Improving the Model-Based Systems Engineering Process

Michael von Wenckstern

Abstract Modern embedded software systems are becoming more and more complex. Engineering embedded systems raise specific challenges that are rarely present in other software engineering disciplines due to the systems' steady interactions with their environment. Research and industry often describe embedded systems as component and connector models (C&C). C&C models describe the logical architecture by focusing on software features and their logical communications. In C&C models, hierarchical decomposed components encapsulate features, and connectors model the data flow between components via typed ports. As extra-functional properties, for example, safety and security, are also key features of embedded systems, C&C models are mostly enriched with them. However, the process to develop, understand, validate, and maintain large C&C models for complex embedded software is onerous, time consuming, and cost intensive. Hence, the aim of this chapter is to support the automotive software engineer with: (i) automatic consistency checks of large C&C models, (ii) automatic verification of C&C models against design decisions, (iii) tracing and navigating between design and implementation models, (iv) finding structural inconsistencies during model evolution, (v) presenting a flexible approach to define different extra-functional properties for C&C models, and (vi) providing a framework to formalize constraints on C&C models for extra-functional properties for automatic consistency checks.

1 Introduction

The industry area of embedded and cyber-physical systems is one of the largest and it influences our daily life. The global embedded systems marked are getting up to 225 billion US dollar by end of 2021 [24]. Example domains of embedded systems

M. von Wenckstern (✉)
Software Engineering, RWTH Aachen University, Aachen, Germany
e-mail: vonwenckstern@se-rwth.de

M. Felderer et al. (eds.), *Ernst Denert Award for Software Engineering 2020*,
https://doi.org/10.1007/978-3-030-83128-8_12

are automotive, avionics, robotics, railway, production industry, telecommunication, consumer electronics, and much more.

Model-based engineering, especially component and connector (C&C) models to describe logical architectures, is one common approach to handle the large complexity of embedded systems. Components encapsulate software features; the hierarchical decomposition of components enables formulating logical architectures in a top-down approach. Connectors in C&C models describe the information exchange via typed ports; they model black-box communication between software features.

The current development of complex C&C-based embedded systems in industry mostly involves the following steps [1, 6]: (1) formulating functional and extra-functional requirements as text in *IBM Rational DOORS*; (2) creating a design model of the software architecture including its environment interactions in *SysML*; (3) developing a complete functional/logical model to simulate the embedded system in *Simulink*; and (4) system implementation based on available hardware in C/C++ satisfying all extra-functional properties.

This current development process has the following disadvantages [9]: (a) *SysML* models do not follow a formalized approach leading to misunderstandings; (b) the check between the informal *SysML* architecture design and the *Simulink* model is done manually, and thus, error prone and time consuming; (c) refactoring of *Simulink* models (e.g., dividing a subsystem) needs manual effort in updating the design model; and (d) most tools do not support a generic approach for different extra-functional property kinds, and thus, these properties are modeled as comments or stereotypes where consistency checks can only be done manually.

My research [23] aims to improve the development process of large and complex C&C models for embedded systems by providing model-based methodologies to develop, understand, validate, and maintain these C&C models. Concretely, my concepts support the embedded software engineer with: (i) automatic consistency checks of C&C models; (ii) automatic verification of logical C&C models against their design decisions; (iii) automatic addition of traceability links between design and implementation models; (iv) finding structural inconsistencies during model evolution; (v) providing a flexible framework to define different extra-functional property types; (vi) presenting an *OCL* framework to specify (company-specific) constraints about structural or extra-functional properties for C&C models; and (vii) generation of positive or negative witnesses to explain why a C&C model satisfies or violates its extra-functional or structural constraints or its design decisions.

Prototype implementations of above-mentioned concepts and an industrial case study in cooperation with Daimler AG show promising results in improving the model-based development process of embedded and cyber-physical systems in industry.

This chapter is a summary of the PhD thesis *Verification of Structural and Extra Functional Properties in Component and Connector Models for Embedded and Cyber Physical Systems* [23]. This chapter focuses on examples how the development process for the left part of the V-model can be improved. Detailed related work analyses comparing different tools and concepts like *Mentor Capital*, *Polarsys Arcadia*, *PREEvision* [23, Section 2.3], as well as related C&C (modeling) languages such as *AADL*, *ACME*, *Ada*, *AutoFOCUS*, *AUTOSAR*, *LabView*, *MARTE*, *Modelica*, *SysML*, *SystemC*, *Verilog*, and more [23, Section 3.2 and Section 5.2] are discussed in the PhD thesis.

2 Systems Engineering Process at Daimler AG

This section presents results of our case study with Daimler AG in 2017 [1]. The case study included several interviews with an employee at Daimler AG to understand the current model-based and component-based development process as well as the challenges engineers are facing.

2.1 Current Development Process at Daimler AG

Both ISO 26262 *Road vehicles—Functional safety* and ISO/SAE 21434 *Road vehicles—Cybersecurity engineering* follow the V-model and are the international standards for automotive industry. The steps on the left side of the V-model from top to bottom are *system design*, *specification of software safety/security requirements*, *software architectural design*, and *software unit design and implementation*; the steps on the right side from bottom to top are *software unit testing*, *software integration testing*, *verification of software safety/security requirements*, and *item integration and testing* [3].

The design of a system is mostly described as textual requirements with links to each other; one famous requirement management tool is *IBM Rational DOORS* (short *DOORS*). Later extra-functional requirements for safety or security of a system's design are identified; examples for safety (security process is based on the safety one) are functional safety concept, technical safety concept, system safety, and hardware failures. These extra-functional and stakeholder requirements are integrated into existing requirements of a system's design.

The design of a software architecture is mostly modeled in *SysML* block diagram definitions. Common *SysML* tools in industry are *Enterprise Architect*, *ArchiMate*, *Metropolis*, *Cameo Systems Modeler*, and *PTC Integrity Modeler*. The requirements are modeled separately in these tools and are linked to the corresponding modeling elements, so that traceability is always given [19].

After the design (high-level interaction between components themselves and their environment) is modeled in *SysML*, engineers at Daimler AG create manually an executable model in *Simulink* regarding to the previously defined design decisions. To have the traceability between requirements, *SysML* design models, and *Simulink* implementation models, engineers at Daimler AG add to every subsystem in *SysML* and in *Simulink* an information block containing a link to the requirement specification in *DOORS* [1]. Adding and maintaining these links manually is time consuming and error prone.

This development process has the following disadvantages:

- The check between the informal *SysML* architecture design and the *Simulink* model is done manually.
- The requirement links must be created manually for architectural design model and for the *Simulink* model.
- There exists no automatic check in finding outdated *Simulink* subsystems after updating *SysML* design models (e.g., due to model evolution).
- If *Simulink* models are refactored (e.g., subsystem is split into several ones), it may occur that the *SysML* design model is not updated, and then the architecture model becomes obsolete.
- Early inconsistencies in the *SysML* software architecture design, created by different persons or even different teams in large companies, must be detected manually.

2.2 Improving the Development Process at Daimler AG

To mitigate most of these above-mentioned disadvantages, this subsection presents a slightly modified development process and verification tools, as shown in Fig. 1. The advantage of this new process is that it is completely compatible to existing tools (cf. right side of Fig. 1). The general workflow of this new process including existing tools is:

1. *DOORS* requirements are automatically extracted to a set of textual requirements.
2. Engineers create manually for each requirement a C&C high-level design model.
3. C&C design models are automatically transferred to graphical *SysML* diagrams.
4. The links between *DOORS* requirement IDs and C&C design models enable to automatically derive tracing information between *DOORS* and *SysML* diagrams.
5. Synthesis algorithms automatically check against structural inconsistencies in high-level C&C design models.
6. Engineers add manually extra-functional properties to the C&C high-level design model based on the textual requirements.
7. The *OCL* (Object Constraint Language) framework checks automatically the consistence of the added extra-functional requirements of the high-level design.
8. Engineers create manually the functional C&C model based on textual requirements and the C&C high-level design models.
9. Verification automatically checks whether the functional C&C model satisfies all C&C high-level design models.

10. The functional C&C model is automatically transformed to a *Simulink* model.
11. The result of step 9 enables to automatically derive tracing information between all *SysML* diagrams and the one *Simulink* model as well as tracing information between all *DOORS* requirements and the one *Simulink* model.[1]
12. The *Simulink* model is executed. Measured runtime information (e.g., timing) automatically enriches the C&C model via extra-functional properties.
13. Engineers enrich manually the C&C model with extra-functional properties based on user manuals of software or hardware components, for example, price or ASIL.
14. The *OCL* framework checks automatically the consistency of the extra-functional properties added in steps 12 and 13 to the functional C&C model.
15. The *OCL* framework in combination with the verification in step 9 validates automatically whether all extra-functional properties in the functional C&C model satisfy all extra-functional requirements in all C&C design models.

