interactions and not measures these temporal constraints

3. Constraint Complexity Measurement Method

The method presented in this paper concentrates on the system designers and analyst's role at the specification level of an embedded software system's development life cycle [10, 11, 30, 35] for two reasons: first, at this level, CBS development has the highest benefits in terms of adaptability [21, 26]; and second, to provide a unifying view on the process of computing the complexity associated with the internal properties of components in an embedded software system. System designer's role is typically to develop a functional profile, identify events and event chains in a system, and to define non-deterministic constraints on events and event chains of components. At this stage, syntactic descriptions are useful for identifying the function points which help in computing the interface complexity of components in an embedded software system. Measures proposed in [5, 6] can be applied to compute the interface complexity of components. Based on these syntactic descriptions, the system analysts then specify various constraints on a component's interfaces. At this stage, a measure is useful for estimating the constraint complexity of components.

In analogy to the UML CBSS constraint complexity measure [5], our method starts with developing a functional profile of system using a UML design tool. In the next step, designers identify different events and event chains responsible for logical signal communication in a system and depict them using MARTE-UML. Then, non-deterministic constraints on the event chains associated with components are defined using TADL [16]. System analysts then specify structural as well as timing constraints on events and event chains using UML-compliant OCL [25] and RT-OCL [19, 20] due to their simplicity in representing event-driven and time-driven aspects of components in a textual form. This is followed by the application of our measure to estimate the constraint complexity of components.

The way that the constraint complexity of components in an embedded software system is measured makes our method unique and substantially distinct from the technique described in [5]. We use new generation UML design tool, such as MARTE-UML [2] for three reasons: first, to overcome the limitation of traditional UML design tools that they do not support the representation of the non-functional properties and architectural aspects of a CBS [5]; second, to facilitate designers in depicting the conceptual descriptions of components in real-time constraint systems; and third, through TADL, it supports the derivation of abstract system configuration boundaries to define non-deterministic constraints on event chains of components. Apart from OCL, we also use RT-OCL in our method as it enables system analysts to specify constraints covering the state consecutiveness, state transitions and time-bounded constraints on components. On the basis of complexity, the proposed measure computes complexity due to both structural and timing constraints. The structural constraint complexity computation process adapted from the UML CBSS constraint complexity measure [5] is used for this purpose. However, we have revised the formula to also include the computation of the timing constraints.

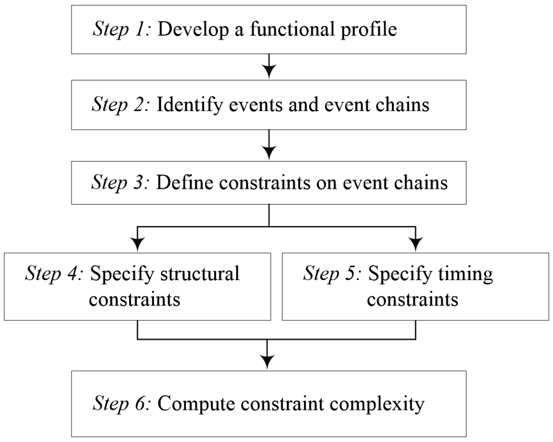
Step 1: Develop a functional profile
Step 2: Identify events and event chains
Step 3: Define constraints on event chains
Step 4: Specify structural constraints
Step 5: Specify timing constraints
Step 6: Compute constraint complexity

Fig. 2. An overview of the constraint complexity measurement method.

Figure 2 depicts an overview of the method and the sequence in which the steps are to be performed.

3.1. *Develop a functional profile*

In the first step, the architectural description of an automotive system, with the use of a UML design tool, is developed. Traditional UML-based tools express models through a rich set of diagrams to describe the static structure, the communication among structural elements, and express restrictions on a system [19], whereas the new generation of UML design tools, such as MARTE-UML [2] and RE-UML [13], support conceptual as well as semantic descriptions of components in real-time constrained systems, and thus benefits system designers by enabling them to design an automotive system's functional profile consisting of individual ECU and component descriptions; perform quantitative timing analysis; and develop complexity management mechanisms such as model views, separation of concerns, and model composition [8, 11].

3.2. *Identify events and event chains*

In the second step, comprehensive syntactic descriptions of events and event chains for each component are defined to gain an understanding of the distributed character of components in an automotive assembly; where individual sensor and actuator components spread over the broad net interact with components in the controller ECU. According to the needs of separation of concerns, event-driven AUTOSAR automotive architecture [1, 12] uses structural models on multiple levels of abstraction, such as Electronic Architecture and Software Tools-Architecture Description Language (EAST-ADL2). Thus, through MARTE-UML, AUTOSAR is able to support the expression of timing constraints in the context of functional architecture by adding end-to-end deadlines to the computation paths; maximum jitter requirements to signals; and time correlation constraints between events [11, 18]. A function specifies how inputs, outputs, and a component's state relate and the effects of calling the functions on that relationship. Since all functions in component-based event-driven paradigms are triggered by events, deadlines are defined based on the static priorities of respective functions [30]. An event, therefore, denotes a distinct form of state change at run-time, whereas an event chain consists of segments consisting of one or more inputs for communication between functions (stimulus) and at least one output of the sequence of times (response) indicating the time during which each event is predicted to occur [15, 33].

