Modeling Nonfunctional Requirements: A Basis for dynamic Systems Management *

Michael Dinkel
BMW Group Forschung und Technik
80992 München, Germany
Michael.Dinkel@bmw.de

Uwe Baumgarten
Institut für Informatik
Technische Universität München
85748 Garching, Germany
baumgaru@in.tum.de

ABSTRACT

The management of dynamic systems is an upcoming challenge for software engineers in automotive and other embedded systems. The complexity of current automotive computing systems is already difficult to handle for car makers and the expected growth in the area of electronic devices in vehicles will even intensify this situation. This paper presents a model based approach for enabling automatic configuration of distributed component oriented systems. Nonfunctional requirements and capabilities of software components and platforms are explicitly modeled and provide for well-founded statements whether a component is able to execute on a certain platform or not. With application models and platform models the validity of a configuration is defined in this paper. The models even allow reconfigurations based on information regarding the actual system context like user behavior, backend or environmental sensor information.

Keywords

Component, Configuration, NFR, Nonfunctional Requirement, Capability, System Management, Software Management, Application Model, Platform Model

1. INTRODUCTION

In the automobile industry the development of electronic control units (ECUs) is currently done with multiple suppliers for each new model. The job of the vehicle manufacturer is to specify the requirements for each ECU and subsequently integrate these black box modules into one system. Most ECUs are currently static systems and use static operating systems like OSEK. As a result software changes are difficult to handle and complete ECUs need to be flashed in order to update a single feature or to fix a single bug. Since this is an obvious problem the OEMs[1] tend toward a platform strategy. AUTOSAR [11] for example is an approach that aims at modeling the communication dependencies of software components and to generate the necessary middleware. This is a big step towards an improved handling of complexity but still results in a statically defined runtime environment (RTE). Currently modularity is only available at source code level like done in OSEK with the modules of applications, operating system, communication layer and drivers still compiled to one big binary per ECU. In contrast to conventional methods AUTOSAR will allow to define functionality spanning multiple ECUs.
Newer developments introduce runtime platforms and frameworks like Java/OSGi into vehicles and other embedded systems. These runtime platforms enable modular software exchange which may even happen at runtime. In the future such platforms will not only be present at the head-unit of an infotainment system, but also in other more powerful ECUs like domain controllers. Each of the different platforms is able to install, update and remove software components at runtime. This paves the way for more dynamic and more up-to-date systems in modern vehicles. One issue that gains more and more importance is the configuration of the vehicle computing systems. Configuration means to determine a layout that describes which software component will be executed on which platform in the distributed system. While such systems are currently configured manually once in the development phase of a new car series we generally avoid major changes in the configuration later on since the consequences can be hardly predicted. With the upcoming dynamics of vehicular software systems a lot of additional use cases like short life-cycle software and frequent security updates become achievable and necessary. Even new business models like rental of software are thinkable. In the evolving scenarios configuration is an essential aspect of software and system management. With this paper we present a model based approach for enabling automatic configuration in dynamic systems.

The remainder of this paper is organized as follows: in section two we will introduce the general approach for configurations and define *functional* and *nonfunctional requirements* as well as *capabilities*. Section three presents the meta

[1]Original Equipment Manufacturer

models for *application models* and *platform models* and defines the communication flow between components. In section four configurations are addressed. The validity of configurations is defined on the basis of the previously presented models. Additionally we explain how a dynamic reconfiguration can be based on ratings of *criticality* and *utility*. We present use cases that benefit from the described models in section five while an overview over related work is given in section six. Section seven concludes and points out further work.

2. FUNCTIONAL AND NONFUNCTIONAL REQUIREMENTS

The configuration of a system defines the mapping of software components to nodes or platforms. Our approach for determining whether a component can be mapped to a certain platform is based on functional and nonfunctional requirements. In this paper we will focus on the requirements of components not the requirements of users. Functional requirements (FRs) define what is necessary for a piece of software to perform its work. That means a component needs its data inputs in order to be generally operational. The functional requirements of a set of components can be modeled as a data-flow graph that defines which component uses whose outputs as input data. The nonfunctional requirements of a software component define all other requirements like the need for encrypted communication or redundancy requirements. This is what current approaches to reconfiguration lack.