Even though the new toolchain is larger, there are less manual steps needed due to the higher automation of the steps in this new toolchain. Creating *SysML* diagrams

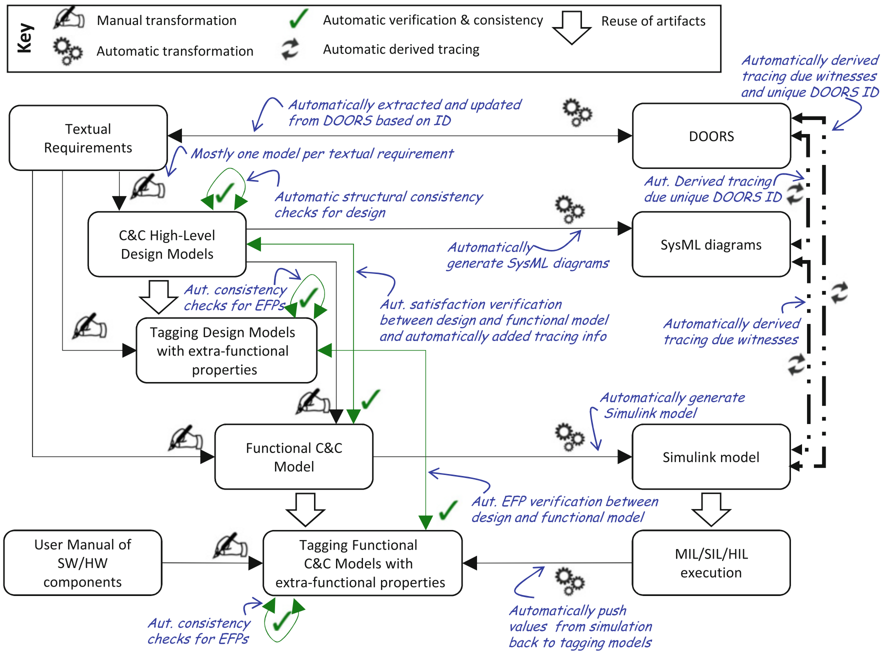

Key
Manual transformation
Automatic transformation
Automatic verification & consistency
Automatic derived tracing
Reuse of artifacts
Textual Requirements
DOORS
Automatically extracted and updated from DOORS based on ID
Automatically derived tracing due witnesses and unique DOORS ID
Mostly one model per textual requirement
Automatic structural consistency checks for design
Aut. Derived tracing due unique DOORS ID
C&C High-Level Design Models
SysML diagrams
Automatically generate SysML diagrams
Aut. consistency checks for EFPs
Aut. satisfaction verification between design and functional model and automatically added tracing info
Automatically derived tracing due witnesses
Tagging Design Models with extra-functional properties
Automatically generate Simulink model
Functional C&C Model
Simulink model
Aut. EFP verification between design and functional model
User Manual of SW/HW components
Tagging Functional C&C Models with extra-functional properties
MIL/SIL/HIL execution
Automatically push values from simulation back to tagging models
Aut. consistency checks for EFPs

Fig. 1 Modified development process, compatible to V-model (only left side of V-model is shown here)

[1] Due to existing tracing information between *SysML* and *DOORS* due to steps 2 and 3.

based on textual requirements needs one manual step in the existing approach: *DOORS* $\rightarrow$[2] *SysML* diagrams. The improved toolchain also needs only one manual step to translate textual requirements to C&C high-level design models as shown in Fig. 1: *DOORS* $\Rightarrow$ Textual Requirements $\rightarrow$ C&C High-Level Design Models $\Rightarrow$ *SysML* diagrams. The same holds to create *Simulink* models based on *DOORS* requirements and *SysML* diagrams, where the additional manual step in the existing approach is to create *Simulink* models manually, whereas in the new toolchain the functional C&C models are created manually: *DOORS* $\rightarrow$ *SysML* diagrams $\rightarrow$ *Simulink* model $\equiv$ *DOORS* $\Rightarrow$ Textual Requirements $\rightarrow$ C&C High-Level Design Models $\rightarrow$ Functional C&C Model $\Rightarrow$ *Simulink* model.

In the existing approach, the tracing between *DOORS* and *SysML* diagrams, between *DOORS* and *Simulink* model, as well as between *SysML* diagrams and *Simulink* model is done manually. In contrast, **the new toolchain does the tracing between C&C high-level design models and functional C&C model automatically**. Thus, only the tracing between textual requirements and C&C high-level design models is done implicitly manually as each C&C design model belongs to one requirement. Based on this implicit relation between textual requirements and C&C high-level design models as well as the automatically generated tracing between C&C high-level design models and functional C&C model, the tracing for textual requirements and functional C&C model can also be done automatically. The two automatic transformations enable to automatically derive the tracing between *DOORS* requirements and the *Simulink* model. This means three manual tracing relations in the old approach are equivalent to only one manual tracing relation in the new toolchain. **Thus, the new toolchain saves a lot of work, especially in agile systems engineering, and it prevents manual tracing errors.**

Furthermore, **the new toolchain adds** due to its unique semantics **many additional automatic verifications** to ensure better model quality and **to prevent modeling errors as early as possible**: steps 5, 7, 9, 14, and 15.

C/C++ compiler/linker toolchains create one executable file based on many C/C++ source code text files. In a similar way, the EmbeddedMontiArc toolchain creates one C&C high-level design based on multiple textual text input files. The combined C&C high-level design can be graphically displayed and/or logged into one "merged" larger text file. The advantage of splitting up the design decisions into several textual files (similar as programming languages do it) is the ability to version and merge changes in these files separately. Commercial *SysML* tools such as *PTC Integrity Modeler* (short *PTC IM*) use a database approach, which supports to version only the entire (design) model including all *SysML* elements used by different development teams. In *PTC IM* different teams work in one database model, as otherwise (tracing) links between elements—created in different layers or by different teams—are not possible. In contrast to the database linking approach, the presented textual C&C modeling language family (cf. Sect. 4) to specify C&C high-level designs as well as functional C&C models uses readable full qualified

[2] $\Rightarrow$: automatic transformation; $\rightarrow$: manual transformation.

names (instead of generated encrypted IDs by *PTC IM*) to establish the linking process.

The synthesis algorithm enables to check the C&C high-level design against inconsistencies [10, 12]. If this algorithm generates a functional C&C model based on the specified high-level design, then the design is consistent; otherwise the specified design is inconsistent. For inconsistent designs, the synthesis algorithm generates user-friendly error messages, which include a natural text of the problem description, and a minimal C&C witness containing the involved components causing the conflict. Since these checks are completely automatic, they can be integrated in a commit-based or nightly continuous integration process. These algorithms are described by Maoz and Ringert [15].

The high-level design can be enriched with extra-functional properties such as safety, performance, or security ones. The strong typed tagging mechanism allows to tag only correct elements which reduces human errors (e.g., shifting a line lower). An example of a check for the tagging mechanism is unit correctness: A velocity tag of a car cannot be `9 kg`. Since for each extra-functional property consistency constraints can be defined, the validation framework (cf. Sect. 7) can check full-automatically (no further user action is required) the correctness of the design model with its enriched extra-functional properties. For example, the tool can check whether the price of a component is larger than the sum of the prices of its subcomponents.

EmbeddedMontiArc (cf. Sect. 6) is a textual modeling language extending *Simulink* with new features such as complete unit support as well as component and port arrays. These extensions facilitate an easier description of functional C&C models: (1) Model references must not be copied to be used multiple times, and (2) stronger types with units prevent inconsistencies when connecting ports. Additionally, our textual approach is based on the modular Java class concept that supports to split one model into several textual files to be modified and versioned by different teams.

Furthermore, the layout algorithm [23, Subsection 8.5.1.] creates nice graphical representations with boxes and lines of the textual model. These graphical representations enable an easier navigation between different components. Furthermore, the layout algorithm avoids manually (and time-consuming) adaptions of the graphical model when adding new ports.[3] Based on the automatically calculated layout of the textual model, a *MATLAB* script file containing *Simulink* API calls with x and y coordinates of subsystems creates the *Simulink* model. Hence, the here presented workflow can be easily integrated into the existing workflow of Daimler AG being based on *SysML* and *Simulink* tools.

