A Component Model Family for Vehicular Embedded Systems

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

In this paper we propose to use components for managing the increasing complexity in modern vehicular systems. Compared to other approaches, the distinguishing feature of our work is using and benefiting from components throughout the development process from early design to development and deployment, and an explicit separation of concerns at different levels of granularity. 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, and describes a realization of such a component model.

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 tailored to perform a specific task by combination of software and hardware. In comparison to general purpose computers, one important characteristic of embedded systems is that they typically have to function under severe resource limitations in terms of memory, bandwidth and energy, and often under difficult environmental conditions (e.g. heat, dust, constant vibrations).

As embedded systems are often used in safety-critical applications, 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. The criticality of tasks performed by embedded systems also implies that they have to be thoroughly tested or better still, 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. Moreover, 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 in the vehicular domain — decomposing the functionality into subsystems that are realised by dedicated nodes with their own CPU and memory — does not scale any more due to restrictions in physical space and communication bandwidth. Instead there arises a need to collocate several subsystems on one physical unit which even more adds to complexity as resources have to be shared. All this introduces a new challenge in software development for embedded systems in vehicular domain.

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, in order to effectively employ CBD in embedded systems it is necessary to adapt it to support specifics of the embedded systems from the 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, 20]) to use CBD in embedded systems. Although these approaches were successful in solving particular pieces of the puzzle, an approach that supports the use of components throughout all stages of the embedded system development process is still missing.

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 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 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. Section 5 outlines a suitable component model family, and Section 6 discusses a realization of this component model family. Related work is described in Section 7, and Section 8 concludes the paper.

2 Motivating Example

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

The physical system architecture of a modern car consists 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.

In the automotive domain, low production cost is a very important concern, since each car model is manufactured in 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 system design (with respect to cost, but this typically means minimal in terms of resources, as well) that can provide the desired functionality with a sufficient level of dependability.

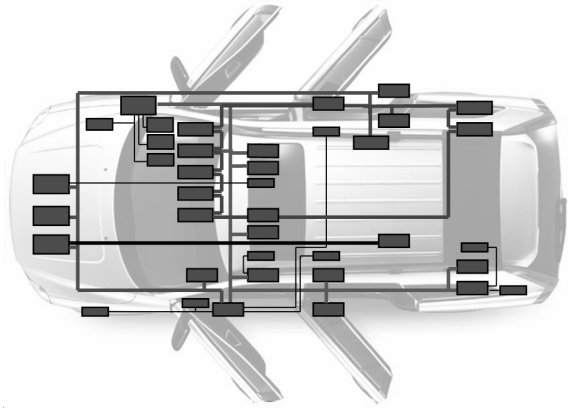

Figure 1. The electronic system architecture of Volvo XC90.

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 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 deactivated in a safe way and the driver should be informed.

Figure 2 shows the ABS subsystem architecture. In its simple form the ABS includes rotation sensors physically placed on or close to the wheels, a brake valve actuator, and an ECU that includes control software. Typically the ECU includes a set of software components that together provide the service. It is clear that different types of communication would be required between components within the ECU and between components, sensors and actuators.

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 deactivated, 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. These concerns are tightly connected to the behavior of the actual car interacting with its environ-

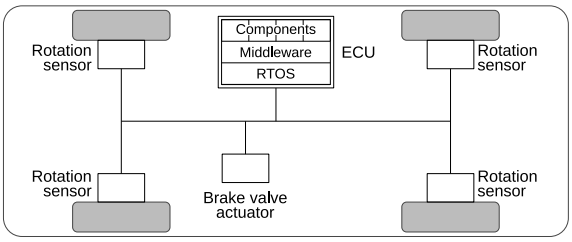

Figure 2. The ABS subsystem architecture.

ment, 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 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 communication bus. The current trend in automotive systems is towards running multiple subsystems on the same physical node, which introduces additional temporal dependencies due to scheduling.

It is common that subsystems are developed relatively independently by a few large manufacturers, who sell them to car manufacturers to be used (with some modifications) in several car brands. This brings the necessity to be able to reuse the overall design of the ABS at a level which abstracts from the interference from other subsystems 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 subsystem 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. This also calls for a separation of software- and hardware design, yet many properties of the ABS will depend on both software and hardware characteristics.

3 The PROGRESS approach

Our work on development of vehicular software 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 throughout the 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 codes of components are merged, optimized and mapped to artifacts 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

This paper concerns the component modelling aspects of PROGRESS, and thus we analyze in this section the main modelling concers with respect to early design and high level of predictability.

In a broad sense the development of an embedded system or a subsystem 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.

