Towards Component Modelling of Embedded Systems in the Vehicular Domain

Tomáš Bureš, Jan Carlson, Séverine Sentilles and Aneta Vulgarakis
Mälardalen Real-Time Research Centre, Västerås, Sweden.
{tomas.bures,jan.carlson,severine.sentilles,aneta.vulgarakis}@mdh.se

Abstract

The complexity of software and electronics in vehicular systems has increased significantly over last few years — up to the point when it is difficult to manage it with existing development methods. In this paper we aim at using components for managing the complexity in vehicular systems. Compared to other approaches, the distinguishing feature of our work is using and benefiting from components throughout the whole development process (from early design to development and deployment). Based on the elaboration of the specifics of vehicular systems (resource constraints, real-time requirements, hard demands on reliability), the paper identifies concerns that need to be addressed by a component model for this domain. It also outlines basic features and characteristics of such a component model and discusses how relevant existing formalisms and component models relate to it and how they could be reused within the proposed approach.

1 Introduction

Vehicles of various types have become an integral part of the everyday life. In addition to cars, which are the most common, they comprise other transportation vehicles (such as trucks and busses) and special purpose vehicles (e.g. forestry machines). It is a general trend that the level of computerization in the vehicles grows every year. For example in the automotive industry, the complexity of the electrical and electronic architecture is growing exponentially following the demands on the driver's safety, assistance and comfort [7].

The computerization is present in vehicles in the form of *embedded systems*, which are special-purpose built-in computers. They are tailored to perform a specific task by combination of software and hardware. In comparison to general purpose computers, one fundamental characteristic of embedded systems is that many of them have to function under severe resource limitations in terms of memory, bandwidth and energy, and under difficult environmental conditions (e.g. heat, dust, constant vibrations).

As the embedded systems are often used for safety-critical tasks, there are typically requirements on *real-time behaviour*, meaning that a system must react correctly to events in a well-specified amount of time (neither too fast nor too slow) since any infraction of these requirements can lead to a catastrophe. A popular example to illustrate this is a car airbag. In case of an accident, the airbag has to inflate suitably at a particular point in time, otherwise it is useless or even harmful for the driver.

The criticality of tasks connected with embedded systems implies that embedded systems have to be thoroughly tested or better formally verified for correctness (both functional and with respect to timing).

The restrictions in available resources (power, CPU and memory), environmental conditions and harsh requirements in terms of safety, reliability, worst-case response time, etc. make the development of embedded systems rather difficult and time-demanding. And what may be feasible when the embedded systems in a vehicle are few and simple gets immensely more difficult when they grow in number, get more complex and become mutually dependent (many systems are designed as distributed systems communicating over some network) — as is the trend today. Even the typical solution having been applied so far — encapsulating an embedded system into a dedicated ECU with its own CPU and memory — does not scale any more due to restrictions in physical space and available power. Instead there arises a need to collocate several embedded systems on one physical unit which even more adds on complexity as resources have to be shared.

All this introduces a new challenge in software development for embedded systems in vehicular domain. Already taken apart, each of the aforementioned issues (resource limitations, correctness of system behaviour, reliability, distribution, etc.) is challenging by itself but together they represent a very complex problem. A problem which the current embedded systems development methods do not seem to be able to easily cope with.

A promising solution lies in the adoption of a Component-Based Development (CBD) approach, which allows construction (resp. decomposition) of software systems out of (resp. in) independent and well-defined pieces of software, called *components*. CBD has the potential to significantly alleviate the management of the ever-increasing complexity and give possibility to reuse already developed elements — thus increasing reliability and shortening the development time.

CBD has already proved to be successfully used in enterprise systems, service-oriented and desktop domains [6]. However, the in order to employ it in embedded systems it is necessary to adapt it to support specifics of the embedded systems in vehicular domain (i.e. strong dependence on hardware, distribution, real-timeness, to mention just a few).

There have been several approaches (e.g. [1, 8, 9, 10, 18]) to use CBD in embedded systems. Although, these approaches were successful in solving particular pieces of the puzzle, a holistic approach using CBD throughout all the stages of the embedded system development process is still missing.

1.1 Goals of the paper and structure of the text

Striving for a CBD process in vehicular embedded systems, we have taken a step back and re-evaluated the requirements of embedded systems in the vehicular domain with the goal of setting up CBD and underlying component models that would allow using components throughout the whole development process (from early design to deployment).