Additionally, my PhD thesis [23, Section 7.4f] also defines formally when a functional model satisfies all its design models. If the design verification was successful, then the tooling infers automatically all tracing information/links. In

[3] *Simulink* does not have a layout algorithm, yet [7]. But other modeling tools such as *Ptolemy* II [4] and *LabView* [18] have one.

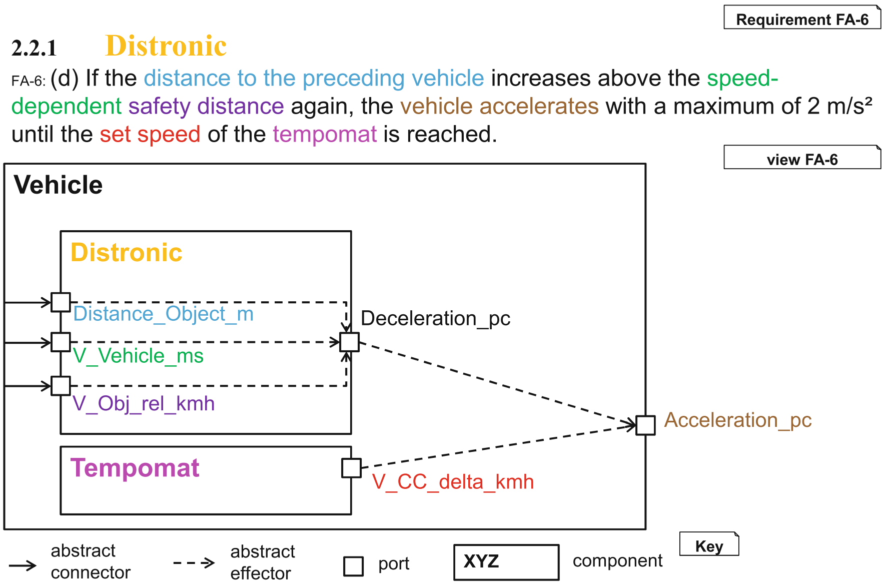
case the functional model does not satisfy the design model, then non-satisfaction witnesses with user-friendly error messages pointing directly to the error locations are generated.

Besides the case study focusing on structural consistency checks which is explained in this chapter, there exist also case studies with Daimler AG [21], BMW Group [9], and FEV GmbH [20] with focus on behavioral parts of C&C and *Simulink* models. All these case studies in the automotive domain helped to understand the model-based systems engineering process in detail and how to improve it.

3 Creating C&C High-Level Designs Based on Requirements

In step 2 in Sect. 2.2 the engineer creates for each textual requirement, a C&C high-level design model. The top part of Fig. 2 shows one requirement of the ADAS (advanced driver assistant system) requirement `FA-6` based on our case study with Daimler AG. The prefix `FA` is an abbreviation of `Fahrerassistenzsystem` which is the German word for ADAS. The requirement `FA-6` is part of the functions describing the `Distronic` feature.

The bottom part of Fig. 2 shows the manual created C&C view. The word C&C view is used as synonym for C&C high-level design in this chapter. This view views


Requirement FA-6

2.2.1 Distronic

FA-6: (d) If the distance to the preceding vehicle increases above the speed-dependent safety distance again, the vehicle accelerates with a maximum of 2 m/s² until the set speed of the tempomat is reached.

Fig. 2 Requirement FA-6 of unit `Distronic` of ADASv4 (top) and the view created for this requirement by the domain experts (bottom); copied from [1, Fig. 5].

the high-level architectural design model for this specific requirement, for example, how the dataflow between different C&C model should be. The colors in the text and in the C&C view illustrate the mapping between both. The names in the if condition phrase are mapped to input ports, as the `Distronic` component needs to read these values to produce the correct reaction.

The solid arrows in Fig. 2 represent abstract connectors. The left top abstract connector going from `Vehicle` to the `Distance_Object_m` abstract port of the `Distronic` components states that the `Vehicle` component has an input port which delegates its value without modifying it to an input port of the `Distronic` subsystem having the signal name `Distance_Object_m`. As the view shows only an abstraction (one viewpoint), the hierarchy between `Vehicle` and `Distronic` is not direct; thus, in the *Functional C&C Model* the vehicle may have the subcomponent `AdaptiveCruiseControl`, and `AdaptiveCruiseControl` has as one subcomponent `Distronic`.

The dashed arrows in Fig. 2 represent abstract effectors. The top right abstract effector going from the abstract port `Deceleration_pc` of the `Distronic` component to `Acceleration_pc` of the `Vehicle` component states that the output port with the signal name `Deceleration_pc` of the `Distronic` component influences the value of the output port with the signal name `Acceleration_pc` of the `Vehicle` component. Influence means that value of `Deceleration_pc` may be modified by other components.

The abstract port `Deceleration_pc` is not mentioned in the FA-6 requirement. However, the domain experts[4] included this abstract port in the C&C view as the deceleration value (100% deceleration means the car is not accelerating at all, 0% deceleration means that the car accelerates with its maximal acceleration) is a limiting factor of the vehicle's acceleration, and the domain experts meant that this port is crucial to understand the implementation of this requirement.

4 Automatic Structural Consistency Checks for Design Models

An advantage of the improved development process is the automatic consistency check in step 5 between two C&C high-level design models.

Figure 3 shows two further created C&C design models by two different engineers. The synthesis algorithm [16] throws an exception as it cannot generate a valid functional C&C model based on the given three views: `view FA-6` in Fig. 2 as well as `view FA-31` and `view FA-32` in Fig. 3. The exception message would be similar to *Design conflict between `view FA-31` and `view FA-32`: The*

[4] The word domain expert in this chapter refers to the domain expert (an employee at Daimler AG in 2017) who created the C&C high-level designs based on the textual requirements in the industrial case study together with Daimler AG.

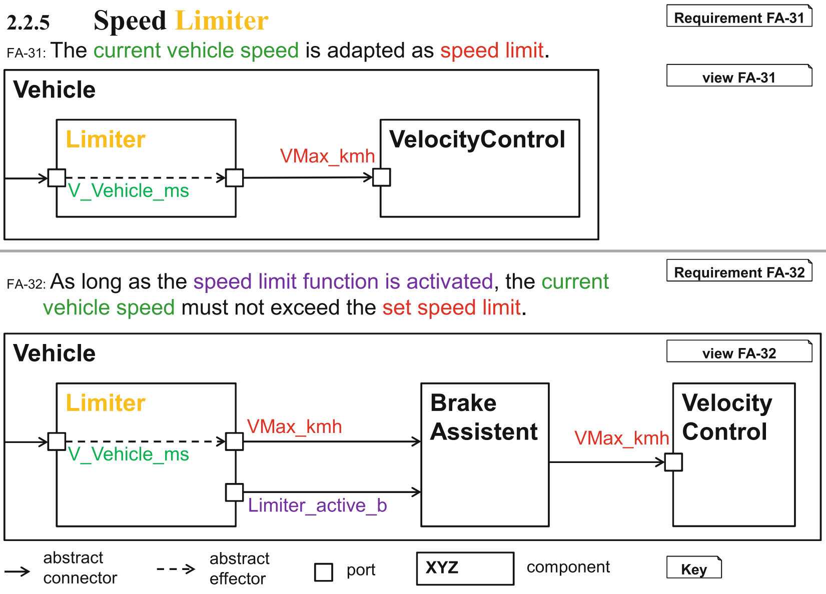


Fig. 3 Conflict in C&C high-level design models `view FA-31` and `view FA-32`

port `VMax_kmh` *of* `VelocityControl` *component can only progress a single value for each time step. However it receives two different values: (1) one value from the* `Limiter` *component in* `view FA-31` *and (2) another value from the* `BrakeAssistent` *component in* `view FA-32`*.*

These precise exception messages of the synthesis algorithm help the architects who are creating the C&C high-level design models to resolve inconsistencies in the architecture while they are creating their view design models for each requirement.

5 Satisfaction Verification Between Design and Functional Model

In step 8 in Sect. 2.2 the engineer creates the functional C&C model based on all textual requirements and based on all C&C high-level design models. In our case study with Daimler AG, the ADAS part contained of 68 requirements [1, Table 1] and the corresponding C&C model had over 1 000 C&C components and over 3 500 C&C ports [23, Table 8.17]. For such a large functional model it is very time consuming to verify manually whether it satisfies the previously in step 2 defined architecture containing of many C&C high-level design models. Therefore, the Software Engineering chair including the author of this chapter developed a verification tool to automatically check the satisfaction verification between all C&C design models and the one C&C functional model. To increase the trust of the result provided by the verification tool, the tool generates for each C&C design model one satisfaction witness illustrating why the functional model satisfies this C&C design model.