3.3. *Define constraints on event chains*

In the third step, system designers consider the logical interdependencies on events to define constraints on event chains using information-type models and/or state charts developed with the use of AUTOSAR-compliant language such as TADL [3, 15, 16]. AUTOSAR-based projects, such as TIMMO, aid designers to define constraints in a methodical manner thus avoiding the possibility of an incorrect constraint definition on events and event chains. The TIMMO project [3, 8] introduced timing constructs at the meta-model level with the use of the model-driven EAST-ADL2 and the MARTE standard to deal with the implementation of the dependent timing properties of components in the form of deadlines, periods, and offsets [1, 15]. TADL uses only a few modeling concepts and properties of the MARTE standard to specify timing constraints [2, 11, 16, 18]. TADL benefits from MARTE standard in two ways: (a) semantically, because MARTE-UML serves as a basis for a UML implementation of the concepts and provides a foundation for the timing analysis of end-to-end delay constraints in event chains; and (b) syntactically, as MARTE-UML uses OCL to annotate many common timing aspects in control systems [8].

The TADL methodology supports the analysis, verification and traceability of timing constraints across different levels of abstraction and development phases through a standardized exchange format. The three types of timing constraints considered by TADL on components are repetition rate, delay, and input/output synchronization. TADL constraints refer to the structural elements via events and event chains [14, 16]. End-to-end timing constraint latency refers to maximum and minimum latency between each stimulus and response occurrence of that event chain. Repetition-rate constraints place restrictions on the distribution of the occurrence of an event [22]. TADL defines such occurrences that can follow any of the periodic, sporadic, pattern or arbitrary distribution types. Delay requirements constrain the sensor sampling semantics solely in terms of the times at which stimulus and response events occur [18]. Using age and reaction types, TADL expresses two fundamentally different perspectives on delay constraints applicable to common events connecting segments [16]. To semantically strengthen the age and reaction constraints, TADL further defines their specializations as input/output synchronization to demonstrate the maximum temporal inclination allowed between input/output events. Figure 3 shows the component view of the end-to-end timing, mostly determined by the logical flow of the signal between the input and output ports of the components in an automotive ABS.

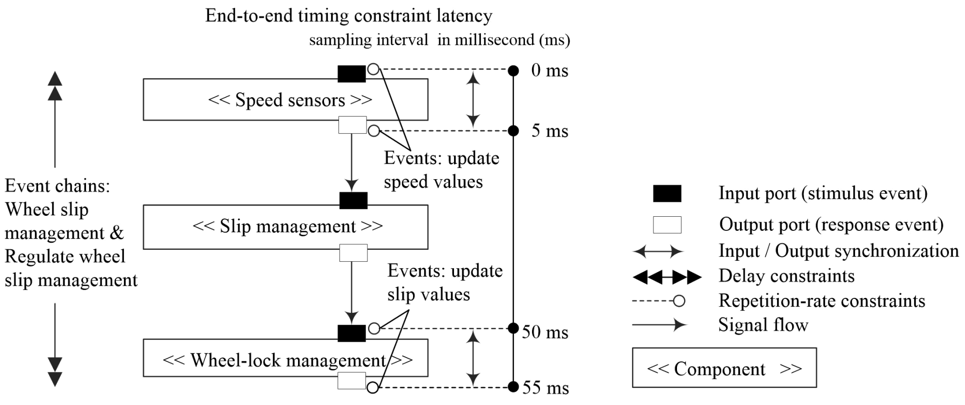
End-to-end timing constraint latency
sampling interval in millisecond (ms)
<< Speed sensors >>
<< Slip management >>
<< Wheel-lock management >>
0 ms
5 ms
50 ms
55 ms
Events: update speed values
Events: update slip values
Event chains: Wheel slip management & Regulate wheel slip management
Input port (stimulus event)
Output port (response event)
Input / Output synchronization
Delay constraints
Repetition-rate constraints
Signal flow
<< Component >>

Fig. 3. End-to-end timing constraints on events and event chains.

3.4. *Specify structural constraints*

In the fourth step of this phase, structural constraints on components are specified using UML-compliant OCL [25]. In OCL, each expression is written in the context of an instance of a specific type, represented by sets of invariants, pre- and post-conditions [27]. While pre-conditions are a predicate over an event's input parameters, post-conditions guarantee that an event in an event chain continues to hold these assertions over both input and output parameters, provided the pre-conditions were true when the event was triggered. Pre-conditions, post-conditions and invariants of a constraint comprise one or more OCL clauses. According to the standard OCL library, "self" is a contextual instance that provides a reference point for the interpretation of an OCL expression. Similarly, other instance types, different to the type represented by a contextual instance, are commonly accessed by OCL functions, such as variable definitions through the "let" expression, "if" expression condition, predefined iterator variables, etc. The occurrence of OCL clauses listed in Table 2 is useful for specifying the structural constraint complexity of components at the specification level [5].