The goal of this paper is to establish concepts and requirements for a CBD process for vehicular embedded systems and to characterize the component models underlying it — with the main objectives of (a) aligning the CBD with specifics of vehicular embedded systems, (b) reducing system complexity, (c) increasing dependability by allowing for various kind of analyses (functional behaviour, timing behaviour, reliability), and (d) reducing development time by supporting reuse. An emphasis also lies in supporting components in all stages of the development process.

The remainder of this paper is organized as follows: Section 2 introduces a concrete example of an embedded system in the vehicular domain and Section 3 describes the background of this work. Section 4 identifies key concerns to be addressed when applying CBD to the vehicular domain and Section 5 outlines a suitable component model family. Related work is described in Section 6, and Section 7 concludes the paper.

2 Example

As a running example demonstrating the specifics of the vehicular domain, we will consider the electronic systems of a modern car, focusing on an anti-lock braking system (ABS) in particular.

Today, electronics and software stands for a substantial part of the producton cost of a car (as much as 40% [5]). This is explained by the fact that much of the competitive edge of a modern car is provided by functionality realised by software. Such functionality includes infotainment, climate control and navigation systems, but also core functionality such as engine control, shift-by-wire, anti-lock braking and airbags.

The physical system architecture can consist of a fairly large number of computational nodes (ECUs), connected by a number of different networks. For example, a Volvo XC90, depicted in Figure 1, has around 40 ECUs, two CAN busses of different speed, several LIN busses, and a MOST bus for the infotainment systems.

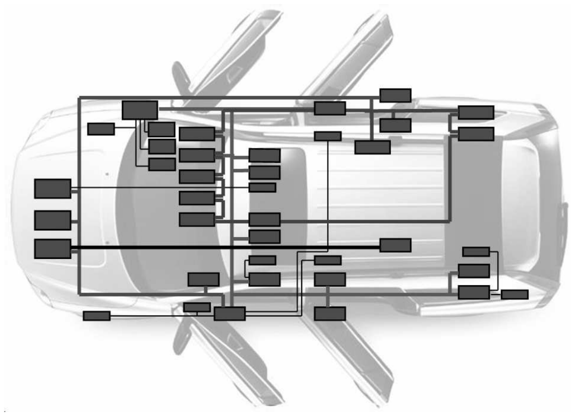

Figure 1. The electronic system architecture of Volvo XC90.

In the automotive domain, low production cost is a very important concern, since each car model is manufactured in such large quantities. At the same time, many of the electronic systems are highly safety-critical, and some are subject to hard real-time constraints. Thus, a key design challenge is finding a minimal (with respect to cost, but this typically means minimal in terms of resources, as well) that can provide the desired functionality with a sufficiently level of dependability.

Looking specifically at the ABS, its role is to improve the braking performance by preventing the wheels from locking. When a wheel is about to lock — a situation characterised by the speed of that wheel being significantly lower than that of the other wheels — the brake force should be decreased until the wheel starts to move faster again.

In addition to the main functionality, the ABS is responsible for monitoring for hardware or software faults, including faults in the associated sensors and actuators. Transient faults should be handled locally, and in case of persistent problems, the system should be shut down in a safe way and the driver should be informed.

Functionally, the ABS is fairly independent from other subsystems, although it shares some information about the state of the vehicle with other subsystems. For example, if the ABS is shut down, other subsystems might want to change the way they operate. Also, the ABS could share wheel speed sensors and brake actuators with, e.g., a traction control system (TCS).

At a more fine-grained level of detail, there are many design decisions to be made in order to achieve an optimal performance: what wheel speed difference should be tolerated without the system considering it a locking situation, exactly how much and for how long should the braking force be adjusted, etc. This type of concerns are tightly connected to the behavior of the actual car interacting with its environment, and might require significant testing and fine-tuning. For many of them, control theory provides well established parameterised solutions that can be adjusted by simulations and actual tests.

The correctness and quality of the ABS system strongly depends on its real-time behaviour, e.g., how often the wheel speed is sampled and the time delay between sampling and actuating. This adds to the complexity, since these temporal aspects depend on many factors outside the ABS, such as other subsystems using the same bus. The current trend in automotive systems is towards running multiple subsystems on the same physical node, which introduces additional temporal dependencies.