Figure 4 presents the generated satisfaction witness reasoning why the C&C functional model[5] satisfies the C&C design model in Fig. 2. The verification algorithm only creates textual output of witnesses; the layout tool, which also transforms the textual C&C functional models to graphical *Simulink* models in step 10, generates good understandable graphical witnesses such as the one shown in Fig. 4.[6]

The blue highlighted connectors in the bottom left part of Fig. 4 belong to the connector chain of the witness representing the abstract connector going from `Vehicle` (unknown port) to `Distronic`'s `V_Obj_rel_kmh` port in the C&C high-level design. Additionally, Fig. 4 highlights the witness elements (i.e., upper colored atomic blocks and signal lines in the C&C functional model) belonging to the abstract effector starting at the `Distance_Object_m` port and ending at `Deceleration_pc` port of the `Distronic` subsystem.

Figure 4 shows all elements of the generated satisfaction witness, that is, it contains all components, ports, and connectors so that all elements of the C&C high-level design in Fig. 2 are matched at least once. Note that the satisfaction witness shows for each abstract connector and abstract effector only the shortest path in the C&C functional model. In contrast, the tracing witness contains ALL paths of the C&C functional model which match any element described in the C&C high-level design. The verification witness is used as argument to reason why or also why not a functional model satisfies a specific design model.

The purpose of a tracing witness[7] is to trace down to all components, ports and connectors in the functional model based on a design model. A practical use case for the tracing witness is the following: Due to a product update some requirements and, thus also, some high-level design models are updated. The generated tracing witness identifies all elements in the functional C&C model which are affected by the design and requirement updates. The size and complexity of the generated tracing witness enables a first effort and price estimation for the product update. Furthermore, the

[5] The complete graphical C&C model is available from:https://embeddedmontiarc.github.io/webspace2/svg/vis/v4/daimler.v4.oeffentlicher_Demonstrator_FAS_v04.dEMO_FAS.dEMO_FAS.subsystem.dEMO_FAS.dEMO_FAS_Funktion_extended.html.

[6] The generated graphical output of the layout tool is available from https://embeddedmontiarc.github.io/webspace2/svg/vis/v4/FA6_Witness/daimler.v4.oeffentlicher_Demonstrator_FAS_v04FA6.dEMO_FAS.dEMO_FAS.subsystem.dEMO_FAS.dEMO_FAS_Funktion_extended.html. Figure 4 shows a manually and slightly modified layout which is space and color optimized for this chapter.

[7] The generated layout of the tracing witness for requirement `FA-6` is available from https://embeddedmontiarc.github.io/webspace2/svg/vis/v4/FA6_TracingWitness/daimler.v4.oeffentlicher_Demonstrator_FAS_v04FA6Tracing.dEMO_FAS.dEMO_FAS.subsystem.dEMO_FAS.dEMO_FAS_Funktion_extended.html.

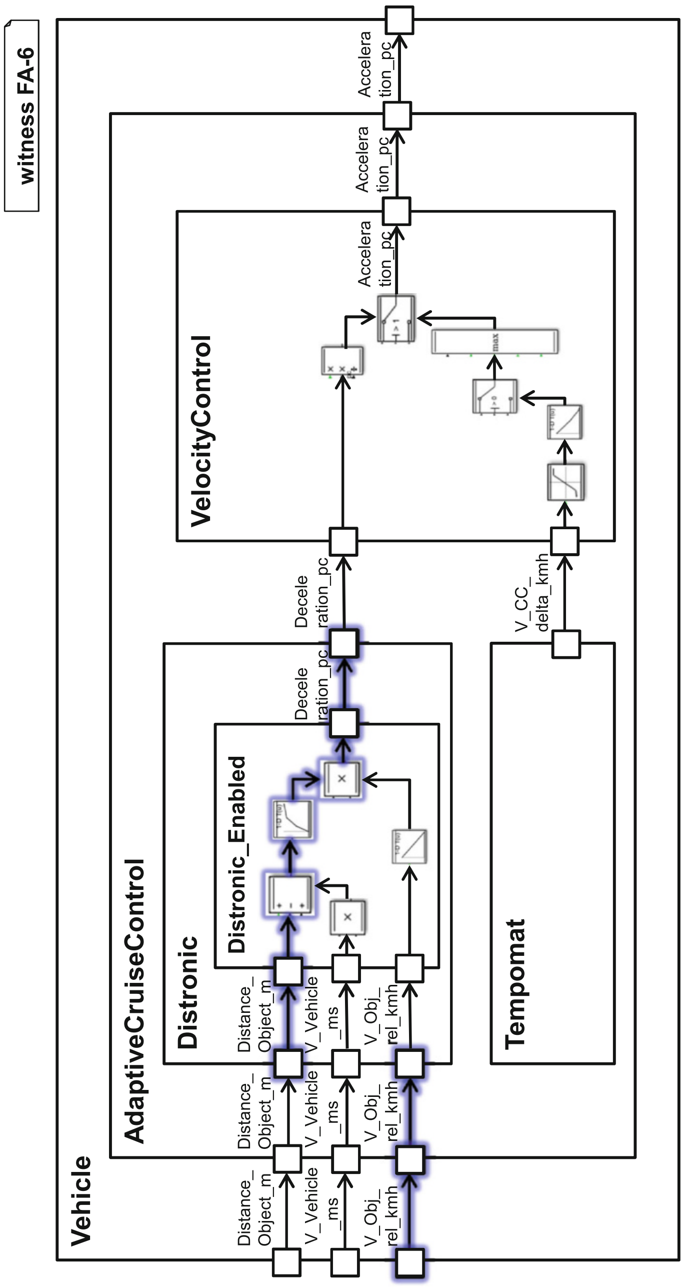


Fig. 4 Satisfaction witness of view FA-6

components inside the generated tracing witness supports management in booking the needed teams for this product update.[8]

6 Creating C&C Functional Models Efficiently with *EmbeddedMontiArc*

EmbeddedMontiArc is a textual domain-specific language to create functional C&C models for cyber-physical systems in an efficient way. Therefore, *EmbeddedMontiArc* supports the international systems of units, generics, component libraries, configuration parameters, as well as arrays of ports and component instantiations to facilitate modular and reusable functional architectures.

EmbeddedMontiArc is a textual modeling family to describe both structural (as already shown in the previous sections) and also behavior models. Behavioral languages part of *EmbeddedMontiArc* family are: automata, *MontiMath* (typed version of *MATLAB*), *MontiMathOpt* (math plus nonlinear optimization problems), *CNNArch* (convolutional networks for deep learning), and *OCL* (object constraint language for logical declarative description of components). This chapter does not focus on the behavioral languages. However, the publications of Kusmenko and von Wenckstern [8, 9, 11–13] contain behavioral models describing the logic of self-driving cars created with *EmbeddedMontiArc*. A complete behavioral *EmbeddedMontiArc* model for the logic of PacMan to escape four ghosts including a simulator and a debugger exists as online demonstrator.[9]

The functional C&C model in Fig. 4 shows the OEM perspective, where the `Vehicle` component receives the distance to the preceding vehicle as number input directly from a smart sensor doing already the image capturing as well as the object recognition. The smart sensor (developed by an automotive tier-1 supplier for different OEMs) receives as input an image matrix of the front car camera, performs spectral clustering to divide the images into segments, the object detector can separate the objects to identify the car in front and to measure the distance to it.

This section explains the features and the syntax of *EmbeddedMontiArc* on an image spectral cluster component [14] which could be part of the smart sensor.

The basic idea is depicted as a functional C&C model in Fig. 5 and can be summarized as follows. Let $x_{ij} \in [0, 255]^3$ be the three-dimensional pixel value of an image at position (i, j) encoding a point in the HSV (hue, saturation, value) color space. For better handling, an $N \times M$ image is represented as a vector, mapping a position (i, j) to the vector index $M \cdot i + j$, where N and M are

[8] Mostly subcomponents can be matched to teams; for example, team one is responsible for `Tempomat` and `VelocityControl`, and team two works on `Distronic`, `Distancewarner`, and `EmergencyBrake`.

[9] The online demonstrator is available from: https://embeddedmontiarc.github.io/webspace/InteractiveSimulator/indexPacman.html.

