UNIVERSITY OF GOTHENBURG

Exploring variation mechanisms in the automotive industry

A case study

EMIL M. JANITZEK
MARCUS P. LJUNGBLAD

Bachelor of Software Engineering and Management Thesis

Report No. 2010:008
ISSN: 1651-4769

University of Gothenburg
Department of Applied Information Technology
Gothenburg, Sweden, May 2010

Exploring variation mechanisms in the automotive industry: a case study

Emil Janitzek
emil.janitzek@quandoo.se

Marcus Ljungblad
marcus.ljungblad@quandoo.se

ABSTRACT

Today car manufacturers are expected to deliver cars configured for each customer. Through software, and by adapting software product line methods, car manufactures respond to the increased customization needs. This emphasizes the need for careful variant handling. Thus, based on a problem definition from Volvo Cars this case study explores variation techniques to support massive numbers of built-to-order cars using AUTOSAR. *In essence, this study argues that run-time variability, tested with a prototype development, is a way to meet this need. It establishes the publisher-subscriber pattern as a viable run-time variation mechanism, and identifies limitations and areas to consider related to subscription management, real-time performance and data transparency within an automotive environment. Finally, this study also demonstrates how run-time variability enables manufacturers to better support after-market services and enable 3rd party integration.*

Keywords: *AUTOSAR, Volvo Car Corporation, Variability, Software product lines*

1 INTRODUCTION

For a car manufacturer like Volvo Cars, producing 400 000 customized cars each year, it is essential to have access to a range of techniques to meet customer demands and to support after-market services (Weiss et al. 2009). For instance, each manufactured car is designed with the customer's preferences of stereo, engine capacity and appearance. Since it is impossible for manufacturers to anticipate all configurations at design-time and since they have limited access to the car once it has left the factory, some choices about how the software operate must be delayed. The concept of software product lines was introduced to increase the ability to re-use software and to meet customization requirements (Bosch 2000). Software product lines focus on the methods and tools required to create similar products based on a collection of software assets. One of the vital aspects in software product lines is the management of variants, or in other words, the combination of assets to form a single, possibly unique, product. The automotive industry, which has a long tradition of using product line manufacturing, has over the past years been adopting similar techniques for their software development (Broy 2006). Effectively, this means manufacturing organizations are transforming to software organizations, relying on effective use of software methods.

As a consequence of this transformation, the organizations are increasingly concerned with variability management. A number of variability patterns and mechanisms have been suggested by researchers applicable from system design time to run-time (Bachmann & Bass 2001, Svahnberg et al. 2001, Van Der Hoek 2004, Bachmann & Clements 2005). However, extant research overlooks variability within industry standards. Solutions for variability are created by manufacturers and suppliers independently, which forces manufacturers to carefully design the requirements when outsourcing sub-system development. For many automotive manufacturers the AUTomotive Open System ARchitecture (AUTOSAR) standard is currently being integrated into production (Fürst 2009), and the latest version 4.0 supports only limited variant handling as will be illustrated in the following sections.

Creating a unique design for each manufactured car is obliviously unrealistic. Thus, architectures with clear variation points are used. Based on a case study from Volvo Cars this paper argues for run-time variation as one way to support the massive numbers of configurations. Specifically, our study elicits the key elements required to support run-time variability using the AUTOSAR 4.0 standard. We chose to develop a prototype of a run-time variation mechanism due to the lack of current research on variability in AUTOSAR, as well as for the importance of having a full range of variability options in software product lines to chose from.

Subsequently, this study contributes in three ways. One, we illustrate that run-time variability is possible using the AUTOSAR standard - something which is likely to improve after-market services, 3rd party integration and increase software re-use. Two, we provide hands on details of how to in practice translate the standard guidelines into an actual prototype. Three, we extend the variability catalog by reflecting on the implications our findings have on current research.

2 RELATED WORK

Variant handling is the ability to modify a system without making a big impact on the system or imposing a need to restructure the design (Svahnberg et al. 2001). It is a part of the foundation of software product lines. To conceptually understand the variations, lets look at an example. Car manufactures today want to use the same software, or as similar software as possible, in each car model even though they differ from each other. The industry often talks about families of cars; cars which originates from a common platform, including both hardware and software. For example, the inside lighting system depends on if the model has two or four doors. With different car models, different doors are used with different lights. The lighting system needs to know which type of doors are installed in a particular car so that it can switch on the right lamp at the right time. This variation, variation in technology as Bachmann & Bass (2001) calls it, must be described, linked to requirements, as well as realized using a variation mechanism sometime during development. However it does not end there, each car within a family can be adapted individually this creates higher demands on the variation technology in place.