It is common that sub-systems are developed relatively independently by a few large OEM manufacturers, who sell the sub-systems to car manufacturers. In the case of ABS it means that the ABS sub-system produced by one company is used (with some modifications) in several car brands. That brings the necessity to be able to reuse the overall design of the ABS at a level which abstracts from the interference from other sub-systems in the car. Although the overall functionality remains unchanged when the ABS subsystem is reused in a different car model, it is typically necessary to adjust details, e.g., how much the brake force should be decreased in a locking situation, depending on the characteristics of the car. Thus, it is not enough to reuse the ABS sub-system just as a “black box”. Instead, it is necessary to be able to access the internal structure to make adjustments on the appropriate level of detail.

3 The PROGRESS approach

The work presented in this paper is conducted as a part of the larger research vision of PROGRESS, which is a Swedish national research centre for predictable development of embedded systems. In this section we provide a brief overview of the PROGRESS vision as it provides background and motivation for our work.

The goal of PROGRESS is to provide theories, methods and tools to increase quality and reduce costs in the development of systems for vehicular, automation and telecommunication domains. Together they are to cover the whole development process, supporting the consideration of predictability and safety througout the development. To support this idea and propose a basis for work, PROGRESS relies on a holistic approach using CBD throughout all the stages of the embedded system development process together with an interlacing of various kind of analysis and an emphasis on reusability issues.

To be able to apply a CBD approach across the whole development process (starting from a vague specification of the system based on early requirements up to its final and precise specification and implementation ready to be deployed), PROGRESS adopts a particular notion for component. Similarly to SaveCCM [1] and Robocop [10], a component is considered as “a whole”, i.e. a collection gathering all the information needed and/or specified at different points of time of the development process. That means a component comprises requirements, documentation, source code, various models (e.g. behavioural and timing), predicted and experimentally measured values (e.g. performance and memory consumption), etc., thus making a component a unifying concept in the whole development process.

In addition to modelling with components (which is the topic of this paper), PROGRESS puts a strong emphasis on analysis and deployment.

The analysis parts of PROGRESS aim at providing estimations and guarantees of different important properties. The analysis is present throughout the whole development process and gives results depending on the completeness and accuracy of the components’ models and description. This means that an early (and rather inaccurate) analysis may be performed during design to guide design decisions and provide early estimates. Once the development is completed the analysis may be used to validate that the created components and their composition meet the original requirements. The different analyses planned for PROGRESS include reliability predictions, analysis of functional compliance (e.g. ensuring compatibility of interconnected interfaces), timing analysis (analysis of high-level timing as well as low-level worst-case execution time analysis) and resource usage analysis (e.g. memory, communication bandwidth).

Deployment in PROGRESS is strongly conforming to specifics of embedded real-time systems. The design and development of components is supplemented by deployment activities consisting of two parts: (1) allocation of components to physical nodes and (2) code synthesis. In code synthesis, the code of components are merged, optimized and mapped to artefacts of an underlying real-time operating system. This step also includes creating real-time schedules. The binary images resulting from code synthesis are ready to be executed at the target physical nodes.

4 Towards CBD in vehicular systems

In this section we outline two important requirements on CBD when used in the vehicular domain, focusing on how CBD can be integrated with early design and a high level of predictability.

In a broad sense the development of an embedded system or a sub-system means going from an abstract specification to a concrete product. Starting with vague or incomplete descriptions, information regarding the software structure, timing, the physical platform, etc., is gradually introduced in order to approach a finished system. As discussed earlier, this whole process should be supported by analysis to support early detection of problems, and to achieve a high quality in the final product. When a system is developed by reusing existing components, which is a key idea in CBD, this progression from abstract to concrete becomes more complex, since concrete reused components are mixed with early (i.e., abstract) versions of components to be developed from scratch.

Another important concern — conceptually separate from the progression from abstract to concrete — relates to component granularity. In a system as complex as those found in the vehicular domain, it is clear that components representing big parts of the whole system are different from those responsible for a small part of some control functionality.

4.1 Abstract to concrete

The development of an embedded system or a subsystem typically starts with use-cases, domain diagrams and basic sketches of the system. These abstract models are then gradually detailed to eventually end up with an implementation.

Some properties of the system may be specified in a very concrete and detailed manner already in early stages of development (e.g. real-time requirements, messages used for interaction with existing systems, etc.), however, it is the fact that the overall system is far from a concrete implementation that makes it abstract at this stage.