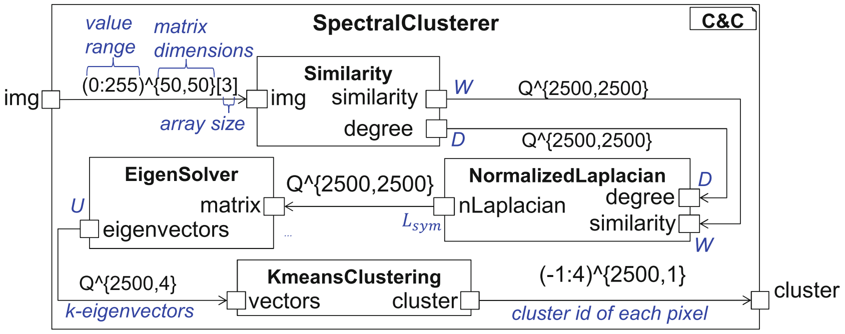


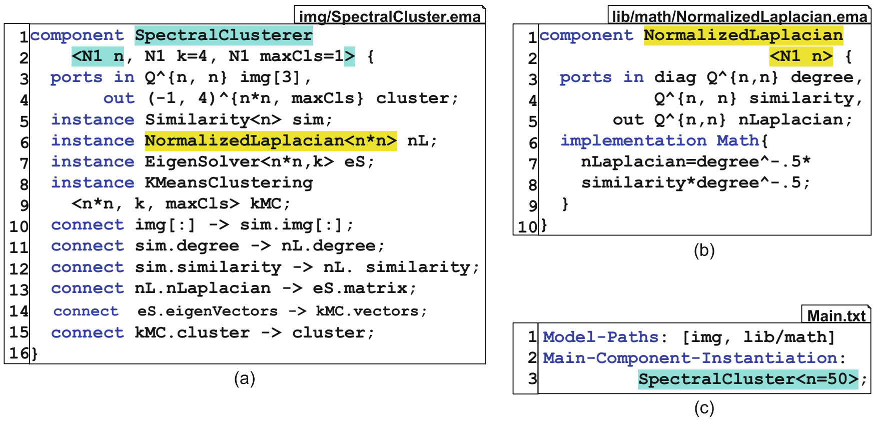
Fig. 5 C&C architecture of the `SpectralClusterer`

the height and the width of the image, respectively. First, a symmetric similarity matrix $W \in \mathbb{R}^{NM \times NM}$ is computed. Consequently, the entry of W at position (h, k) provides information on the similarity of the two pixels corresponding to the indexes h and k. Pixel similarity may be defined in terms of distance, color, gradients, etc. Second, the so-called graph Laplacian is computed as $L = D - W$ where D is the so-called degree matrix defined as $D = \text{diag}\,(W\mathbb{1}_{N \times M})$ with $\mathbb{1}_{NM}$ being an $N \cdot M$ dimensional column vector full of ones. Often it is advantageous to use the symmetric Laplacian

$$L_{sym} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}} = \text{diag}\,(\mathbb{1}_{NM}) - D^{-\frac{1}{2}} W D^{-\frac{1}{2}} \tag{1}$$

as outlined in [22]. For efficiency reasons, as they do not carry valuable cluster information the identity matrix and the minus depicted in red are often dropped in concrete implementations obtaining the simplified term highlighted in blue. Note that computing L_{sym} requires a matrix inversion on the diagonal matrix D as well as two matrix multiplications. Now the eigenvectors corresponding to the k smallest eigenvalues of L_{sym} have to be computed where k is the number of clusters we want to detect. If this number is unknown, an index can be used to estimate it [5]. Furthermore, let U be an $NM \times k$ matrix with the k eigenvectors as its columns. Each row of this matrix represents one pixel in a feature space which should be easier to cluster by the standard k-means algorithm.

EmbeddedMontiArc is a textual domain-specific language to model logical functions in a C&C based manner. *EmbeddedMontiArc* places emphasis on the needs of the embedded, cyber-physical systems, as well as the automotive domains and is particularly used for controller design [9]. As an example, the elaborate numeric type system allows declarations of variable ranges as well as accuracies. Furthermore, units are an inherent part of signal types, and hence tedious and error-prone tasks like checking the physical compatibility of signals (weights cannot be added to lengths) as well as unit or prefix conversion (feet to meters, km to m) are delegated to the *EmbeddedMontiArc* compiler.



img/SpectralCluster.ema

```
 1 component SpectralClusterer
 2    <N1 n, N1 k=4, N1 maxCls=1> {
 3   ports in Q^{n, n} img[3],
 4        out (-1, 4)^{n*n, maxCls} cluster;
 5   instance Similarity<n> sim;
 6   instance NormalizedLaplacian<n*n> nL;
 7   instance EigenSolver<n*n,k> eS;
 8   instance KMeansClustering
 9     <n*n, k, maxCls> kMC;
10   connect img[:] -> sim.img[:];
11   connect sim.degree -> nL.degree;
12   connect sim.similarity -> nL. similarity;
13   connect nL.nLaplacian -> eS.matrix;
14   connect  eS.eigenVectors -> kMC.vectors;
15   connect kMC.cluster -> cluster;
16 }
```

(a)

lib/math/NormalizedLaplacian.ema

```
 1 component NormalizedLaplacian
 2                         <N1 n> {
 3   ports in diag Q^{n,n} degree,
 4            Q^{n, n} similarity,
 5        out Q^{n,n} nLaplacian;
 6   implementation Math{
 7     nLaplacian=degree^-.5*
 8     similarity*degree^-.5;
 9   }
10 }
```

(b)

Main.txt

```
1 Model-Paths: [img, lib/math]
2 Main-Component-Instantiation:
3            SpectralCluster<n=50>;
```

(c)

Fig. 6 Textual *EmbeddedMontiArc* code of the graphical C&C `SpectralClusterer` component shown in Fig. 5. (**a**) EmbeddedMontiArc model of `SpectralClusterer`. (**b**) EmbeddedMontiArc model of atomic component `NormalizedLaplacian`. (**c**) Main component instantiation

Figure 6 shows how the spectral cluster in Figure 5 is modeled in *EmbeddedMontiArc*. Figure 6b represents one subcomponent of (a). As *EmbeddedMontiArc* is inherent to C&C languages, the main language elements of *EmbeddedMontiArc* are `component` and `connect`. While the former defines a new component followed by its name, for example, in line 1 of Fig. 6a, the latter connects two ports of subcomponents with each other, for example, in lines 10–15 in Fig. 6a.

The behavior of a component can be either defined by a hierarchical decomposition into subcomponents as in the case of Fig. 6a or using an embedded behavior description language.

The behavioral language shown in this code example is a matrix-based math language, used in lines 6–8 of Fig. 6b respectively. As *EmbeddedMontiArc* is strongly typed, errors like wrong matrix dimensions are caught at compile-time, in contrast to *MATLAB/Simulink* where this is a runtime exception. A matrix property system leverages performance optimizations as well as further compatibility checks in the compilation phase. If a matrix is declared to be diagonal, both memory and computational complexity of the generated code can be reduced dramatically. If furthermore, the domain of the matrix is constrained to non-negative entries, it can be inferred that the matrix is positive-semidefinite allowing the inversion function to be used on it and guaranteeing that the result will be positive-semidefinite again [2].

As this spectral clustering example shows each *EmbeddedMontiArc* component resides in its own text file so that multiple users or teams can work on one large C&C modeling project simultaneously. Compared to other C&C languages where models are stored in a proprietary binary format in one single file, such as Simulink's slx format, this facilitates the usage of version control systems, merging, and conflict solving but also textual searching in model repositories.

The next paragraphs explain *EmbeddedMontiArc* textual modeling language more in detail. The modeling paradigm of *EmbeddedMontiArc* is based on Java with language elements from Ada. The `Main.txt` file in Fig. 6c is similar to Java's manifest file. Line 1 defines model paths where the parser looks for component type definitions; in this example it includes the `img` (for image) and the `library/mathamatics` folders to find the `SpectralCluster` and the `NormalizedLaplacian` type. Lines 2 and 3 specify the top level component by creating one component instance of the `SpectralCluster` component type whereby it bounds the generic parameters with the following values `n=50`, `k=4`, and `maxCls=1`. The last two parameters may not be specified as shown in line 3 in (c), as `k` and `maxCls` have default values. The general generic concept is based on Java, and the concepts to have default parameters and to also enable parameter binding by name and not only via position are borrowed from Ada.

The support of generics in *EmbeddedMontiArc* enables high reusage which is not possible in *Simulink* yet. For example, for applying the same clustering algorithm on a 100×100 image instead of the small 50×50, one requires only to replace the expression `n=50` to `n=100` in line 3 in (c).