2.1 NFRs and Capabilities

In order to carry out a configuration that is mapping a set of components to a set of platforms, we need to make sure that, among other things, all nonfunctional requirements of each component are met by the assigned platform. Therefore we need to explicitly model the nonfunctional requirements. Once the NFRs are modeled we become able to check the NFRs against a counterpart in order to verify the assignment. As can be seen in figure 1 both NFRs and capabilities are a special kind of property. Properties offer a name and a value that can be used to characterize entities.

Capabilities are defined as a subclass of property, so they offer a name and a value. In addition to that capabilities may provide further properties that can be used to precisely specify a capability. In this way we can describe the features of a capability. This is very useful if we want to specify a platform that offers a special hardware like a certain sensor for example and we also need to specify the location of this sensor. So capabilities enable a detailed description of what kind of nonfunctional services are offered by an entity.

The base class of nonfunctional requirement (NFR) is conditional property which itself inherits property. As depicted in figure 1 the conditional property adds a condition to the property. Such a condition could be "*greater than*", "*less or equal*" and so on. At first a NFR is a conditional property, also called primary conditional property, that defines the concrete needs of an entity. Each NFR may have further conditional properties in order to accurately characterize the NFR. With NFRs we can specify for example that an entity needs at least a certain amount of memory.

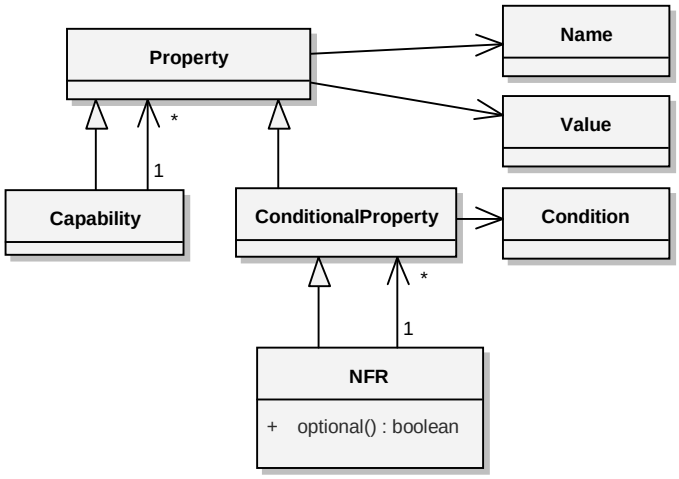

Figure 1: Nonfunctional requirements and Capabilities

3. MODEL-BASED CONFIGURATION

Our approach for creating configurations is based on two types of models for describing the counterparts that need to be mapped together in order to produce a working system. *Application models* define the software components and their interactions while it is the job of the *platform models* to specify the layout and capabilities of physical nodes and their network connections.

3.1 Application Model

The models that define the inter operating software components are called application models. They define the software components and their *functional* and *nonfunctional* requirements. The functional requirements are embodied in the communication relations between the software components. They are modeled as a kind of data flow diagram that defines the routes that data items take from one component to the other. Data flow diagrams are a frequently used means to describe the relations between components in distributed systems [2]. A representation of nonfunctional requirements is what lacks in usual data flow diagrams. So in application models the nonfunctional requirements (NFRs) are represented as a kind of features of model elements. As depicted in figure 2 all application model elements may have nonfunctional requirements as well as capabilities in the way they have been defined in figure 1. Model elements of the application model can either be components, ports, channels, applications or systems. The model elements are defined in the following way:

A software ***component*** *is a unit of decomposition with contractually specified interfaces [10]. The interior of a component is not visible or accessible from outside of the component. A component uses InPorts and OutPorts to communicate.*

A communication ***channel*** *connects two or more software components in an publish/subscribe manner. Channels represent the flow of information between the software components. Components use InPorts and OutPorts to connect to channels.*