With regard to CBD, the abstract-to-concrete concern typically means that a system is first modelled by a set of components, which however have only vague boundaries and only some properties and requirements specified. Also the communication among the components is perhaps only represented by lines representing arbitrary exchange of some data. Gradually during the development this abstract view is made more concrete, meaning that components are assigned behaviour, communication is detailed, concrete interfaces are identified and components are implemented.

A closer inspection reveals that this process from abstract to concrete is far from a straightforward linear progression in a series of well-defined system wide steps. In particular, the following issues must be taken into consideration:

- It is often necessary to move back and forth between the abstraction levels in order to explore and reject different design alternatives.
- At a particular point in time, different parts of the system will be modelled on different levels of abstraction — for example, when reusing an existing (concrete) component in a system which is not yet so mature otherwise, or when the development of different parts is not performed concurrently and at the same pace (which is the typical case).
- Some analysis techniques require a certain level of abstraction, either because the required information is not present at higher abstractions, or because the complexity of a more concrete level makes the method prohibitively expensive.

This requires the component model supporting this process to provide support for initial and abstract design as well as detailed and concrete design. An important requirement is also supporting the transformation (progression) between abstract and concrete (as opposed to having just two descriptions without any direct correspondence between them). Moreover a component should contain the information from all levels of abstraction through which it has progressed, so that even a reused concrete component may be used in the abstract design together with other abstract components.

Two particular aspects of the abstract-to-concrete scale are discussed further: *structural decomposition* and *target platform*. Other important concerns, which are not elaborated here, include *data*, *timing* and *resource consumption*.

4.1.1 Structural decomposition

In an abstract form, a component can be modelled as a black box, not because the internal structure must remain hidden but because it has not been decided yet. The functionality of the component, as well as aspects related to timing, resource consumption, communication, etc., can be modelled with respect to the externally visible interface of the component, which allows the information to be taken into account in the analysis.

As one important part of the progression to a concrete system, the internal structure of the component should be elaborated. This includes, for example, deciding whether to realise the component by means of composed subcomponents (reusing existing or developing new), or to implement it as an atomic unit.

4.1.2 Target platform

The coupling between the software and the target platform is typically quite high in an embedded system. One reason is to achieve the required functionality with the least manufacturing costs, especially when producing a system in large quantities. As the result, the hardware is typically quite restricted and the software is tailored and optimized specifically for that particular hardware and real-time operating system.

The target platform is often predetermined to some extent already by the initial requirements on the system, and additional knowledge comes from experience with previous versions of the system, or similar products. However it is not always fully known in all details. A lot of details are refined as the actual system is being developed and assumptions of individual components on the target platform are being clarified. Thus the development of a system influences and in turn is influenced by the target platform specification.

In our example, it is known a priori that the ABS will be distributed over at least five physical nodes, dictated by the physical location of the wheel speed sensors and the actuators. We would also typically be able to make some assumptions about the nature of these nodes and the network between them, based on experience from other systems. However, the final choice of hardware might be made later, as well as the decision whether the main functionality of the ABS will be allocated to a dedicated node or if it will share a node with other subsystems.

This reality of system development being interwoven with target platform specification is however in contrast to the main goals of CBD — component reusability. This poses a challenge for the component model and the associated CBD process, which must be able to take into account the target platform while not sacrificing the reusability of components.

4.2 Component granularity

In a system as complex as a typical vehicular system, it is clear that components representing big parts of the whole system are different from those responsible for a small part of some control functionality. Components at different granularity have different needs in terms of execution model, communication style, synchronisation, etc., but also with respect to the kind of information that should be associated with the component and the type of analysis that is appropriate.

In general, the big components encapsulate complex functionality but they are relatively independent. In current systems it is often the case that each of those big components are allocated to one or several dedicated ECUs. Thus, the communication between big components often manifests as messages sent over a bus in order to share data (e.g. the current vehicle speed used by several sub-systems) or to notify other components of important events. The small components (e.g. control loops, tasks), on the other hand, tend to have dedicated, restricted functionality, simple communication and stronger synchronisation. The semantics of small components is also tailored for some specific purpose (e.g. control logic).

With respect to the component model this means having different kinds of components with different semantics depending on at which level of granularity the component lies and what it is meant for (e.g. modeling control logic vs. modelling a user interface). Having these several levels of components it is vital to establish relation between them so as a big component may be modelled out of small components.