Lines 1 and 2 in Fig. 6a define the `SpectralClusterer` component type which has three input and one output ports as well as three generic parameters. The three generic parameters are of type `N1` representing the mathematical type $\mathbb{N}_1$ accepting values $1, 2, 3, \ldots$.

Line 3 defines the three input ports by applying port array notation. The names of the input ports are `img[1]`, ..., `img[3]` for the three color channels HSV. All three ports accept as values $n \times n$ (which is in our example 50×50) rational matrices which is defined as `Q^{n, n}` in *EmbeddedMontiArc*; `Q` represents the math type $\mathbb{Q}$. All three ports could be used independently, for example, by connecting each of them to different subcomponents. The `SpectralClusterer` produces as output a $n \cdot n \times maxCls$ matrix (which is in this example a vector with 2 500 elements) with the cluster id of the pixel; all elements of the `cluster` matrix are in the interval from -1 to $+4$ (including both end points).

Line 5 instantiates one `Similarity` subcomponent instance which accepts as input a 50×50 ($n = 50$) matrix and produces as output the similarity and the degree matrix of type $\mathbb{Q}^{2\,500 \times 2\,500}$.

Line 6 instantiates the `NormalizedLaplacian` subcomponent instance and bounds the value `NormalizedLaplacian.n` to 2500 (`SpectralClusterer.n` $= 50$) which is defined in Fig. 6b.

Lines 7 and 8 instantiate the two further subcomponents. Lines 10 to 15 connect the ports of the defined subcomponents as shown in Fig. 5.

EmbeddedMontiArc also supports to add algebraic information for matrix types as shown in line 3 in Fig. 6b with the keyword `diagonal`.

Additionally, *EmbeddedMontiArc* supports units. For example, `port in (0 km/h : 250 km/h) V_Obj_rel_km` defines the input port `V_Obj_rel_kmh` of the `Distronic` component as shown in Fig. 4. The strong type system with units prevents errors like connecting ports with incompatible units such as `Distance_Object_m` having port type `(0m : 500m)` with port `V_Obj_rel_kmh`.

The textual syntax for C&C design models for step 2 in Sect. 2.2 is very similar to the textual syntax of *EmbeddedMontiArc* shown in Fig. 6a. For example, the port definition in design models support a question mark as port type to express underspecification: `port out ? cluster`. Detailed information about the concrete and abstract syntax of the textual C&C high-level design and functional modeling languages is available in my PhD thesis [23, Chapters 3, 4, 7].

7 Enriching C&C Functional Models with Extra-Functional Properties in a Consistent Way

Section 6 showed the textual modeling languages for C&C high-level design models and for C&C functional models. Section 3 presented the concept how to derive the high-level design models from textual requirements and how to create one functional C&C model based on the high-level designs and the requirements. These both sections covered the steps 1–11 of Sect. 2.2.

This section continues with step 13 how the engineer enriches the functional C&C model with extra-functional properties. Therefore, this section shows first how to define a new tag schema to enrich C&C components with ASIL (automotive safety integrity level) information. Later, this section presents a tag model which is to conform to the previously defined tag schema to tag the `Vehicle` subcomponents of Fig. 4 with concrete ASIL levels. At the end this section explains how to formulate the extra-functional constraint for ASIL in *OCL*: A composed component cannot have higher ASIL than its subcomponents.

Figure 7 shows how to define a new tag schema and how to enrich it with functional C&C models. Line 1 in (a) gives the tag schema a name so that it can be used in tag models. Line 2 in (a) creates the new tag type `asil` which accepts as

TagSchema

```
1  tagschema EFP1Schema {
2    // enumeration type: asil can have one of the values QM, ..., ASIL-D
3    tagtype asil: [QM | ASIL-A | ASIL-B | ASIL-C | ASIL-D] for Component;
4  }
```

(a)

TagModel

```
1  conforms to EFP1Schema;
2  tags AsilTags{
3    tag AdaptiveCruiseControl with asil = QM;
4    tag Tempomat with asil = ASIL-C;
5    tag VelocityControl with ASIL-D;
6    tag Distronic with asil = ASIL-B;
7  }
```

(b)

Fig. 7 ASIL tag schema and tag model example. (**a**) Tag schema definition of extra-functional property `asil`. (**b**) Tag model enriching components with extra-functional property `asil`

OCL

```
1   context ComponentInst inv ASIL:
2     let
3       asilNb = List{ QM, ASIL_A, ASIL_B,
4                      ASIL_C, ASIL_D };
5       selection = subs;
6       aggregation = min {
7         min asilNb.indexOf(s.asil.value) ?: 0 |
8         s in selection.component            };
9       compareTo = min asilNb.indexOf(
10                     component.asil.value) ?: 0;
11    in          // comparison
12      aggregation >= compareTo
```

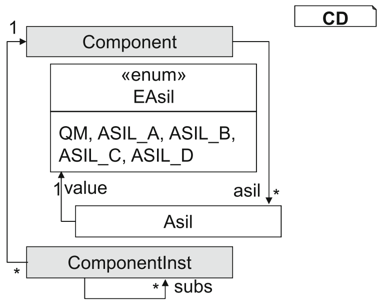


Fig. 8 *OCL* code to force that composed component cannot have higher ASIL than its subcomponents. The right part shows the class diagram of the abstract syntax of *EmbeddedMontiArc* with gray background merged with the abstract syntax defined by the tag schema in Fig. 7a with white background. Detailed information about the abstract syntax is available in my PhD thesis [23, Chapter 4]

values `QM` (not safety relevant), `ASIL-A`, ..., `ASIL-D` (highest safety level). One tag schema can contain multiple tag types to build up tag libraries. For example, a tag schema *SafetySecurity* could contain the tag types *asil*, *reliability*, *encryption*, and *firewallConfig*.

Figure 7b presents how to tag the components `AdaptiveCruise`, `Tempomat`, and `VelocityControl` in Fig. 4 with the `asil` levels `QM`, `ASIL-B`, `ASIL-C`, and `ASIL-D`.

Figure 8 shows the *OCL* code checking that the ASIL of all subcomponents must be higher or equal than the ASIL of the composed component. Lines 3 and 4 define the `asilNb` helper list. This list maps each ASIL to a number so that a comparison of ASILs is possible. The expression `asilNb.indexOf(x)` returns `0` for `QM`, `1` for `ASIL_A`, `2` for `ASIL_B`, `3` for `ASIL_C`, and `4` for `ASIL_D`.

The `indexOf`[10] function is extended to accept a set as parameter by applying the Java `indexOf` operator to each element of the set; for example, `asilNb.indexOf( {QM, ASIL_C} )` returns `{0, 3}`. *OCL* extends all Java operators to set and list operators when applying it element-wise makes sense; this way *OCL* expressions—often dealing with sets due to its navigations of associations—come along with less `forall` or `exists` statements, which makes the code easier to read.

Line 7 takes the lowest number of an ASIL when a component is tagged with two different ASIL values, and it takes `0` for QM when the subcomponent has not been tagged at all. Line 12 does the comparison.

Figure 9 shows the graphical representation of the generated positive consistency witnesses of the ASIL constraint for the `AdaptiveCruisteControl` subcomponent in the `Vehicle` example in Fig. 4. Every witness shows (a) the context (which is the `AdaptiveCruiseControl` component instance being matched by

[10] cf. http://mbse.se-rwth.de/book2/index.php?c=chapter3-2.

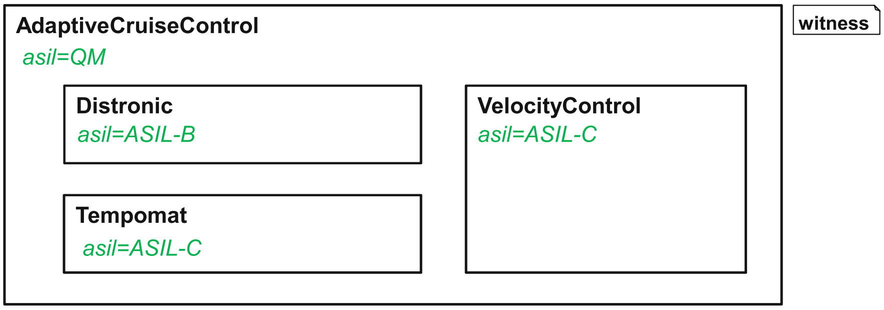


Fig. 9 Positive consistency witness of `asil` constraint for `AdaptiveCruiseControl` example in Fig. 4 enriched with the values shown in Fig. 7b.

line 1 in Fig. 8), (b) all elements stored in the `selection` variable (which are the three subcomponents `Distronic`, `Tempomat`, and `VelocityControl` being matched by line 5 in Fig. 8), (c) filling elements to understand the witness (e.g., if a port is part of (a) or (b), then also the corresponding component to which this port belongs to is shown in the witness), and (d) all extra-functional properties being addressed in the *OCL* constraint (which is in this case only the `asil` one being used in lines 7 and 10 in Fig. 8).