3.5. *Specify timing constraints*

To examine the dynamic behavior of a component, a clear specification of its timing constraints is mandatory. Since standard OCL [25] with operations to select, access, and manipulate values and objects does not sufficiently specify constraints over the dynamic behavior of a UML model, a few authors have proposed OCL-compliant extensions [19, 27], thus, enabling system analysts to specify constraints covering the state consecutiveness, state transitions, and time-bounded constraints on components in embedded software systems. In [19, 20], such a real-time OCL extension, named RT-OCL, is proposed and applied to demonstrate a process by which state configurations from UML diagrams are extracted to define additional predicates over states and state configurations. The RT-OCL temporal operators listed in Table 2 are useful for specifying the timing constraints on components at the specification level, where "@post ( )" returns a set of future execution sequences in an interval [a, b]; "@next ( )" is similar to "@post ( )", but returns a set of execution sequences after the next time step [1, 1]; and the "until" operator expresses causal dependencies between subsequent configurations. Since components in an embedded software system also depict a similar type of dynamic behavior, in this step, we apply RT-OCL to specify various types of timing constraints on events and event chains associated with components.

Table 2. OCL clauses and RT-OCL temporal operators.

OCL clauses	RT-OCL operators
self	@post ( )
let	@next ( )
if	until
forAll	
isUnique	
exist	
@pre	
iterate	

3.6. *Compute constraint complexity*

In the final step, the constraint complexity of events as well as the event chains of a component is computed with inputs from our constraint complexity measure. We have further divided this step into three stages. In the first stage, the occurrences of OCL clauses and RT-OCL temporal operators in pre- and post-condition sets, which contribute to the constraint complexity of each sub-function in an event chain, are counted. During the second stage, an extended constraint complexity measure is applied to compute the constraint complexity associated with events and event chains of a component. In the third stage, a uniform combination of the count and complexity of event chains is calculated using the average constraint complexity measure definition.

(1) Counting OCL clauses and RT-OCL temporal operators: To approximate the constraint complexity of a component, at this stage, we first count the

occurrences of each, among the group of eight, standard OCL clauses contributing to structural constraint complexity. Similarly, we also count the occurrences of each, among the group of three, RT-OCL temporal operators representing complexity due to timing constraints.

(2) Calculating constraint complexity: At this stage, a measure, capable of estimating both structural and timing constraints, is applied to compute the constraint complexity of components. For this purpose, we have extended an existing constraint complexity measure proposed in [5], where a UML CBSS structural constraint complexity measure, based on McCabe's cyclomatic complexity $V(G)$ [23], is defined. According to this definition, the complexity of each OCL clause (predicate) is one and the constraint complexity measure for each event is obtained by adding one to the number of predicates provided by UML/OCL. Since, like standard OCL clauses, RT-OCL temporal operators are also binary decision statements, we consider the complexity count of each occurrence of a standard OCL clause as well as the RT-OCL temporal operator as one.

The constraint complexity of an embedded software system's component x, denoted as $V(GC_x)$, is thus defined as

$$V(GC_x) = \sum_{i=1}^{p} \sum_{j=1}^{q} V(GE_{ij}), \tag{1}$$

where p is the number of event chains in x; q is the number of events in i event chain; and $V(GE_{ij})$ is the count of the occurrences of the predicates in event j of the event chain i. This new measure, $V(GC)$, is applied to components to compute their constraint complexity.

(3) Calculating average constraint complexity: To compute a uniform combination of the count and constraint complexity of events in an event chain of a component x, the average constraint complexity, represented by $A(GC_x)$, is defined as

$$A(GC_x) = \frac{V(GC_x)}{NE_x}, \tag{2}$$

where $V(GC_x)$ is the constraint complexity of an event chain of a component x; and NE_x is the combined number of events of all event chains in x. $A(GC)$ is applied to each component in an embedded software system in order to calculate their average constraint complexity.

4. An Application of Our Method

The application of the method is carried out on an automotive ABS. According to the models described in [18, 32, 35], sensors update brake pedal angle, the longitudinal speed of wheels, and vehicle speed to the ABS controller ECU. The ABS controller ECU performs two main tasks: first, it manages the slip of each wheel at a level which guarantees the highest braking performance and maneuverability of the car; and second, it continuously performs system diagnosis, while the ABS is active. The ABS controller sends manipulated signal values to the actuator to maintain the thresholds of the wheel brake pressure in order to avoid a wheel-lock danger. Under these assumptions, we have focused only on the basic features of an ABS to sufficiently address the constraints associated with workload events and the event chains of its components, while preserving the functional semantics.

4.1. *Develop a functional profile*

We used the RE-UML design tool [13] to describe the functional specifications of an automotive ABS, including the modeling of interdependencies among components and the functional environment. Figure 1 depicts the functional profile of an ABS. The ABS controller ECU consists of five components, namely "*Wheel Slip*", "*Regulate Wheel Slip*", "*Self-Diagnosis*", "*Start-up Diagnosis*" and "*Store Error Code*". The "*Wheel Slip*" and "*Regulate Wheel Slip*" components are responsible for managing slip of the wheels, whereas the remaining three components perform diagnostic activities.