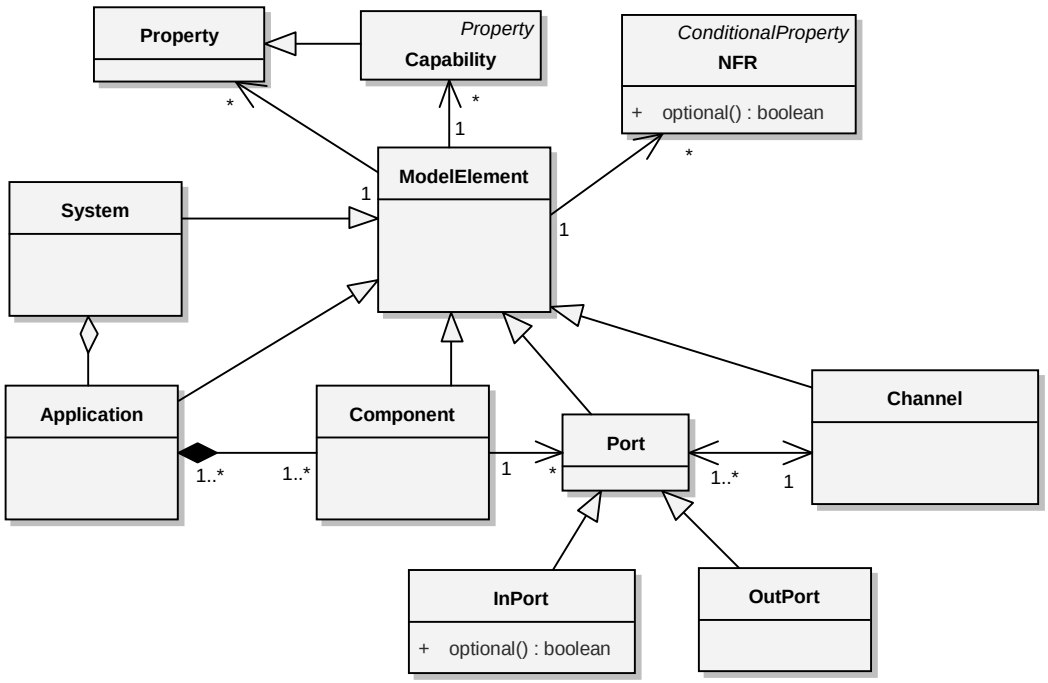

Figure 2: The meta model of application models

*An **application** is a set of communicating components that provide coherent, user-perceivable functionality. Applications are not necessarily disjoint. That means certain components may be part of multiple applications.*

*The **system** is an aggregation of applications.*

3.1.1 Rating model Elements: Criticality and Utility

At runtime when one of the networked nodes fails it may happen, that the resources available are no longer sufficient for all the software previously installed in the vehicle system. A reconfiguration becomes necessary that configures the remaining system in a way that the available hardware is used economically and that the important applications and components remain functional. Therefore we need a possibility to determine which components are more important than others and hence have to run in any case. This could be done by additional properties. Such properties can be attached to elements of the application model as shown in figure 2. In literature like [8] a utility value is defined for certain components, named features. Their utility value is statically defined and allows to choose a configuration by maximizing the overall system utility.

We chose a more flexible approach since a static assignment of importance seems to be too restrictive. We have two properties, named *utility* and *criticality* that address a similar but not identical purpose. The value for utility can be adapted at runtime while criticality is a predefined value by the car maker and has always higher priority when it comes to a reconfiguration. Allowing runtime changes on one of these two values enables the system management to base a reconfiguration not only on statically defined ratings. But the current environment and the user behavior and preferences can also be accounted for a reconfiguration. Among others the following adaption mechanisms are possible to change the utility rating of an application model element:

- *No adaption:* This would result in a static rating that does not change at runtime.
- *Usage statistics:* Information about what applications are used how much would make reconfigurations adaptable to the users behavior and preferences so that a user could still use his personal favorite applications even in the face of failures.
- *Environmental information analysis:* Environmental information like temperature or weather forecast could be used as input for the utility value. For example temperature below zero degrees Celsius and a black ice warning could result in high utility value for a DSC[2] component even though it has not been used recently.