The generated positive (or negative) consistency witnesses are as minimal as possible. Thus, it is very easy for the engineer to understand why (or also why not) a large functional C&C model satisfies a specific consistency constraint rule as the ASIL one in this example.

This section presented a model-driven approach for adding extra-functional properties to functional C&C models. The here shown tagging mechanism enables noninvasive extensions of functional C&C models and also C&C high-level design models (shown in the next section) with attributes for extra-functional properties. Importantly, this concept provides means for integrated formal analyses of the consistency of tagged values. Consistency ranges from type-safety and units of quantitative measures to complex dependencies across component hierarchies as well as between component definitions and their instances. The *OCL* framework provides a way to define and to check rich consistency rules of extra-functional property values based on selection (line 5 in Fig. 8), aggregation (lines 6–8), and comparison (line 12) operators. This work allows for independent definition and organization of tagged properties to support reuse across models and development stages.

More *OCL* consistency examples for real-life problems are available in my PhD thesis [23, Chapter 6] and in the paper [17].[11]

[11] The *OCL* verification tool is available under https://git.rwth-aachen.de/monticore/publications-additional-material/-/tree/master/OCLVerifyTool.

8 Automatic Extra-Functional Property Verification Between Design and Functional Models

The previous section illustrated how easy it is to enrich functional C&C models with extra-functional properties. Additionally, Sect. 3 presented how company specific consistency constraints can be easily defined with our *OCL* framework and how the generated positive consistency witness looks like.

This section shortly introduces the last step 15 of the improved model-based development process elucidated in Sect. 2.2. This section explains the general idea how to check whether all extra-functional properties in the functional C&C model satisfies all extra-functional requirements specified in all C&C high-level design models.

The top part of Fig. 10 presents an extra-functional requirement about the worst case execution time for the critical path inside the `Distronic` component: *If the distance to the preceding vehicle decreases, distronic decelerates within* 20 *ms*.

The bottom left part shows the corresponding graphical C&C high-level design model; the effector from the `Distance_Object_m` port to the `Deceleration_pc` port has been tagged with the extra-functional requirement: `wcet` $\leq$ `20ms`

The bottom right part reprints the `Distronic` component of Fig. 4. Based on the measured runtime information (cf. step item 12), the atomic subcomponents of `Distronic` have been enriched with the extra-functional property execution `time` values: (i) both multiplication blocks `mult1` and `mul2` need 7 ms; (ii) the `linear` and `saturate` need 10 ms, as well as the `sum` block needs only 5 ms.

Figure 11 defines the tag types `et` (execution time) for `Components` and `wcet` (worst case execution time) for `Effectors`. Both the tag schema and the model language support all SI units completely as shown in lines 4 and 7: the execution time of a component can only have a value between 0 ms and 1 min.

2.2.1 Distronic

FA-146: (k) If the distance to the preceding vehicle decreases, distronic decelerates within 20ms.

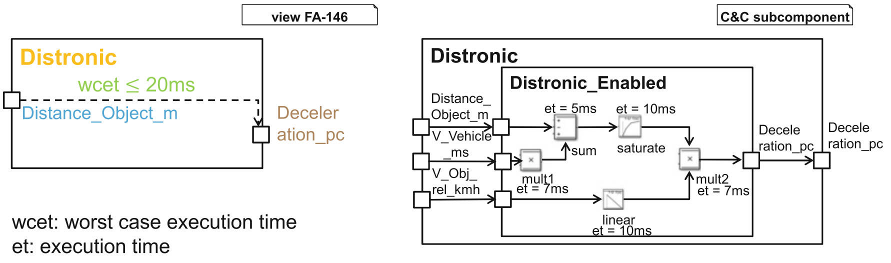


Fig. 10 ASIL tag schema and tag model example

TagSchema

```
1  tagschema ExecutionTime {
2    // number type of unit time: et (execution time) can be b/w 0 ms and 1 min
3    // only components in the functional C&C model can be tagged with et
4    tagtype et: (0ms : 1 min) for Component;
5    // only effectors in C&C high-leve desing model can be tagged with wcet
6    // (worst case execution time)
7    tagtype wcet: <= (0ms : 1 min) for Effector;
8  }
```

Tag schema definition of extra-functional properties `et` and `wcet`.

Fig. 11 ASIL tag schema and tag model example

TagModel

```
1  conforms to ExecutionTime;
2  tags WcetTags{
3    within Distronic {
4      tag effector Distance_Object_m -> Deceleration_pc with wcet <= 20ms;
5    }
6  }
```

(a)

TagModel

```
1  conforms to ExecutionTime;
2  tags EtTags{
3    within Distronic_Enabled {
4      tag sum with et = 5ms;
5      tag saturate, linear with et = 10ms;
6      tag mult1, mult2 with et = 7ms;
7    }
8  }
```

(b)

Fig. 12 ASIL tag schema and tag model example. (**a**) Tag model enriching C&C high-level design view `FA-146` with extra-functional property `wcet`. (**b**) Tag model enriching C&C high-level design view `FA-146` with extra-functional property `wcet`

Tag types to enrich the functional C&C model, as already presented in Sect. 3, represent concrete extra-functional property values. Therefore, these always have an assignment operator (=), which can be skipped as shown in line 4.

Tag types to enrich the C&C high-level design model represent underspecified extra-functional requirements expressing a Boolean constraint. For this reason the Boolean comparison operator (such as $<$, $\leq$, $\subseteq$, $\in$, $\supset$, $\geq$, $>$) must be specified as shown with $<=$ in line 7.

Figure 12a shows that C&C high-level design models can be enriched with extra-functional requirements nearly in the same way as functional C&C models with extra-functional properties. Ports are identified via their component name and their port name, because different components may have the same port name. The `within` keyword in line 3 is syntactic sugar to express that all expressions inside it refer to the component specified. Line 4 enriches the effector with the extra-functional requirement as illustrated in the bottom left part of Fig. 10. An equivalent notation of lines 3–5 is the following: `tag effector Distronic.Distance_Object_m -> Distronic.Deceleration_pc with wcet <= 20ms;`.

Figure 12b displays the textual code how to enrich the functional C&C model with the measured execution times as illustrated in the bottom right part of Fig. 10. The tag model language supports to enrich multiple C&C elements at once with the same value as shown in lines 5 and 6.

In our example, we assume that the (here not shown *OCL*) constraint does the following: First, it selects all components of the C&C subcomponent which are matched by effector shown in `view FA-146` in Fig. 10. These are the components `Distronic`, `Distronic_Enabled`, `sum`, `saturate`, and `mult2`. Second, it aggregates all the execution time values of these components whereby a missing `et` tag is interpreted as 0 ms. The aggregation result is $22ms = 0ms + 0ms + 5ms + 10ms + 7ms$. Third, it compares whether the aggregation result of the functional C&C model is smaller or equal to the `wcet` (worst case execution time) requirement of the enriched effector. In this example, the aggregation result is $22ms$ and is NOT smaller or equal to the requested worst case execution time of $20ms$ of the enriched effector in the `view FA-146`. Thus, the functional C&C model in the bottom right part of Fig. 10 does not satisfy the view as well as it violates the `FA-146` requirement.

The verification tool does not only automatically check whether all extra-functional properties of the functional C&C model satisfy all extra-functional requirements specified in all C&C high-level designs, it also generates useful witnesses directly pointing to the mismatch. Figure 13 shows the graphical representation provided by the verification algorithm for this concrete example. The satisfaction witness creation algorithm for extra-functional properties is similar to the consistency witness creation algorithm presented in the previous section.