5 Conceptual component model family

In previous sections we have described the general PROGRESS view and outlined important concerns when applying a CBD approach to vehicular systems. However, a component model is needed to provide the formal grounds for the CBD process. This component model must be able to handle the concerns presented above and also facilitate the analysability and synthesisability of the system.

Ideally, the whole range from abstract to concrete but also from big to small components should be addressed by a single unified component model. However, since the demands differ significantly between the end points of the two scales, this is not an easy task. Instead, we envision splitting the abstract to concrete scale into two distinct levels of abstraction. Similarly, in order to address the differences related to component size, the concrete half is further split into two levels of granularity. This partitioning of the problem into three distinct segments is depicted in Figure 2.

Regarding the abstract to concrete scale, the abstract half represents the formalisms used to capture overall requirements, scenarios, etc. It also includes abstract models of resource usage, functional behaviour and timing.

The component models used for the concrete segments

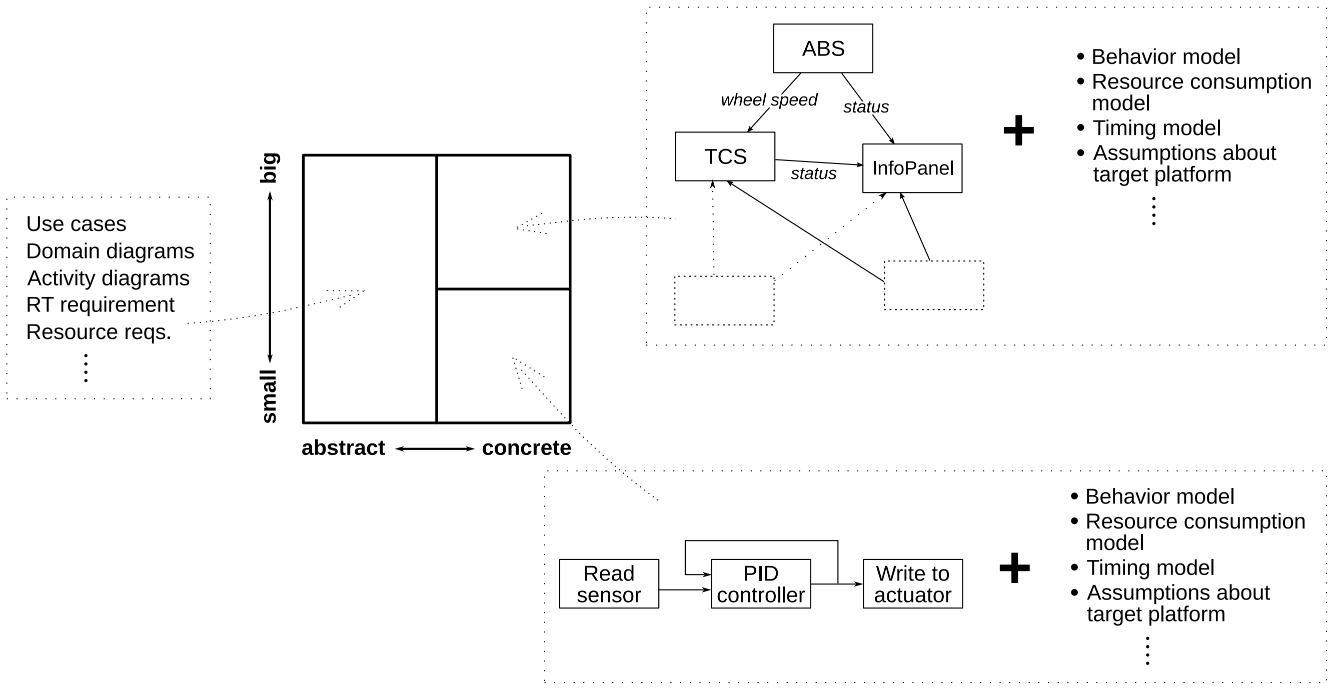

Figure 2. Proposed component model family.

are concrete in the sense that they allow modelling of concrete concerns (e.g., communication ports and concrete resource usage) and eventually end up having code implementation for all primitive components. It is however important to note that they target a rather large interval of the abstract to concrete scale, and not just a single point.