3.1.2 Communication style

As defined in figure 2 components use channels for communication. They send data via OutPorts and receive data via InPorts. Each port has exactly one channel while a channel can transport data from multiple InPorts to multiple OutPorts in M-to-N style. That means if one component sends data to a channel all other components will receive the data if they have InPorts to this channel. This kind of communication has been chosen because of several reasons:

- Data-flow diagrams are often used to specify embedded systems but the usual 1-to-1 data flow is just a special form of our M-to-N style.

[2]Dynamic Stability Control

- Channels serve as indirection mechanism that enables loose coupling between components since they do not need to know their communication partners.

- The indirection allows isolated specification of components. The communication ability of a component can be specified by it's InPorts, OutPorts and the connected channels without involving other components.

- A flexible exchange of components is made possible by the indirect M-to-N communication. With conventional data flow diagrams a component can only be exchanged by another component that provides exactly the same data. If the required data is produced by a different component after a modification of a few components the complete data flow diagram changes. Our indirect communication keeps the changes in the diagram very local and hence eases the handling of runtime dynamics.

3.1.3 *Examples for NFRs*

In order to clarify the notion of NFRs let us consider the different elements of the application model meta model and their need for nonfunctional specifications.

- Software components have to specify their runtime environment, like required CPU-performance, main memory, persistent memory, availability of special hardware.

- Ports are targets of NFRs that define the nonfunctional aspects of input or output of data. OutPorts may specify the bandwidth, communication characterization (cyclic, event-triggered, streaming...) and latency (time between sending and actual transmission).

- Communication channels accumulate the NFRs of the connected ports. So a channel will need at least the sum of the bandwidth of all OutPorts sending data over it. Additionally data encryption may be a NFR of a channel.

- Applications (sets of components) usually define targets for non-functional requirements. For example it makes no sense if only a few components of an application are deployed redundantly, but some parts of the application remain as single point of failure. Each application in the model can be characterized by nonfunctional requirements that count for the whole application like for example a certain software integrity level (SIL) as defined by IEC 61508 [4].

- A system as a whole may also be target of nonfunctional requirements like overall flexibility or other global NFRs.

All the before mentioned NFRs have one thing in common, that is that they have to be known in advance in order to be useful for our approach. This is especially difficult since worst-case analysis is known to be a hard thing.

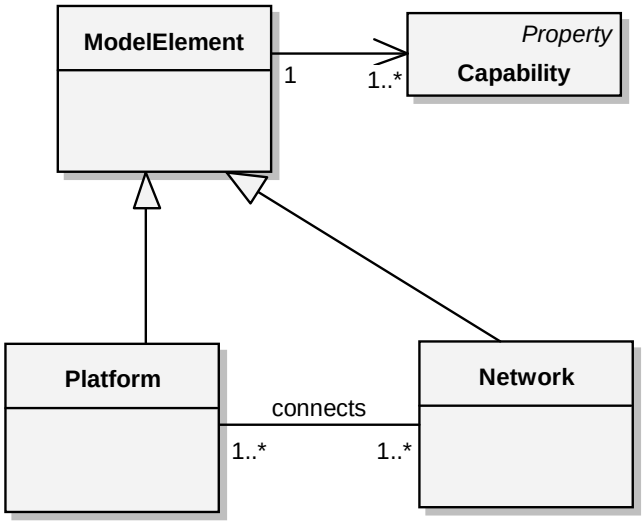

Figure 3: The meta model of platform models

	Platform	**Network**
Infrastructure Layer	OS, Drivers, Communication	Protocols
Hardware Layer	CPU, Memory, Special Hardware	Physical Medium, Topology

Table 1: Layers of a platform model

3.2 Platform model

Platform models specify the counterpart of application models which is needed since we want to map software components to platforms. Each platform model consists of platforms connected by networks. The meta model of platform models can be found in figure 3.