These two concerns, the scale from abstract to concrete and component granularity, are discussed further in the remainder of this section.

4.1 From 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 and refined 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 transition from abstract to concrete 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 (see Figure 3). 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 at 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 underlying component model to provide support for initial and abstract design as well as detailed and concrete design. An important requirement is also to provide traceability between abstract and concrete (as opposed to just having multiple 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

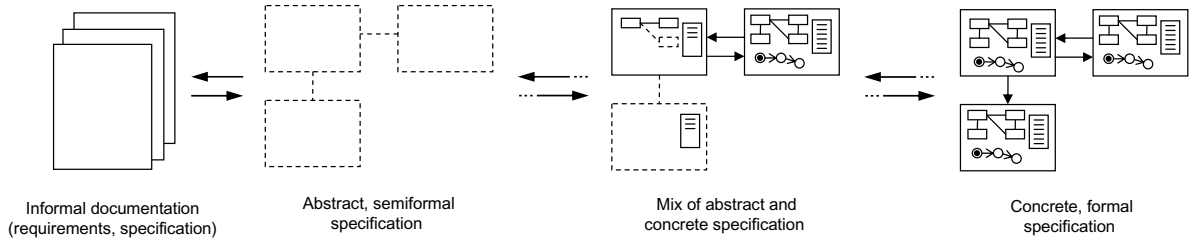

Figure 3. Development process.

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 distributed embedded system, components constituting big parts of the system are different from those responsible for a small part of some low-level control task. 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 is 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 subsystems) 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. Having these multiple levels of components it is vital to establish the relation between them, for example allowing a big component to be modelled out of several small components.

5 Conceptual component model family

Next, we present a conceptual component model family that addresses the requirements identified in the previous section. 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

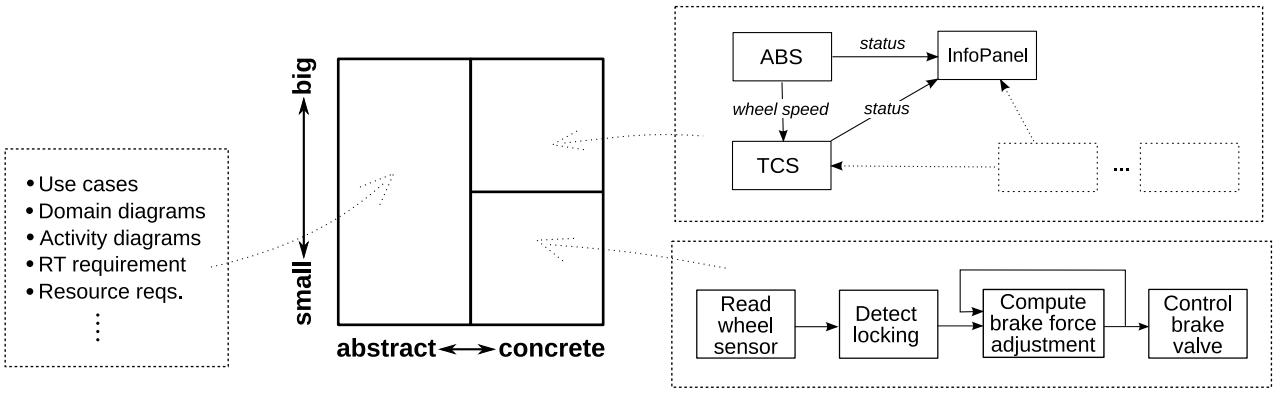

Figure 4. Proposed component model family.

two scales, this is not an easy task. Instead, we split 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 into three distinct segments is depicted in Figure 4. The benefit of this separation is that a different formalism can be used for each segment, with semantics matching the concerns of that particular level.

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, dependability and timing.

The component models used for the concrete segments 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 the most abstract point, since 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). 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.

In the component model targeting the upper level of granularity, components represent the concept of subsystems in the vehicular domain. These subsystem components are quite large, relatively independent and they are units of distribution and binary packaging. Furthermore, they can have their own threads of activity and the communication between them is realized by asynchronous message exchange, following the typical way in which subsystems are built in industry today.

A subsystem can in turn be composed out of smaller subsystems, thus forming a hierarchical component model. On the top-level a composition of subsystems forms a system, which in our case corresponds to all the software running in a vehicle.

The decomposition into smaller subsystems stops at primitive subsystem components. These can, however, be further modelled in the component model for the lower level of granularity. At that level, components serve for modelling the control logic, such as reading data from sensors, controlling actuators, etc. In this respect they provide an abstraction of the tasks and control loops typically found in control systems.