The concrete component models support components also in relatively abstract forms, i.e., where the internal structure, allocation to physical nodes, etc. is yet to be determined. It is possible to manipulate such "unfinished" components in the same way as the concretized ones (i.e., storing them in a repository, composing them with other components, include them in analysis, etc.). Gradually, as a component is filled with information, including realisation in terms of source code or an internal structure of subcomponents, it is available to more analyses and eventually to synthesis.

In order to address the coupling between components and the target platform, we allow components to express their partial assumptions about the platform (e.g. the minimum available memory, required operating system functionality). However, the detailed specification of the hardware and the platform, as well as the allocation of components to physical nodes, are given by separate models connected with deployment — i.e., they are not part of the component specification.

Although decoupled from the component model point of view, the overall development process and tool support shall allow the design and development of the software functionality to be interleaved with decisions regarding the target platform and component allocation. Such parallel refinement and specification of components, target platform and allocation is important for analysis which will often have to use all of them to produce relevant results.

On the granularity scale we divide the concrete half of the overall component model to two segments (component models) for "big" and "small" components, respectively.

The big components represent the sub-systems in the vehicular domain. These components are quite large, relatively independent and they are units of distribution and binary packaging. A sub-system may in our model consist of other sub-systems, thus forming a hierarchical component model. On the top-level a composition of sub-systems forms a system, which in our case corresponds for example to all the software running in a car.

The decomposition of a sub-system stops at primitive sub-system components, which are those realized by legacy code or modelled by small granularity components. The small components in vehicular domain serve for modelling the control logic, such as reading data from a sensor, controlling an actuator, etc. In this respect they provide an abstraction of the tasks and control loops typically found in control systems. During deployment, the small components are synthesised together to make up the code of the primitive sub-system.

In the realization of the concrete component models we thus envisage relatively simple component model for the "big" components with asynchronous message-passing as the main communication paradigm. For the "small" components we envision a component model based on a pipe-and-filter architectural style with explicit support for modelling the workflows of control loops.

To demonstrate the proposed component model family on the running example: The abstract design would specify the general requirements on functionality, timing and resources of the whole system and of individual subsystems such as the ABS. Some of these requirements would be formulated in specialised formalisms, for example the desired behaviour in case of transient faults, or the stability of the control signal.

The "big" granularity concrete design would, in its most abstract form, list the main subsystem component and indicate communication among them. This design would then be gradually concretised by elaborating on the internal structure, in which the ABS subsystem would be further split up into six sub-system components — four of them modelling sensing the speed of one wheel, one for controlling the braking and one for monitoring. The developer would also introduce assumptions about the target platform, and detail the communication needed to share the wheel speed and for informing the driver and other subsystems about possible malfunction.

At the "small" granularity level, the ABS sub-systems would consist of components responsible for interacting with speed sensors, computing the desired brake force adjustment, and so forth.

6 Related work and candidate technologies

In this section, we provide a brief overview on how and where the most relevant existing technologies for developing component based systems can fit into the proposed component model family for vehicular systems.

For the abstract part of the abstract-to-concrete scale, general purpose modeling languages such as UML [13] could be applied, in particular when targeting the whole system or big components. Use-case, interaction and deployment diagrams are suitable for capturing vague information about early requirements and modelling, but have no clear mapping to code. Issues related to timing and resource usage are addressed by specialized profiles, e.g., MARTE [14] for modelling real-time and embedded systems.

The detailed control functionality can also be modelled in some formalism that abstracts from the concrete system structure. As an example, Simulink [17] from MathWorks is a tool for modelling dynamic systems in either continuous or sampled time. These models can be simulated and analyzed, and there is support for synthesising executable code. There is however no support for adding concrete information about allocation on nodes, structural decomposition or resources.

A general concern when using established tools and formalisms for abstract modelling, is to define the relation between the concepts at this initial level and the formalisms used for the concrete modeling. This relation should ideally allow the designer to move freely back and forth between the abstraction levels, rather then being a one-way transformation from abstract to concrete concepts.

On the concrete side of the scale, an interesting representative of the approaches focusing on the "big" components is the Automotive Open System Architecture (AUTOSAR) initiative from the automotive domain [3]. AUTOSAR aims at defining a standardized platform for automotive systems, allowing subsystems to be more indendent of the underlying platform and of the way functionality is distributed over the ECUs. AUTOSAR c omponents communicate transparently of whether they are located on the same or different ECUs. The supported communication styles are based on the client-server and sender-receiver paradigms.