Each platform is able to execute software components as described by an application model. A platform is defined as a hardware node that runs a certain stack of basic software. The basic software is an essential part of a platform and includes at least a runtime environment and a communication infrastructure. Furthermore we expect the components to "prepared" for executing on the platforms, which includes that components with need for certain capabilities (expressed as NFRs) are able to make use of the provided capabilities. The terms of platform models are visualized in table 1 and are used in the following way:

- *Platform:* A platform contains hardware with some infrastructural software running on it.

- *Network:* A network contains the physical medium, a topology and a stack of protocols. (Topology is part of network because protocols have to do the routing and hence have to know about topology.)

The platform-model is needed to give the configuration software all necessary information about the capabilities and topology of our target layout. The hardware layer, composed of hardware and physical network wires, exposes certain capabilities in terms of CPU performance, memory size

and network throughput. On the one hand the infrastructural software consumes a certain amount of these capabilities: of course a CPU has more reserves with no software running by default. On the other hand not all capabilities in the platform-model have their origin in the hardware layer. The infrastructural layer also provides additional capabilities like security or reliable communication via certain network protocols. Also reliability is a NFR that depends on both, the hardware and the infrastructural software of a platform.

4. CONFIGURATIONS

Configurations describe the mapping of software components to the different platforms in the distributed system. This is the distribution of the software modules to the networked nodes of the platform model. We can distinguish two general kinds of configurations:

*A **valid configuration** is a configuration that fulfills all of the requirements stated in the application model including both, functional and nonfunctional requirements. There may be several valid configurations for the same platform model and application model. A valid configuration has to be complete, which means that all elements of the application model are mapped to elements of the platform model.*

*An **optimal configuration** is a valid configuration that has been additionally optimized for a specific optimization goal. Optimization goals may be reduction of network traffic or reliability and so on. For each optimization goal there is only one optimal configuration.*

4.1 Validity

Before we start to optimize a configuration we have to secure a configuration is valid. Valid means all requirements of the components in the system are met. In this consideration we take for granted that the functional requirements are met which can be validated by checking the communication paths between the assigned components. Even though functional requirements are a difficult problem in general, at the current state we assume a communication model that allows for the validation of functional requirements. For future work we will have to consider appropriate methods for the verification of functionality, for instance assume/guarantee reasoning as described in [3].
For the nonfunctional requirements we can decide this because both nonfunctional requirements and capabilities have been explicitly modeled. A nonfunctional requirement for a target is met if the following validity condition (1) holds. The condition of each NFR and conditional property is noted as $\succ$. The condition of an NFR is used to compare the value an NFR with the value of a capability in case their names match. So evaluating a condition will either result in *true* or *false* which can be summed up by a logical AND.

Validity condition for one NFR:

$$VC_N = (CV \succ NV) \wedge \bigwedge_{i=1}^{n} (CV_i \succ_i NV_i) \qquad (1)$$

In (1) we have defined the validity condition for one NFR (VC_N), by checking whether the value of the capability (CV) holds for the value of the NFR (NV) under the specific condition ($\succ$) as defined by the NFR. We form the logical AND of the primary NFR condition and all its (up to n) conditional properties characterizing the concrete NFR. Based on (1) we can move forward and define validity for one assignment of an application model element to a platform platform model element.

Validity condition for one assignment:

$$VC_A = \bigwedge_{i=1}^{n} (VC_{N_i}) \qquad (2)$$

The validity condition for the assignment of an application model element to a platform model element (VC_A) is *true* if the logical AND of all validity conditions (VC_{N_i}) that belong to that application model element is *true*. With the validity condition for one assignment defined in (2) we can have a look at the validity condition for a complete configuration.