In the following sections we outline how run-time variation is related to other types of variation mechanisms. Based on existing literature, we also motivate what types of variation are usually dealt with in run-time, or in other words, why run-time variability is preferred in some cases. Thereafter, we provide a summary of the variation mechanisms that are available in AUTOSAR. Finally, we outline a theoretical framework for analyzing variation mechanisms.

2.1 Software product lines

With the introduction of software product lines Bosch (2000) realized the need to reflect variability in architectures. Variability is defined in the architecture through variation points (Bosch

Table 1: Variability overview

Variability Pattern	Examples of Mechanisms	Binding time	Usage
Product Architecture Derivation	Configuration management, Generators	Pre-build	Implementations during the architecture and design phase
Compilation	Compiler switches	Pre-build	Compiler flags will resolve to one binary output
Linking	Binary replacement	Pre-build	Linkage with library/binaries to produce one binary output
Runtime	Adaptation during start-up, condition on variable	Post-build	Uses inline code to resolve variability at runtime

et al. 2002), or a specific place in the architecture at which a feature can take one of two or more shapes. In general, variability means the "ability of a system, an asset, or a development environment to support the production of a set of artifacts that differ from each other in a preplanned fashion." (Bachmann & Clements 2005). According to Bachmann & Bass (2001) variation points are in direct relation to requirements made by the customer and should be documented in the architecture.

Bachmann & Bass (2001) outlines six sources for variability: in function, in data, in control flow, in technology, in environment and in quality. Particularly interesting from a run-time perspective, are variation in data and in control flow. In control flow, the application executes differently depending on, for example, available components or settings passed to a module designed to handle variability. Variation in control flow also encompass unknown variation, such that when variables are distributed to multiple components, some which may be developed by 3rd party suppliers, the execution path cannot be determined beforehand. The authors continue to state that adaptation during startup, by reading a configuration file, is a common way of managing variation. Software designed with runtime variability, like adaptation during start-up, must implement all possible variations. This means that the object code will contain all variations.

In the next section we outline common mechanisms organized into four patterns. Several authors describe similar mechanisms but often describe these with varying terminology and detail. We provide an overview of variation mechanisms available as a complement to run-time variability.

2.2 Existing variability patterns

Even though many agree that planning for variability is important, doing this in a unified manner can be hard. Bachmann & Bass (2001) and Svahnberg et al. (2001) represents two attempts to categorize and characterize archetypical approaches for variant management. When looking at variability, one of the most crucial aspects is when the actual variant will occur. This is referred to as variation points. A variation point is a representation by the designer to delay the design decision (Bosch et al. 2002). The point in time when the design decision has been made is referred to as binding time. This is when the actual variant becomes bound, after this point only one implementation is used. Hence, the chosen variant will exist in the system. For variation in run-time, this means that the variation point is in run-time and the binding of that variation point happens during application execution. Not until the data, for example a configuration variable, is passed to a function is the variation bound.

A pattern describes where and how a variation is implemented (Bachmann & Bass 2001). Several mechanisms can be categorized into one pattern. Svahnberg et al. (2001) uses a different terminology and focuses on a lower abstraction level. They focus on implementations, hence the example mechanisms in Table 1. Where to implement the variation is decided by the architect, and this decision will affect the structure of the software from design to execution. However, most mechanisms apply at pre-build time which is natural considering variability stems from requirements or feature diagrams (Bachmann & Clements 2005). The ability to vary decreases the closer to the final product you come since "variation points are bound to selected variants" (Bosch et al. 2002). In this article we will referrer to the patterns defined by Bachmann & Bass (2001) while using the more detailed description of mechanisms provided by Svahnberg et al. (2001). This decision was made since Bachmann & Bass (2001) provides a clear and consistent way when defining the main variability patterns while Svahnberg et al. (2001) goes into the details in different variation mechanisms.

As described in Table 1 there are four main variability patterns that all represent different stages during software development.