4.2. *Identify events and event chains*

The syntactic descriptions of the components in an ABS controller ECU are expressed via event chains, for example, the "*Wheel Slip Management*" event chain is associated with the "*Wheel Slip*" component. For illustration purposes, we focus on describing the "*Wheel Slip Management*" event chain as per the sequence in which events are triggered in an ABS controller ECU. It is important to note here that we have assumed that the signals are updated in 50 ms with a maximum allowable latency of 5 ms, merely to demonstrate the application of our method. However, in a real-life scenario, the timeframe for updating signals and the latency values is determined through stringent research, testing, and extensive consultation with experts who are intimately familiar with the system. About four key events have been identified for the "*Wheel Slip Management*" event chain:

- the "*Update Vehicle Speed*" event receives the current speed of the moving car from the "*Vehicle Speed*" sensor component;
- the "*Update Wheel Speed*" event receives the current speed of the wheels from the "*Wheel Speed*" sensor component, one at a time;
- the "*Calculate Wheel Slip*" event uses "*Wheel Speed*" and "*Vehicle Speed*" to determine the slip of the wheels;
- the "*Update Wheel Slip*" event updates the slip values of all four wheels to the "*Regulate Wheel Slip Management*" event chain.

WheelSlipManagement	
1	UpdateVehicleSpeed (in ID : VehicleID, out vehicle : VehicleSpeed) : Boolean
2	UpdateWheelSpeed x 4 (in ID : WheelID, out wheel : WheelSpeed) : Boolean
3	CalculateWheelSlip x 4 (in wheel : WheelSpeed, in vehicle : VehicleSpeed, out slip : WheelSlip)
4	UpdateWheelSlip x 4 (in wheel : WheelSpeed, in vehicle : VehicleSpeed, out slip : WheelSlip)

Fig. 4. Identified events for the "*Wheel Slip Management*" event chain.

In the "*Wheel Slip Management*" event chain, there are 13 individual events, as shown in Fig. 4.

4.3. *Define constraints on event chains*

Figure 5 depicts a UML-based information-type model [19] for specifying the possible states of the "*Wheel Slip Management*" and "*Regulate Wheel Slip Management*" event chains at the specification level. The constraints, in particular invariants, are associated by a dotted line with the corresponding event chain types. As an example, consider the "*Wheel Slip Management*" event chain, where the wheel speed should not be same as the vehicle speed and the speed values of the wheels and vehicle must be updated between 1 ms and 55 ms, including latency time.

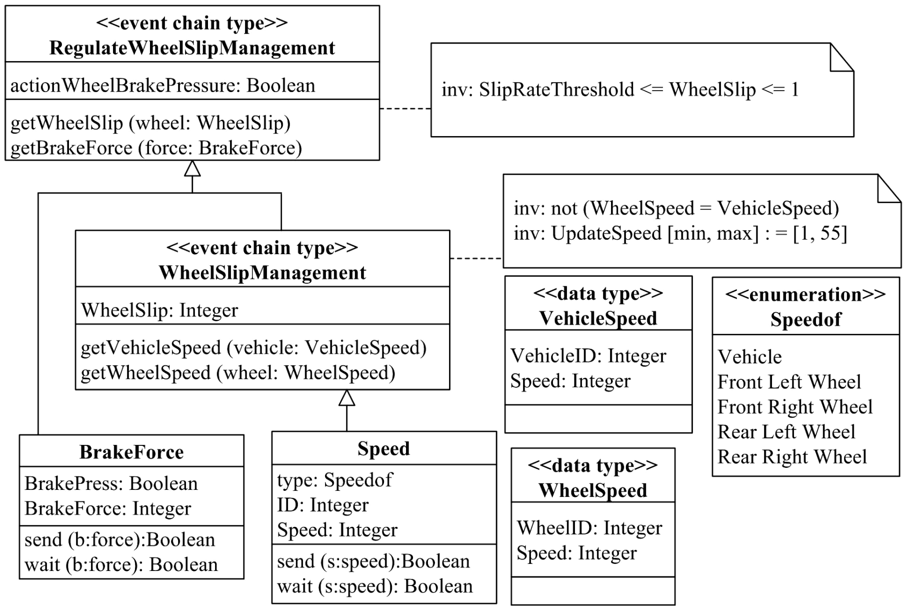

Fig. 5. Information-type model for the "*Wheel Slip Management*" and "*Regulate Wheel Slip Management*" event chains.

Once the static structure in terms of events and event chains is identified, the UML state charts are generated to provide a behavioral description of the system. Figure 6 shows the behavioral specification of the key events in the "*Wheel Slip Management*" event chain in terms of its responsibility to determine the actual slip of each wheel, and update the processed signals to the "*Regulate Wheel Slip Management*" event chain. Here, we have used text annotations to model behavior over time. In the "*Update Wheel Speed*" event, a time interval "update ( ) in [1, 55]" means that the speed has been received and updated within 1 ms to 55 ms. If, for any reason, a failure state is encountered, the "*Self-Diagnosis*" component notifies the error code and results in the termination of the process.