With regard to the granularity, most contemporary component models — including COM [15], CORBA [4] and Enterprise JavaBeans [11] — fall into the segment of "big" concrete components. However, these models consider components only as concrete binary units, thus addressing only the most concrete point at the end of the abstract-to-concrete scale. Also, inadequate timing predictability and the additional computing and memory resources consumed by the run-time component framework make them less suitable for development of embedded real-time systems. Recently, approaches to extend and adapt these component models to better suit this domain have been proposed [9, 16].

Most component models that specifically target embedded systems focus primarily on "small" granularity components. Examples include Philips' Koala component model for consumer electronics [18], the Rubus component model [2] for distributed embedded control systems with mixed real-time and non-real-time functions, the component model for industrial field devices developed in the PECOS project [12] and SaveCCM [1] for embedded control applications in the automotive domain.

Compared to many general purpose component models, these are still abstract in the sense that components are design time entities rather than executable units, and a dedicated synthesis step is assumed in which the component based design is transformed into an executable system. However, compared to pure abstract modeling of functionality, the components here represent concrete units that are realized by individual pieces of source code and usually provide some concrete information about resource usage and timing.

Interesting is also the approach of COMDES II [8], where a two-level model is employed to address the varying concerns at different levels of granularity. At the system level, a distributed system is modeled as a network of communicating actors, and at the lower level the functionality of individual actors is further specified by interconnected function blocks.

7 Conclusion

In this paper we have aimed at establishing concepts, requirements and a component model family for a CBD process in vehicular embedded systems. Compared to existing approaches, we have put emphasis on supporting components throughout the whole development phase (from early design to deployment). We have demonstrated specifics of vehicular embedded systems on the ABS example. Then we have discussed the requirements on the CBD and outlined the family of component models supporting this CBD. Eventually, we have discussed existing approaches to the development of vehicular embedded systems and matched them to the proposed conceptual component family.

As what regards to the on-going work, we work on concretizing the proposed component models and on implementing support for them in an integrated development environment. The development environment should integrate component modeling with related analysis tools and with deployment (i.e. synthesis to executable code).

Acknowledgement

This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS.

References

[1] M. Åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, and M. Tivoli. The SAVE approach to component-based development of vehicular systems. *Journal of Systems and Software*, 80(5):655–667, May 2007.

[2] Arcticus Systems. Rubus Software Components. Available from www.arcticus-systems.com.

[3] AUTOSAR Development Partnership. Technical Overview V2.2.1, Feb. 2008. Available from www.autosar.org.

[4] F. Bolton. *Pure CORBA*. Sams, 2001.

[5] M. Broy. Challenges in automotive software engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 33–42. ACM, 2006.

[6] I. Crnkovic and M. Larsson. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[7] H. Fennel et al. Achievements and exploitation of the AUTOSAR development partnership. Presented at Convergence 2006, Detroit, MI, USA, Oct. 2006. Available from www.autosar.org.

[8] X. Ke, K. Sierszecki, and C. Angelov. COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems. In *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 199–208. IEEE Computer Society, 2007.

[9] F. Lüders. *An Evolutionary Approach to Software Components in Embedded Real-Time Systems*. PhD thesis, Mälardalen University, December 2006.

[10] H. Maaskant. *A Robust Component Model for Consumer Electronic Products*, volume 3 of *Philips Research*, pages 167–192. Springer, 2005.

[11] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly and Associates, 2001.

[12] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. P. Black, P. O. Müller, C. Zeidler, T. Genssler, and R. van den Born. A component model for field devices. In *Proceedings of the First International IFIP/ACM Working Conference on Component Deployment*, pages 200–209. Springer-Verlag, 2002.

[13] Object Management Group. UML 2.0 Superstructure Specification, The OMG Final Adopted Specification, 2003.

[14] Object Management Group. A UML Profile for MARTE, Beta 1, August 2007. Document number: ptc/07-08-04.

[15] D. Rogerson. *Inside COM*. Microsoft Press, 1997.

[16] D. C. Schmidt and F. Kuhns. An Overview of the Real-Time CORBA Specification. *Computer*, 33(6):56–63, 2000.

[17] Simulink, MathWorks. www.mathworks.com, accessed March 2008.

[18] R. van Ommering, F. van der Linden, and J. Kramer. The Koala component model for consumer electronics software. In *IEEE Computer*, pages 78–85. IEEE, March 2000.