- **Product Architecture Derivation**
  The earliest variation point available is during the architecture and design phase. Planning for variability may impose the use for configuration management systems or the need to create generators for product specific behaviors, but this can be both time consuming and costly (Bachmann & Bass 2001). A generator can be used for creating a product specific architecture which only covers the needs for the current product. In the same way a configuration management system will create a specific product by only including the desired modules, this makes it easy to handle different variants within the same configuration management tool. Equally, planning for variability is one of the biggest factors for a successful software product line (Bosch 2000). Doing so well will decrease the risk of problems during development.

- **Compilation**
  During the compilation of software different version can be compiled depending on flags sent to the compiler. This method is commonly used to, for example, produce platform specific binaries, primarily since it requires little effort and is easy to use. However, too many directives can impede on code comprehensibility (Spencer 1992).

- **Linking**
  Similar to the compilation pattern; linking will result in one product specific binary by linking the correct packages/libraries together with the object-code generated after compilation.

- **Runtime**
  Runtime can be further refined into two phases, adaptation during startup and adaptation during normal execution (Bachmann & Bass 2001). Variation points introduced in runtime often depend on the ability or access to the system once deployed.

Van Der Hoek (2004) postulated an alternative approach to variability points. He takes the concept of variability further such that it can be achieved at any time during development. This is achieved by inserting a unbound variation point, instead of choosing one of the variation points as defined by Bachmann & Bass (2001). This gives the ability to change the variability pattern at a later stage without having to make any changes to existing design or code. An example of this is, instead of defining runtime variability with `if`-statements in the code, the unbound variation points would make it possible to change this to a compiler variability with `#ifdef` instead, and producing a unique binary for this variation. Variability is achieved by changing the settings within the anytime variability tool. This would make it possible for any combination of core assets arranged at any time to yield the same final product.

2.3 Variability in the automotive industry

AUTOSAR is an open and standardized architecture for the automotive industry. It was developed jointly by some of the largest companies within the industry together with third party suppliers and tool developers. According to its official web-site the standard aims to improve the way electronic equipment is developed so that ultimately one can increase safety, performance and environmental friendliness. [1]

Variability is only recently introduced in the AUTOSAR standard. As of version 4.0, released early 2010, AUTOSAR explicitly states that managing variability within the standard is more about documentation rather than implementation (AUTOSAR 2010*b*). Nevertheless, UML meta-models are used to describe four variation patterns, which is implemented in three abstraction levels. The most abstract level does not provide any details of variability, thus it primarily models functionality. The second, called Annotated meta-model, provides information about where variability should take place. In other words, this level indicates where the variation points are used in the application. Finally, the lowest UML abstraction level, called Extended meta-model allows the developer to model latest binding time and constraints that apply to these.

There are four patterns described in the standard: aggregation value, association value, attribute value, and property set value pattern. The patterns are primarily discussed at system design level but the standard does support variability all down to post-build. Note that AUTOSAR uses a different notion of post-build compared to Bachmann & Bass (2001). Post-build in the standard is primarily concerned with linking. AUTOSAR distinguishes between post-build and runtime, and explicitly states that runtime variability is excluded.

To our knowledge, there is only one paper discussing run-time variability from a strict automotive perspective. Weiss et al. (2009) describes a method for self-organizing software. For example, if a software controller fails another controller can take over its functionality. This is done via a third component which monitors the health of other controllers and in case of a problem, re-instantiate the software from the faulty controller on the healthy controller. The authors takes a pragmatic approach to building theory and are not concerned about an AUTOSAR compliant implementation.

2.4 Theoretical framework

There are relatively few studies on how to evaluate a variation mechanism or pattern. Largely due to the subjective nature of studying patterns, as it is in the situation which they are used that determines whether they are suitable or not. However, Fritsch et al. (2002) proposes a complete methodology for assessing and categorizing variant patterns used in an organization. It is complete in the sense that it not only looks at one particular pattern at the time, but evaluates them against each other. By acknowledging the subjectiveness of their implementations, through the construction of quality attributes for a pattern, they are able to represent the patterns nature in a particular situation. The quality attributes are organized in a quality tree (see figure 4). Thereafter each quality attribute is given a rating, for example, 1 to 5 where 5 represents that this attribute stands out in this specific area more than in any other pattern. The authors recognize the trade-offs that have to be made when patterns are weighed against each other by emphasizing that it is only guiding, rather than definite. Once the qualities have been determined, the patterns can be organized in a matrix to make it easier for developers and architects to find and use the right pattern at the right time.