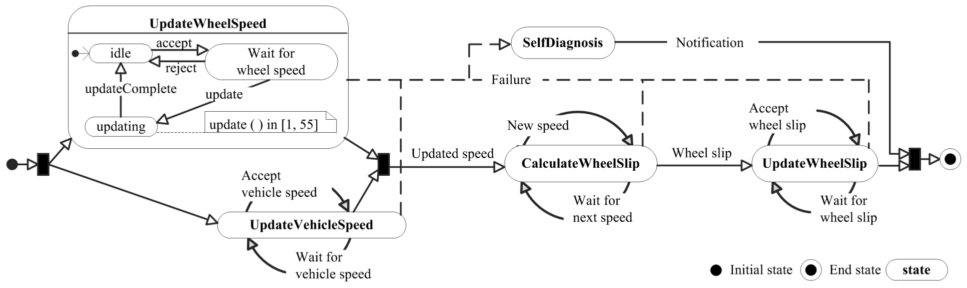

Fig. 6. State chart diagram for the "*Wheel Slip Management*" event chain.

4.4. *Specify structural constraints*

Each event of the "*Wheel Slip Management*" event chain has a set of OCL constraints associated with them. Figure 7 depicts the algorithm for specifying structural and timing constraints on the "*Calculate Wheel Slip*" event of the "*Wheel Slip Management*" event chain. For the "*Calculate Wheel Slip*" event, the pre-condition checks that the "*Vehicle Speed*" and longitudinal "*Wheel Speed*" signals are updated by the "*Update Vehicle Speed*" and the "*Update Wheel Speed*" events (refer to lines 3 and 4 in Fig. 7). This is followed by the execution of the post-condition which calculates the slip for the wheels. The invariants place different restrictions to support the execution of the post-condition, for example, an invariant checks that the wheel speed and vehicle speed values are greater than zero (refer to lines 10 and 11 in Fig. 7). Similarly, the structural constraints on each event in the event chains associated with the components in the ABS controller ECU are specified.

4.5. *Specify timing constraints*

In addition to the structural constraints, there are timing constraints on the events in the event chains. These constraints on the events of the "*Wheel Slip Management*"

```
1    WheelSlipManagement: calculateWheelSlip (in wheel: WheelSpeed, in vehicle: VehicleSpeed, out slip: WheelSlip)
2    def:    ABS::WheelSlip ⟶ isActive( ): Boolean
3    Pre:    wheel ⟶ exist (w | id = wheel.WheelID)                                              and
4            vehicle ⟶ exist (v | id = vehicle.VehicleID)                                        and
5    Post:   WheelSlip @pre ⟶ forAll (sr | sr.slipRef < > slip.slipRef)                          and
6    def:    Let    w = WheelSlip @pre ⟶ asSequence ⟶ first in
7                   w. slip. slipRef = slip.slipRef                                              and
8                   w. slip.id = slip.id                                                         and
9                   slip.id = WheelSpeed.WheelID                                                 and
10   inv:    WheelSpeed: integer ⟶ notEmpty( )                                                   and
11           VehicleSpeed: integer ⟶ notEmpty ( )
12           WheelSlip ⟶ forAll (WheelSpeed < > VehicleSpeed)                                    and
13   inv:    UpdatingWheelSpeedToWheelSlip.OclInState (UpdatingWheelSpeedToWheelSlip::updating)
14           implies
15           UpdatingWheelSpeedToWheelSlip @post (1, 5) ⟶ forAll (s: Sequence (OclState) |
             s⟶exist (u: OclState | u.isActive < > UpdatingWheelSpeedToWheelSlip::updating))
16   inv:    UpdatingVehicleSpeedToWheelSlip.OclInState (UpdatingVehicleSpeedToWheelSlip::updating)
17           implies
18           UpdatingVehicleSpeedToWheelSlip @post (1, 5) ⟶ forAll (s: Sequence (OclState) |
             s⟶exist (u: OclState | u.isActive < > UpdatingVehicleSpeedToWheelSlip::updating))
19   inv:    WheelSlip@post (1, 5) ⟶ exist (s: Sequence (OclState) | s⟶ includes
             (WheelSpeed::updating) until (51, 55) s⟶ includes (VehicleSpeed::updating)
20           Let      subObject: Integer = WheelSpeed ⟶ minus (VehicleSpeed)
21           In       subObject ⟶ notEmpty ( )                                                   and
22           WheelSlip@next (1, 1) = subObject.div (VehicleSpeed) * 100%
```

Fig. 7. Algorithm for specifying constraints on the "*Calculate Wheel Slip*" event of the "*Wheel Slip Management*" event chain.

event chain are specified using a combination of OCL and RT-OCL expressions. For example, the "*Wheel Speed*" and "*Vehicle Speed*" signals are updated to the "*Calculate Wheel Slip*" event with a maximum allowable latency of 5 ms (refer to lines 13–18 in Fig. 7). Following the signal receipt, another invariant of the "*Calculate Wheel Slip*" event checks that the signals are received within the requisite time intervals before calculating the slip of the wheels (refer to lines 19–22 in Fig. 7).