Validity condition for a configuration:

$$VC_C = \bigwedge_{i=1}^{n} (VC_{A_i}) \qquad (3)$$

$$\forall\ ae \in AM\ \exists\ pe \in PM\ with\ ae \text{ is-assigned-to } pe \qquad (4)$$

For a configuration to be valid it takes more than defined in (3) that all assignments are valid. Moreover it is required that all application model elements are assigned to platform model elements as stated in the completeness constraint (4). Here we have application model elements (ae), platform model elements (pe), the application model (AM) and the platform model (PM).
At the first glimpse the previous validity conditions may seem to induce a very pessimistic ressource allocation which is only appropriate for highly critical applications. In fact this is only the case for very stringent NFR conditions like `">="`. For less critical applications we could think of NFRs with conditions that result in a more optimimstic assignment of capabilities. Nevertheless the validity conditions defined above are independent from the conditions of NFRs and conditional properties. With these validity conditions we have a means to determine whether a configuration is valid or not. This can be of great benefit for several use cases as described in section 5.

4.2 Dynamics of models

The NFR-based modeling is targeted at dynamic systems. This means software (elements of the application model) can be added, removed and updated at runtime. Additionally hardware (elements of the platform model) can be added and removed. Hence the models themselves and the capabilities of platforms and networks are subject to change at runtime. By software components located on a certain platform the capabilities like `cpu_rating` and `memory` of this platform are generally reduced since the components consume them. The kind of reduction however depends on the nature of the capability. Obviously the available memory or bandwidth is reduced by the exact amount a component uses. On the

other hand the reliability of a network is generally not reduced only because there is a little traffic on the network. So we have to distinguish between different consumption behaviors. It is also possible for a software component to add additional capabilities to a platform since components may have capabilities on their own. Each time a component that provides capabilities is located on a certain platform, the component's capabilities are accounted to the capabilities already available at the platform. So the number and values of capabilities of a platform can grow and shrink during the life-cycle of a system.

4.3 Optimization

When a mapping of software components to platforms (a configuration) is performed we can try to realize certain optimization goals. The most important goal is usually to be as cost efficient as possible. Since this is a very general optimization goal we define more concrete goals that can be determined when monetary values are not directly modeled. A typical optimization goal that aims to cost reduction is to optimally utilize the available resources. In order to efficiently utilize the available resources we would like to leave only a minimum of unused computing power and resources on each platform of the platform model. Another idea is to keep network communication as local as possible. This means that it is preferable to locate two communicating software components on the same host or same subnet instead of distributing them further. Every communication channel via gateways potentially increases the traffic in the gateway what in the end may lead to the need for more powerful hardware. So a optimization aim could be to minimize the number of communication channels that cross networks and especially gateways. This locality aim can be even increased when we try to co-locate communicating software components at the same node.

5. USE CASES

In this section we will point out a few use cases where the availability of a system description can result in big benefits. This is especially true for complex embedded systems where customers do not primarily see computers but other everyday equipment like vehicles. In cars we encounter a more or less unique situation: we have very advanced technology and users with nearly no IT-understanding. In-car computing systems consist of a great number of control units that include very special computing equipment, sensors and actuators. The ECUs are connected by a number of different communication systems. All in all we have very high reliability requirements and a back-breaking cost pressure. On top of this for the future we will experience even more functionality and innovations based on information processing technology. On the other side from the view of a vehicle user the complete system is "just" a car. Users of cars do not accept abnormal ends of control or entertainment functionality as they would do for personal computers. They don't want to bother about computer problems in their car, they just want it to work. For the area of vehicle applications there are several use cases all over the life-cycle of a vehicle that would benefit from the presented way of system description.

- *Initial configuration:* In the development phase a configuration management can help the developers of the automotive system to optimally layout the software components, to reduce cost and to optimize resource utilization.
- *Update in the field:* Software updates in the field are quite common these days since vehicles have become so complex that there will always be security updates or improvements. We can easily check whether an altered software component is still executable at the previous platform and have the concrete restrictions for each component modeled explicitly.
- *Installation of individual applications:* In the future customers will want to upgrade their in-car computing systems. So these systems need to be flexible and upgradeable both in software and hardware. With the described models we enable a system to evaluate whether a new software component can be executed on the current platforms or whether we need to upgrade the hardware as well. Also for the removal of certain software components we faciliate to make statements whether the system can execute without a certain component, or if not, which other components will be affected in what way.
- *Update, exchange of hardware:* We can easily determine whether a hardware layout modeled in a platform model is sufficient for certain applications. Also reconfiguration and optimal use of resources can be reached.
- *Error recovery:* For purposes of error recovery we can use the application model and the criticality, utility rating to decide which applications should be run in case of reduced resources. By explicitly modeling the capabilities and requirements we pave the way for self-healing systems, that can provide the best configuration for each customer.