For the purpose of this case study, we chose to adapt a subset of the method described by Fritsch et al. (2002) to organize and analyze our data, namely the quality tree. The leaf nodes represents strengths and weaknesses discovered during the implementation. Once they are grouped under quality attributes an indication of which qualities the implementation affects is discovered. The classification is based on which quality attribute the quality primarily concerns. Eventually, the quality tree provides guidance for the architect when making decisions for how to manage variability. In addition to guidance, the quality tree provides a good overview of the characteristics of one specific pattern. This is considered advantageous here since only one implementation was completed, and for future work it is available for comparison against other implementations.

3 Research Method

In this section we present the result of a three month long interpretive exploratory case study (Walsham 1995). After a motivation for this research design we outline the impact of the study by describing the environment we have worked in. The following subsection introduces the data sources used, and finally the process used to implement the prototype is described.

3.1 Case setting

In an interpretive case study the researchers are allowed to make use of preconceptions and use it to their advantage through, for example, deeper insight (Walsham 1995). On that account, we used Volvo Cars's preconceptions for managing variability to implement a variant mechanism for run-time variability. In other words we are basing our findings on what developers at Volvo Cars have already tested and deployed. This establishes a level of credibility and reliability in the data used. Although we could have opted for a positivistic approach, thus neglecting previously discovered advantages and disadvantages, we intentionally continued to build on a solution that is already accepted in industry.

It is difficult to predict the variables that will influence the work, hence we provide insight on what is required of the standard and the developers respectively. In addition it is difficult, since the existing body of knowledge is limited, to state beforehand which variables will influence each other. Contrary to stating a number of hypotheses in advance, we have chosen to provide a starting set of guidelines. These have been elicited from an architectural specification provided by Volvo Cars as well as through meetings with one of its software architects. The guidelines are presented in 3.3.

This study is exploratory in the sense that, despite previous research in variability management and variability mechanisms, no one has looked at the implications on massive product line productions. In addition, we provide insights on what is required for run-time variability to be successful. The huge amount of cars produced at Volvo

[1] AUTOSAR Official website, http://www.autosar.org, 2010-03-14

Table 2: Summarized data sources

Source	Type	Advantages	Limitations
Volvo Cars component specification	Document, architecture specification	Valuable information based on refined domain knowledge	Restricted by confidentiality agreement.
Volvo Cars run-time variability specifications	Document, architecture specification	Valuable information based on refined domain knowledge	Draft document, may change. Restricted by confidentiality agreement.
Software architect at Volvo Cars	Regular discussions	10 years of hands-on architect experience at Volvo Cars	Data is interpreted twice
AUTOSAR 4.0 Specification	Documents and UML meta-models	Publicly available. Thorough and with examples.	Difficult to address all relevant sections.

Cars each year, where all are adapted individually by the customer, presents a unique case with high demands on the variant handling. The ability to alter software and implement variation points decreases further into the product development (Bosch et al. 2002) and most variability studies evaluates design-time, compilation and linker variant mechanisms. There are, at least, three stages in development that variants are used today at Volvo Cars including compilation mechanisms, local parameterization, and distributed parameterization. The last two are examples of adaptation during startup. The compilation mechanism is a variant dealt with before run-time and already supported in AUTOSAR. Consequently, this study focus on the last two mechanisms.

As described in section 2.4 variability is recently introduced in the AUTOSAR standard and it has in most cases not yet been put in to practice. Our direction will be to discover how one will be able to support the variability patterns needed by the automotive industry in the AUTOSAR standard. Generalizing a run-time variability solution is beyond the scope of this paper as that would require studies on how a larger set of manufacturers deals with run-time variation. We provide a better understanding of what is required of one particular industry standard to encompass run-time variability due to the limited insight of the run-time variation problem in the automotive industry.

3.2 Data sources

The primary sources for this case study are provided by Volvo Cars's architectural specifications for handling variance. First and foremost, these specifications concern the adaptation during startup and run-time variability. Due to a confidentiality agreement any specifics from these sources are removed. These documents describe the way Volvo Cars currently are applying variant patterns and mechanisms in their software product lines, and so far AUTOSAR is not addressed in any of them.