4.6. *Compute constraint complexity*

The constraint complexity of the events and event chains is estimated in three stages.

(1) Counting OCL clauses and RT-OCL temporal operators: In the "*Calculate Wheel Slip*" event of the "*Wheel Slip Management*" event chain, there are 13 occurrences of the OCL predicates ("exist", "let", "@pre", and "forAll"), and five occurrences of the RT-OCL predicates ("@post", "@next", and "until"), whereas for the "*Update Vehicle Speed*", "*Update Wheel Speed*", and "*Update Wheel Slip*" events in the "*Wheel Slip Management*" event chain, there are six occurrences of the OCL predicates ("let", "exist", "if", and "forAll"), and two occurrences of the RT-OCL predicate ("@post"). It is to be noted here

that there are four occurrences of the "*Update Wheel Speed*", "*Calculate Wheel Slip*", and "*Update Wheel Slip*" events, corresponding to the number of wheels on a car.

(2) Calculating constraint complexity: To obtain $V(GE)$ of an event as per the definition of the constraint complexity measure, McCabe's constant i.e. one is added to the count of the occurrences of OCL clauses and RT-OCL temporal operators which contribute to the constraint complexity. In Table 3, we present the $V(GE)$ values, obtained using Eq. (1), for the key events of the "*Wheel Slip Management*" event chain. To obtain constraint complexity, $V(GC)$, of the "*Wheel Slip Management*" event chain, $V(GE)$ values of each event are then added.

Table 3. Constraint complexity of the "*Wheel Slip Management*" event chain.

Event	$V(GE)$
Update vehicle speed	9
Update wheel speed × 4	36
Calculate wheel slip × 4	76
Update wheel slip × 4	36
	$V(GC) = 157$

(3) Calculating average constraint complexity: Since there are 13 events in the "*Wheel Slip Management*" event chain of the "*Wheel Slip*" component, using Eq. (2), the average constraint complexity of the "*Wheel Slip*" component is ($A(GC) = 157/13 = 12.08$).

4.7. *Results and discussion*

Table 4 shows that the $A(GC)$ of components in the ABS controller ECU is obtained by averaging the total of $V(GC)$ values against the total events in the respective event chains. Further analysis of the results indicates the following:

(1) The "*Regulate Wheel Slip*" component is the most complex, based on the associated number of events and its constraint complexity, $V(GC)$.

Table 4. Constraint complexity of components in ABS controller ECU.

Component	Events	OCL + RT-OCL	$V(GC)$	$A(GC)$
Wheel slip	13	144	157	12.08
Regulate wheel slip	16	172	188	11.75
Self-diagnosis	5	47	52	10.40
Start-up diagnosis	1	3	4	4
Store error code	2	4	6	3

However, the average constraint complexity, $A(GC)$, of the "*Regulate Wheel Slip*" component is lower than the average constraint complexity of the "*Wheel Slip*" component. This shows that each event in the "*Regulate Wheel Slip*" component has fewer constraints than the "*Wheel Slip*" component.

(2) The "*Wheel Slip*" component has the highest average constraint complexity. This shows that a single operation in the component has the highest constraint complexity.

(3) Of the components responsible for the diagnosis of the ABS controller ECU, namely "*Self-Diagnosis*", "*Start-up Diagnosis*", and "*Store Error Code*", the "*Self-Diagnosis*" component has the highest constraint complexity value as it has more associated events than the other two and it is the only component of the three which has timing constraints.

In order to determine the actual impact on the constraint complexity of a component due to added timing dependencies, in Table 5, we present a comparison of Mahmood and Lai's measure [5] against the extended measure. Using Mahmood and Lai's measure, only the count of the OCL clauses is obtainable whereas our measure not only provides the count of the OCL clauses, but also the RT-OCL operators associated with timing constraints if any on the components in an embedded software system. The average values of the OCL clauses for the components in the ABS controller ECU show that there is a minor difference between the values of the "*Wheel Slip*", "*Regulate Wheel Slip*" and "*Self-Diagnosis*" components, thus making it difficult for analyst to confidently identify the most complex component of these three in terms of their constraint complexity. The difference in $A(GC)$ values of such components becomes apparent only when the average constraint complexity values of the RT-OCL operators is added. This shows that the extended measure is better than Mahmood and Lai's measure, not only in estimating the structural and timing constraint complexity of components early in the specification phase of embedded software system development, but also helps analysts make decisions in cases when near similar estimates for structural constraint complexity are obtained for these components.

Table 5. Mahmood and Lai's measure versus the extended measure.

		Mahmood and Lai [5]		Extended measure		$A(GC)$ = Average$_1$ + Average$_2$ + 1
Component	Events	OCL	Average$_1$*	RT-OCL	Average$_2$**	(McCabe's constant)
Wheel slip	13	106	8.15	38	2.93	12.08
Regulate wheel slip	16	130	8.13	42	2.62	11.75
Self-diagnosis	5	41	8.20	6	1.20	10.40
Start-up diagnosis	1	3	3	—	—	4
Store error code	2	4	2	—	—	3

Notes: *Average$_1$ = OCL clauses/Events; **Average$_2$ = RT-OCL operators/Events.