Contrasting the subsystem components, the small components are passive and do not have their own threads of activity (i.e., once invoked they run to completion). Components are composed into more complex structures by means of connections specifying the data- and control flow. This computational model, with passive components connected in a pipes-and-filters fashion, is suitable for low level modelling of embedded vehicular systems [11]. During deployment, the small components are synthesised together to make up the code of the primitive subsystem.

6 Realization of the proposed component model family

Our research so far has been primarily focused on realizing the concrete part of the proposed component family. The two models that we have developed serve here as the proof-of-concept: SaveCCM [1] and ProCom [5]. SaveCCM was sucessful in providing a solid, concrete model for low-level modelling of control logic. In particular, SaveCCM allows for timing analysis using timed automata, schedulability analysis and transformation of components to executable code.

The experience with SaveCCM has proved its applicability for small and low-level systems. However, high-level design of large distributed systems is relatively complicated — mainly due to different concerns at the higher level of granularity. That led to the development of ProCom, which follows the idea of two distinct levels of granularity.

At the lower level of granularity ProCom uses the ProSave model, which originates from SaveCCM. Among other improvements it strengthens the concept of components as reusable well-defined encapsulated units. A component in ProSave loosely corresponds to a task (in the operating systems sense) and in its simplest form it is realized by a single C function. The fairly restricted semantics of a component and the fact that the data- and control-flow is explicitly captured by connectors helps significantly in analysis and in deployment, which involves transformation of components to tasks and synthesising them to executable code.

At the higher level of granularity, ProCom relies on a newly developed model called ProSys. ProSys components are active and communicate by message passing via explicit message channels. The use of explicit message channels allows the definition of the data being exchanged together with contracts and QoS properties (e.g. stating a maximum frequency of a message, accuracy of measured data, etc.).

The two models (ProSys and ProSave) are inter-related in the way that a primitive ProSys component may be implemented by an assembly of ProSave components, thus following the two levels of granularity. However, a primitive ProSys component can also be realized by legacy code, which simplifies the transition of existing legacy systems to a component-based architecture.

With regard to the abstract part of the proposed component family, we see UML [13] and related languages (e.g. SysML [15]) as suitable candidates. A strong advantage of UML is its extensibility via profiles and a small number of restrictions in modelling. The price of using UML, which typically comes in terms of relatively informal and sligthly loose design, is more than acceptable for the abstract part. Connections to the concrete models can be established using MDD and model-to-model transformations.

7 Related work

To our knowledge, there is no approach specializing on concerns in the vehicular domain and promoting the use of components throughout the development phase. However, concentrating on individual parts of our conceptual family, it is possible to find related approaches. Some of them are reused in our solution, either explicitly or by adopting a similar strategy.

The abstract part of the abstract-to-concrete scale is connected to general purpose modeling languages such as UML [13], 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.

Detailed control functionality can also be modelled in some formalism that abstracts from the concrete system structure. As an example, Simulink [19] 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.

On the concrete side of the scale, an interesting approach focusing on "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 independent of the underlying platform and of the way functionality is distributed over the ECUs. AUTOSAR components communicate transparently regardless 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 [17], CORBA [4] and OSGi [16] — 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, 18].

Most component models that specifically target embedded systems focus primarily on "small" granularity components. Examples include Philips' Koala component model

for consumer electronics [20], Robocop [10], 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.

8 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 development phase from early design to deployment. We have demonstrated specifics of vehicular embedded systems on the ABS example, we have discussed the requirements on the CBD and outlined the family of component models supporting this CBD. We have also shown how we realize the proposed component model family. The experience we have gained so far from concretely realizing the family shows that the conceptual division of the model family significanly simplifies the use of a components throughout the development phase. Mainly because it allows using a fitting component semantics that exactly addresses the concerns in a particular stage of development.

As what regards to the on-going work, we focus on implementing IDE support for the concrete part of the component family and on using model-to-model transformations to interface with abstract modelling in UML.

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] T. Bureš, J. Carlson, I. Crnković, S. Sentilles, and A. Vulgarakis. ProCom – the Progress Component Model Reference Manual, version 1.0. Technical Report MDH-MRTC-230/2008-1-SE, Mälardalen University, June 2008.

[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] A. Möller, M. Åkerholm, J. Fredriksson, and M. Nolin. Evaluation of component technologies with respect to industrial requirements. In *Euromicro Conference, Component-Based Software Engineering Track*, August 2004.

[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, 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] Object Management Group. OMG Systems Modeling Language, V1.0, 2007.

[16] OSGi Alliance. OSGi Service Plaform Core Specification, V4.1, 2007.

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

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

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

[20] 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.