Moreover, discussions with a software architect at Volvo Cars have provided valuable insights in the architecture used in Volvo cars, as well as terminology and concepts used in the automotive industry. Additional discussions with tool developers at Mecel AB have also contributed to detailed technical understanding of the possibilities and limitations of the AUTOSAR specification.

Discoveries made during the prototype implementation have been evaluated together with the software architect from Volvo Cars. As stated earlier, instead of trying to avoid preconceptions, we make use of the insights acquired by the architect's 10 years of experience in the automotive industry. This knowledge contributes to a better understanding of the individual constituents, i.e the mechanisms, as well as the whole, when the mechanism is tested within the standard (Klein & Myers 1999). The results from the discussions with the architect were collected as a set of suggested improvements to build upon. This reviewing process is described by Klein & Myers (1999) as an important constituent in hermeneutics.

Our secondary source is the AUTOSAR 4.0 specification published in January 2010, and more specifically, the template documents concerning variability management and variability patterns in AUTOSAR (2010*a*). There are also UML meta-models provided with the standard which describes the variation patterns, the constraints imposed and how to apply them in development.

Table 2 summarizes the data sources used in this case study according to Creswell (2008) methodology of categorizing data in qualitative studies.

The data outlined above is part of the specific case and thus directly applies to Volvo Cars. Each source is subject to our interpretations. Where applicable, for example through the discussions with the software architect, our interpretations are tested in a broader context where the architect's contribution and background provides added value through rich insight (Walsham 1995). Eventually, the results are applicable to industry and academia as we are able to draw on best practices and provide a conceptualization, the prototype, of variant handling within an industry standard.

3.3 Process

Interpretive case studies promotes an iterative process for data collection and analysis such that initial assumptions and theories can be re-evaluated when needed (Walsham 1995). Therefore, we divided the implementation into three phases. The first and second phase are written entirely in C, and the third is developed using AUTOSAR compliant tools. To assess the specification provided by Volvo Cars, the first phase intentionally avoided any use of AUTOSAR. Consequently, this made it possible to discover what was required of the variant before making the prototype more true to the automotive environment. In the second phase, the findings from the first phase were used to further refine the implementation towards the automotive industry. The implementation in the third phase is based on the findings, with some limitations, from the previous phases.

We started the development based on the following premises and requirements:

- as much as possible takes place in run-time
- a solution must be independent from the data it is supposed to pass
- data used by a service does not have to be stored on the ECU[2] where the service is running

[2]Electronic Control Unit, any embedded system in automotive electronics to control a system or sub-system.

After all phases were completed and summarized in a quality tree (see 2.4), we revisited AUTOSAR's and Volvo Cars's specifications to see what could be learned from existing variant patterns. It also opened up the opportunity to find the places where the AUTOSAR specification needs to be modified to support run-time variability.

4 Research Data

The results outlined in this section are coming from a prototype implementation of a run-time variability pattern. It is based on the architectural specification provided by Volvo Cars and concerns the distribution of configuration parameters in run-time. Gamma et al. (1995) named this pattern publisher-subscriber which is a mechanism for components, during application execution, to subscribe to state updates generated by another component, the so called publisher. A publisher broadcasts, upon state change, new information about the state to all its subscribers. A broker, or distributor, is occasionally introduced to support multiple publishers and delegate the responsibility of the actual transaction of data. Take note, however, that Volvo Cars's specification in its current state does not require subscription to take place in run-time. Our implementation, on the other hand, does. This decision was taken to fully understand the complexity of complete run-time variability where no decisions are made beforehand.

4.1 Implementation phases

The three phases are: 1) a pure implementation using the C programming language, 2) an improved implementation which more closely resembles the automotive environment, especially that of a Controller Area Network (CAN), which is an network implementation widely used within the automotive industry (ISO 11898-1 2003), and 3) an implementation which follows the AUTOSAR 4.0 specification. During all three phases the main source of influence for the design decisions were the AUTOSAR specification, and an iterative approach was used where each consecutive phase built upon an earlier. Descriptions of the implementations of all three phases are outlined below.

4.1.1 Phase 1: C Implementation

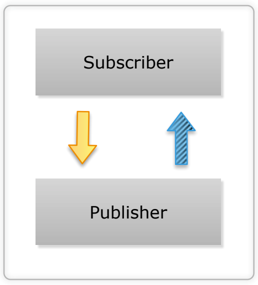

Figure 1: Diagram showing the c-implementation

In the initial phase we wanted to explore the general characteristics of a publisher-subscriber implementation. The implementation was made completely in C and to simulate the data network mandatory in cars, we used a native TCP-sockets with listeners and receivers in separate threads and processes. A component that wants to make a subscription does, in other words, so using a TCP request to a port specified by the publisher. This design allowed us to early on investigate the distributed constraints of the pattern.