5. Conclusions and Future Work

In this paper, we have presented a method for measuring the structural and timing constraint complexity of components in automotive embedded software systems. Our method consists of six steps at the specification level in order to sufficiently address the system designers and analyst's role in the distributed development environment. In the first three steps, designers use AUTOSAR-based MARTE-UML and TADL to develop an embedded software system's functional profile, identify events and event chains associated with components and define the non-deterministic constraints on the event chains of the components. During the next two steps, analysts use UML-compliant OCL and RT-OCL to specify the structural and timing constraints on events and event chains. The method then employs an extended constraint complexity measure we have developed, as an alternative to the traditional UML CBSS measurement technique [5], to also include an estimate of the timing constraint complexity. The extended measure uses a count of the occurrences of certain OCL clauses and RT-OCL temporal operators, which contributes to structural constraint and timing constraint complexity, to compute the constraint complexity of components in an embedded software system.

These constraint complexity numbers together with the interface complexity numbers help analysts obtain a quantitative measure of the internal properties, i.e. syntactic and semantic properties, of embedded software system components. At an early development phase of an AUTOSAR-based system, the measures of the internal properties and interaction properties of components aid in the identification of those components which require more attention during the integration and testing exercises. We also applied the method to an automotive ABS and demonstrated its viability by measuring the constraint complexity of an ABS controller ECU. The application results are very much in line with the notion that the addition of timing constraints increases the complexity of components and gives an encouraging indication about the applicability of UML CBSS complexity measures on embedded software systems.

Since the automotive ABS is modeled as a periodic data flow system, our method demonstrates its usefulness in computing constraint complexity when there are only periodic occurrences of the repetition rate constraints on workload events. Thus, in the future, we will undertake an assessment of the effectiveness of the extended constraint complexity measure by estimating various types of repetition rate constraints associated with events and delay constraints on the event chains of components in an automotive assembly. This will also aid in defining an appropriate scaling mechanism to determine the constraint complexity levels of components in embedded software systems. Furthermore, we will conduct an empirical analysis to validate the degree of correctness of the extended constraint complexity measure against a wider spectrum of complex real-time applications with stringent end-to-end dependability constraints.

Acknowledgment

The financial support from AutoCRC for supporting this research work is hereby acknowledged by the authors.

References

1. AUTOSAR. Release 4.0.3, http://www.autosar.org/, last accessed September 2015.
2. Object Management Group (OMG), UML profile for MARTE: Modeling and analysis of real-time embedded systems, Version 1.0, OMG document number: formal/2009-11-02, 2009.
3. TIMMO Consortium, Deliverable D6: TADL: Timing Augmented Description Language version 2, www.timmo.org, ITEA2 project TIMMO, 2009.
4. M. Broy, I. H. Kruger, A. Pretschner and C. Salzmann, Engineering automotive software, *Proc. IEEE* **95**(2) (2007) 356–373.
5. S. Mahmood and R. Lai, A complexity measure for UML component-based system specification, *Softw. Pract. Exp.* **38** (2008) 117–134.
6. N. S. Gill and P. S. Grover, Few important considerations for deriving interface complexity metric for component-based systems, *Softw. Eng. Notes* **29**(2) (2004) 1–4.
7. J. Han and K. K. Ker, Ensuring compatible interactions within component-based software systems, in *Proc. 10th Asia–Pacific Software Engineering Conf.*, 2003, pp. 436–445.
8. H. Espinoza, K. Ritcher and S. Gerard, Evaluating MARTE in an industry-driven environment: TIMMO's challenges for AUTOSAR timing modeling, Design Automation and Test in Europe (DATE), MARTE Workshop, 2008.
9. V. L. Narasimhan and B. Hendradjaya, A new suite of metrics for the integration of software components, Technical report (The 1st International Workshop on Object Systems and Software Architectures), University of Adelaide, Australia, 2004.
10. J. Fredriksson and R. Land, Reusable component analysis for component-based embedded real-time systems, in *Proc. 29th Int. Conf. Information Technology Interfaces*, 2007, pp. 615–620.
11. M. D. Natale, Design and development of component-based embedded systems for automotive applications, in *Ada-Europe 2008*, LNCS Vol. 5026 (Springer, Berlin/Heidelberg, 2008), pp. 15–29.
12. M. Krause, O. Bringmann, A. Hergenhan, G. Tabanoglu and W. Rosentiel, Timing simulation of interconnected AUTOSAR software-components, in *Conf. Design, Automation and Test in Europe*, 2007, pp. 474–479.
13. S. Mahmood and R. Lai, RE-UML: A component-based system requirements analysis language, *Comput. J.* **56**(7) (2013) 901–922.
14. O. Scheickl, M. Rudorfer, C. Ainhauser, N. Feiertag and K. Richter, How timing interfaces in AUTOSAR can improve distributed development of real-time software, in *GI Jahrestagung*, 2008, pp. 662–667.
15. F. Stappert, J. Jonsson, J. Mottok and R. Johansson, A design framework for end-to-end timing constrained automotive applications, in *Proc. Embedded Real-Time Software and Systems*, 2010.
16. H. Blom, R. Johansson and H. Lönn, Annotation with timing constraints in the context of EAST-ADL2 and AUTOSAR — The timing augmented description language, in *Proc. Workshop on the Definition, Evaluation, and Exploitation of Modeling and Computing Standards for Real-Time Embedded Systems*, 2009, pp. 2–5.