6. RELATED WORK

In this section we present related work. Beus-Dukic motivates the need for NFRs [1] in the field of commercial off-the-shelf components (COTS) and defines NFR's informally as the general qualities of a software product.

6.1 Requirements and capabilities

The term requirements is generally used to state the needs of an entity that have to be fulfilled by another one. Especially in the field of computer systems this may happen at different levels. From a development process point of view one or many real-world problems have to be solved and hence have requirements that a system has to fulfill. Requirements define the key demands a software system has to accomplish. These requirements are mostly stated in form of human readable text, rarely in a mathematical formal way. At the level of abstraction of interest to us, we consider components and applications to be the entities that pose requirements. The requirements have to be fulfilled by platforms and communicating sets of platforms. The requirements at different levels of abstraction have a different degree of freedom. At the component level the range of possible requirements is more restricted and much more concrete, compared to the kind of requirements we find at

the beginning of a software development process. The IEEE Std 1233 IEEE Guide for Developing System Requirements Specifications [12], defines requirements as:

(a) A condition or capacity needed by a user to solve a problem or achieve an objective;

(b) A condition or capability that must be met or possessed by a system or system component, to satisfy a contract, standard, specification or any other formally imposed document;

(c) A documented representation of a condition or capability, as defined in either (a) or (b)

6.1.1 CC/PP

With the Composite Capability/Preference Profiles (CC/PP) the World Wide Web Consortium (W3C) has issued a recommendation that describes how to achieve device independence for mobile clients. The recommendation [5] considers capabilities of terminal equipment in order to achieve an optimal adoption of content for a wide range of devices. CC/PP is designed to be a XML based format for describing the capabilities of user agents in order to enable servers to provide content that is tailored to the capabilities of each device. The general structure defined by CC/PP is a two level tree consisting of components and attributes. Attributes have a name and a predefined set of values. A CC/PP profile contains multiple components and each component may have multiple attributes. As the name CC/PP indicates the same mechanism and XML scheme can be used for capabilities and preferences, both with the aim of delivering specially tailored content. In contrast to the work presented in this paper the CC/PP approach only addresses one side of the challenge: the description of capabilities. The recommendation assumes that servers know about their content and hence does not provide a description of content requirements.

6.1.2 Process^{NFL}

In [9] a language for describing nonfunctional aspects of software has been designed. Process^{NFL} is aimed at the software development process, to support different levels of abstractions for NFRs, their relations, conflicts and non-direct implementation nature. The notion of NF-Attributes is used to describe NFRs in a hierarchical way. NF-Attributes may either be directly realized by NF-Actions or may be affected by them in either a positive or negative manner. NF-Properties allow to impose constraints and priorities on NF-Attributes. Compared to our approach Process^{NFL} is a more general approach that is suitable for different steps in the software development process. The counterpart of our NFRs are NF-Attributes which form a qualitative view on the requirements while NFRs are explicitly designed to quantitatively state the needs of application model entities. Each NFR has its counterparts in a certain capability identified by its name. The NF-Attributes of Process^{NFL} can be either implemented by one NF-Action or if it cannot be directly implemented there may be multiple NF-Actions that affect the NF-Action in a positive or negative way. The overall aim of Process^{NFL} is different since our model based approach with NFRs and Capabilities is specifically aimed at providing a means for describing software in a dynamic system and not to provide a general approach for describing nonfunctional aspects at different levels of abstraction in the development process.

6.2 Reconfiguration

With the aim of graceful degradation in embedded systems a team around Philip Koopman has built up a system based on a software product family approach. In the RoSES project software components are modeled in a data flow diagram and so-called features have a statically defined utility value. The RoSES approach [7] however does not address exact modeling of requirements and capabilities so far.