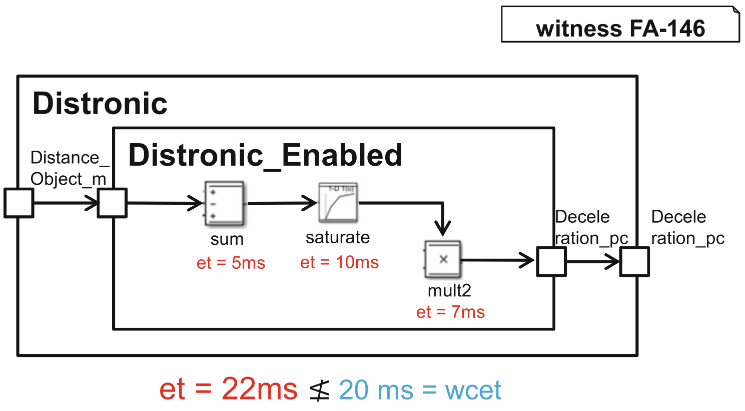


Fig. 13 ASIL tag schema and tag model example

9 Conclusion

This chapter showed the developed methodology to improve the model-based systems engineering process of large and complex C&C models for embedded and cyber-physical systems, especially in the automotive domain. The here presented approach extends the current model-based development process of large car manufactures identified during common case studies.

The main achievements of this work are concepts and tools for automatic consistency checks between requirements, high-level design models, functional C&C models, and extra-functional properties. These automatic checks with its user-friendly witness generations prevent errors at design and implementation phase, and thus, provide a way to improve quality, to increase development speed, and to save money.

This chapter demonstrated the capabilities and benefits of the proposed model-based process improvements on a running example of an advanced driver assistant system.

References

1. Bertram, V., Maoz, S., Ringert, J.O., Rumpe, B., von Wenckstern, M.: Component and connector views in practice: an experience report. In: Conference on Model Driven Engineering Languages and Systems (MODELS'17), pp. 167–177. IEEE, Piscataway (2017). http://www.se-rwth.de/publications/Component-and-Connector-Views-in-Practice-An-Experience-Report.pdf
2. Borgmann, M.: Matrix taxonomy (2006). https://www.nari.ee.ethz.ch/teaching/ha/handouts/linalg3p.pdf
3. Brenner, C.: How to ensure functional safety, according to ISO 26262 (2013). https://blogs.itemis.com/en/how-to-ensure-functional-safety-according-to-iso-26262. Accessed 29 April 2021
4. Cheng, C.H.: autoCode4 integrated inside Ptolemy II (ver. 11.0.devel) (2016). https://youtu.be/ImSHmsnUyeA?t=34s. Accessed 31 July 2018
5. Desgraupes, B.: Clustering indices. Univ. Paris Ouest-Lab Modal'X **1**, 34 (2013)
6. Drave, I., Greifenberg, T., Hillemacher, S., Kriebel, S., Kusmenko, E., Markthaler, M., Orth, P., Salman, K.S., Richenhagen, J., Rumpe, B., Schulze, C., Wenckstern, M., Wortmann, A.: SMArDT modeling for automotive software testing. Softw. Practice Exp. 49(2), 301–328 (2019)
7. Goser, A.: MATLAB Answers: Clean up Simulink block diagram (2012). https://de.mathworks.com/matlabcentral/answers/30016-clean-up-simulink-block-diagram. Accessed 31 July 2018
8. Grazioli, F., Kusmenko, E., Roth, A., Rumpe, B., von Wenckstern, M.: Simulation framework for executing component and connector models of self-driving vehicles. In: Proceedings of MODELS 2017. Workshop EXE, CEUR 2019 (2017). http://www.se-rwth.de/publications/Simulation-Framework-for-Executing-Component-and-Connector-Models-of-Self-Driving-Vehicles.pdf
9. Hillemacher, S., Kriebel, S., Kusmenko, E., Lorang, M., Rumpe, B., Sema, A., Strobl, G., von Wenckstern, M.: Model-based development of self-adaptive autonomous vehicles using the SMARDT methodology. In: Proceedings of the 6th International Conference on Model-Driven

Engineering and Software Development (MODELSWARD'18), pp. 163–178. SciTePress, Setúbal (2018)
10. Kriebel, S., Kusmenko, E., Rumpe, B., von Wenckstern, M.: Finding inconsistencies in design models and requirements by applying the SMARDT process. In: Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme XIV (MBEES'18). Univ. Hamburg (2018). http://www.se-rwth.de/publications/Finding-Inconsistencies-in-Design-Models-and-Requirements-by-Applying-the-SMARDT-Process.pdf
11. Kusmenko, E., Pavlitskaya, S., Rumpe, B., Stüber, S.: On the engineering of AI-powered systems. In: O'Conner, L. (ed.) ASEW19. Software Engineering Intelligence Workshop (SEIW19), pp. 126–133. IEEE, Piscataway (2019). http://www.se-rwth.de/publications/On-the-Engineering-of-AI-Powered-Systems.pdf
12. Kusmenko, E., Ronck, J.M., Rumpe, B., von Wenckstern, M.: EmbeddedMontiArc: textual modeling alternative to simulink. In: Proceedings of MODELS 2018. Workshop EXE (2018)
13. Kusmenko, E., Rumpe, B., Schneiders, S., von Wenckstern, M.: Highly-optimizing and multi-target compiler for embedded system models: C++ compiler toolchain for the component and connector language EmbeddedMontiArc. In: Conference on Model Driven Engineering Languages and Systems (MODELS'18). IEEE, Piscataway (2018)
14. Kusmenko, E., Rumpe, B., Strepkov, I., von Wenckstern, M.: Teaching playground for C&C language EmbeddedMontiArc. In: Proceedings of MODELS 2018. Workshop ModComp (2018). http://www.se-rwth.de/publications/Teaching-Playground-for-C-and-C-Language-EmbeddedMontiArc.pdf
15. Maoz, S., Pomerantz, N., Ringert, J.O., Shalom, R.: Why is my component and connector views specification unsatisfiable? In: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 134–144 (2017). https://doi.org/10.1109/MODELS.2017.26
16. Maoz, S., Ringert, J.O., Rumpe, B.: Synthesis of component and connector models from crosscutting structural views. In: Meyer, B., Baresi, L., Mezini, M. (eds.) Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13), pp. 444–454. ACM, New York (2013). http://www.se-rwth.de/publications/Synthesis-of-Component-and-Connector-Models-from-Crosscutting-Structural-Views.pdf
17. Maoz, S., Ringert, J.O., Rumpe, B., von Wenckstern, M.: Consistent extra-functional properties tagging for component and connector models. In: Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'16), CEUR Workshop Proceedings, vol. 1723, pp. 19–24 (2016). http://www.se-rwth.de/publications/Consistent-Extra-Functional-Properties-Tagging-for-Component-and-Connector-Models.pdf
18. National Instruments: Automatische Bereinigung von LabVIEW-Blockdiagrammen (2009). http://www.ni.com/tutorial/7386/de/. Accessed 31 July 2018
19. Plataniotis, G., Ma, Q., Proper, E., de Kinderen, S.: Traceability and modeling of requirements in enterprise architecture from a design rationale perspective. In: Research Challenges in Information Science (RCIS), 2015 IEEE 9th International Conference on, pp. 518–519. IEEE, Piscataway (2015)
20. Richenhagen, J., Rumpe, B., Schloßer, A., Schulze, C., Thissen, K., von Wenckstern, M.: Test-driven semantical similarity analysis for software product line extraction. In: International Systems and Software Product Line Conference (SPLC '16), pp. 174–183. ACM, New York (2016). http://www.se-rwth.de/publications/Test-Driven-Semantical-Similarity-Analysis-for-Software-Product-Line-Extraction.pdf
21. Rumpe, B., Schulze, C., von Wenckstern, M., Ringert, J.O., Manhart, P.: Behavioral compatibility of simulink models for product line maintenance and evolution. In: Software Product Line Conference (SPLC'15), pp. 141–150. ACM, New York (2015). http://www.se-rwth.de/publications/Behavioral-Compatibility-of-Simulink-Models-for-Product-Line-Maintenance-and-Evolution.pdf
22. Von Luxburg, U.: A tutorial on spectral clustering. Stat. Comput. **17**(4), 395–416 (2007)

23. von Wenckstern, M.: Verification of Structural and Extra Functional Properties in Component and Connector Models for Embedded and Cyber Physical Systems. Aachener Informatik-Berichte, Software Engineering, Band 44. Shaker Verlag (2020). http://www.se-rwth.de/phdtheses/Diss-von-Wenckstern-Verification-of-Structural-and-Extra-Functional-Properties-in-Component-and-Connector-Models-for-Embedded-and-Cyber-Physical-Systems.pdf
24. Zion Market Research: Global Embedded Systems Market Will Reach USD 225.34 billion by 2021 (2017). https://tinyurl.com/ofetbpzw. Accessed 14 February 2021