17. R. Lai and M. Garg, A detailed study of software reliability models, *J. Softw.* **7**(6) (2012) 1296–1306.
18. F. Mallet, M. A. Peraldi-Frati and C. Andre, Marte CCSL to execute EAST-ADL timing requirements, in *Proc. Int. Symp. Object/component/service-oriented Real-Time Distributed Computing*, 2009, pp. 249–253.
19. S. Flake and W. Mueller, An OCL extension for real-time constraints, in *Proc. Object Modeling with the OCL*, 2002, pp. 150–171.
20. S. Flake and W. Mueller, Specification of real-time properties for UML models, in *Proc. 35th Annual Hawaii Int. Conf. System Sciences*, 2002, pp. 3977–3986.
21. M. Torngren, D. Chen and I. Crnkovic, Component-based vs. model-based development: A comparison in the context of vehicular embedded systems, in *Proc. 31st EUROMICRO Conf. Software Engineering and Advanced Applications*, 2005, pp. 432–440.
22. R. Racu, A. Hamann, R. Ernst and K. Ritcher, Automotive software integration, in *Proc. 44th Annual Design Automation Conference*, 2007, pp. 545–550.
23. T. J. McCabe, A complexity measure, *IEEE Trans. Softw. Eng.* **2**(4) (1976) 308–320.
24. C. Szyperski, D. Gruntz and S. Murer, *Component Software — Beyond Object-Oriented Programming*, 2nd edn. (Addison-Wesley, Reading, 2002).
25. Object Management Group (OMG), Object Constraint Language (OCL), version 2.2, OMG document number: formal/2010-02-01, 2010, http://www.omg.org/spec/OCL/2.2/.
26. J. Willander, J. Eliasson, A. Kvruglyak, P. Lindgren and J. Nordlander, Enabling component-based design for embedded real-time software, *J. Comput.* **4**(12) (2009) 1309–1321.
27. A. Kleppe and J. Warmer, Extending OCL to include actions, in *≪UML≫ 2000 — The Unified Modeling Language*, eds. A. Evans, S. Kent and B. Selic (Springer-Verlag, Berlin, 2000), pp. 440–450.
28. S. Neema, J. Sztipanovits, G. Karsai and K. Butts, Constraint-based design-space exploration and model synthesis, in *Int. Workshop on Embedded Software*, 2003, pp. 290–305.
29. T. Wijayasiriwardhane and R. Lai, Component point: A system-level size measure for component-based software systems, *J. Syst. Softw.* **83** (2010) 2456–2470.
30. O. Scheickl, C. Ainhauser and M. Rudorfer, Distributed development of automotive real-time systems based on function-triggered timing constraints, in *Proc. Embedded Real-Time Software*, 2010, pp. 1–6.
31. G. N. Vo, R. Lai and M. Garg, Building automotive software component within the AUTOSAR environment — A case study, in *Proc. 9th Int. Conf. Quality Software*, 2009, pp. 191–200.
32. S. B. Choi, Antilock brake system with a continuous wheel slip control to maximize the braking performance and the ride quality, *IEEE Trans. Control Syst. Technol.* **16**(50) (2008) 996–1003.
33. K. Richter and R. Ernst, Event model interfaces for heterogeneous system analysis, in *Proc. Conf. Design, Automation and Test in Europe*, 2002, p. 506.
34. S. Sedigh-Ali, A. Ghafoor and R. A. Paul, A metrics-guided framework for cost and quality management of component-based software, in *Component-Based Software Quality*, Vol. 2693, eds. A. Cechich, M. Piattini and A. Vallecillo (Springer, Berlin, 2003), pp. 374–402.
35. H. Y. Kim, K. Jerath and F. Sheldon, Assessment of high integrity software components for completeness, consistency, fault-tolerance, and reliability, in *Component-based Software Quality*, eds. A. Cechich *et al.* (Springer-Verlag, Berlin, 2003), pp. 259–286.

36. D. Durisic, M. Nilsson, S. Staron and J. Hansson, Measuring the impact of changes to the complexity and coupling properties of automotive software systems, *J. Syst. Softw.* **86** (2013) 1275–1293.
37. M. Garg and R. Lai, Measuring the constraint complexity of automotive embedded software systems, in *Proc. Int. Conf. Data and Software Engineering*, 2014, pp. 1–6.
38. E. S. Cho, M. S. Kim and S. D. Kim, Component metrics to measure component quality, in *Proc. 8th Asia–Pacific Software Engineering Conf.* 2001, pp. 419–426.
39. J. K. Chhabra and V. Gupta, A survey of dynamic software metrics, *J. Comput. Sci. Technol.* **25**(5) (2010) 1016–1029.