6.3 Automobile Software

In the automotive area the AUTOSAR consortium [11] is working on an open standard for an automotive E/E architecture. The main aims are to achieve modularity, scalability, transferability and re-usability of functions by providing a common software infrastructure based on standardized interfaces. AUTOSAR defines a standard for component interfaces and is also confronted with the problem of locating software components on nodes (ECUs). For this purpose three descriptions are used, namely the ECU resource description, the software component description and the system constraints description. While our approach is targeted at dynamic systems and will serve as a basis for decision making at runtime AUTOSAR is completely static and has to solve the configuration problem only once in the development cycle. There definitly is some related work here, however the AUTOSAR specifications are not available for public yet.

7. CONCLUSIONS AND FUTURE WORK

With software systems of vehicles and other distributed embedded systems getting more and more complex in development and maintenance, this paper provides a model based approach for enabling automatic software management in dynamic systems. By modeling nonfunctional requirements and capabilities we can determine valid configurations and provide a basis for reconfiguration and the optimization of configurations. The application model with the notion of criticality and utility can even help us to determine configurations that are specially adopted to the current context of a system. Benefits of the presented approach are expected for updates and changes of hardware and software of a system in the field and in the area of error recovery.

As the modelling of requirements and capabilities is only in the beginning, additional points have to be investigated. One of the major challenges for a prototypical implementation is the development of a way to measure and describe the perfomance needs of software components which is of course platform-dependent. We have to consider this especially for automotive systems that consist of many different computing architectures. On the way towards self-healing and self-organizing there will be further work to be done in the area of end-to-end real-time requirements and we will have to examine how well known techniques like rate monotonic analysis (RMA) [6] can be used in a dynamic environment. Also the management functionality for reconfigurations and the optimization of configurations has to be addressed by further work.

8. REFERENCES

[1] L. Beus-Dukic. Non-functional requirements for COTS software components. In *Proceedings of ICSE workshop on COTS Software*, 2000.

[2] T. DeMarco. *Structured analysis and system specification*. Prentice-Hall, 1979.

[3] T. A. Henzinger, M. Minea, and V. Prabhu. Assume-guarantee reasoning for hierarchical hybrid systems. In M. D. D. Benedetto and A. L. Sangiovanni-Vincentelli, editors, *Hybrid Systems: Computation and Control (HSCC), 4th International Workshop*, volume 2034 of *LNCS*, pages 275–290. Springer-Verlag GmbH, 2001.

[4] M. Kieviet. Anwendung der IEC 61508 im Automobil. *Elektronik Automotive*, 1/2004:49–53, 2004.

[5] G. Klyne, F. Reynolds, C. Woodrow, H. Ohto, J. Hjelm, M. H. Butler, and L. Tran. Composite capability/preference profiles (CC/PP): Structure and vocabularies 1.0. W3C recommendation, W3C.org, January 2004.

[6] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[7] W. Nace and P. Koopman. A product family approach to graceful degradation. In *International Workshop on Distributed and Parallel Systems (DIPES2000)*, Oktober 2000.

[8] W. Nace and P. Koopman. A graceful degradation framework for distributed embedded systems. In *Workshop on Reliability in Embedded Systems (in conjunction with Symposium on Reliable Distributed Systems/SRDS-2001)*, New Orleans, Oktober 2001.

[9] N. S. Rosa, P. R. F. Cunha, and G. R. R. Justo. PROCESS^{NFL}: A language for describing non-functional properties. In *Proceedings of the 35th Hawaii International Conference on System Sciences*. IEEE - Computer Sciety, 2002.

[10] C. Szyperski, D. Gruntz, and S. Murer. *Component Software - Beyond Object Oriented Programming*. Addison-Wesley, second edition, November 2002.

[11] AUTOSAR development partnership. Automotive open system architecture. http://www.autosar.org, 2005.

[12] IEEE Standards Organization. IEEE guide for developing system requirements specifications, 1998.