As showed in figure 1 the system consist of two components, the arrows indicate communication between the components where the yellow arrow is an incoming request and the blue striped arrow shows the outbound parameter.

The two following phases implements a push technique in contrast to a pull technique which this phase uses. Hence, the publishers are not aware of the subscribers and decisions about which data to keep is delegated to a message broker. This is in-line with the AUTOSAR specification.

4.1.2 Phase 2: CAN-bus implementation

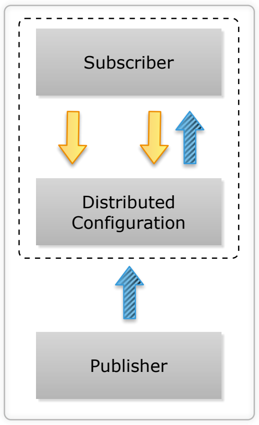

Figure 2: Diagram showing the implementation with CAN-bus

Phase two aimed to develop a prototype true to the cars actual communication system to simulate the CAN-network more accurately. The implementation in phase one used communication between subscribers and publishers with external means of communication, hence the sockets. When adapting to the CAN bus, broadcasting cannot be done to individual addresses. This lead to a function calls for subscriber communication, instead of using sockets to register parameters. As showed in Figure 2, between the publisher and subscribers a message broker was constructed. The message broker listened for incoming parameters and saved only those parameters that were registered by subscribers present in the same process, illustrated by the dashed line. Each subscriber ran in a separate thread communicating with the broker in the same process. The publisher ran in a separate process to simulate a different ECU and communicates with the broker over a socket sending all variables one after each other.

4.1.3 Phase 3: AUTOSAR implementation

In Figure 3 the proposed architecture for how to implement a publisher subscriber pattern within the AUTOSAR standard is illustrated. It contains three types of elements (see figure 3). First, the Distributed Configuration running as a basic software component which is available on every ECU. Second, the publisher running as an application software component. This component have the responsibility of reading and publishing parameters to the Distributed Configuration. Also note that there could be more than one publisher per ECU. The last, and third, type of elements are the actual software components which acts as subscribers. All application software components have the ability to act as a subscriber by simply registering their interest of a parameter to the Distributed Configuration. When accessing the parameter they use the AUTOSAR RTE which connects the applications with the Distributed Configuration and the parameters are available throughout the run cycle.

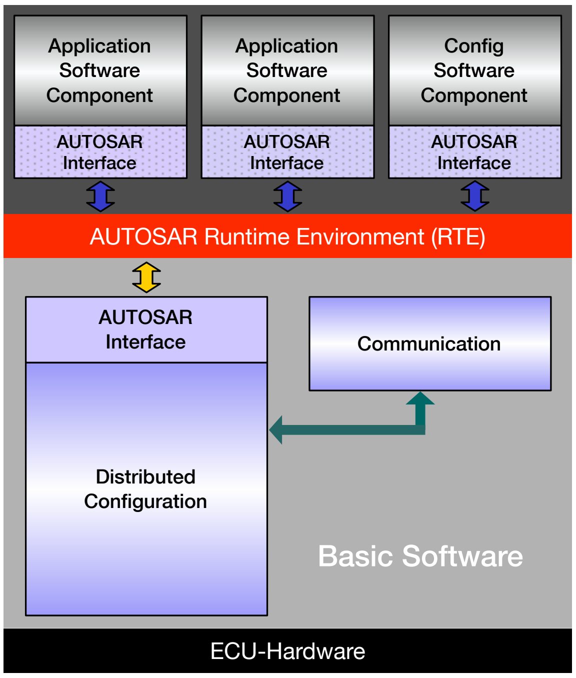
Application Software Component
Application Software Component
Config Software Component
AUTOSAR Interface
AUTOSAR Interface
AUTOSAR Interface
AUTOSAR Runtime Environment (RTE)
AUTOSAR Interface
Communication
Distributed Configuration
Basic Software
ECU-Hardware

Figure 3: Proposition for AUTOSAR implementation of Configuration System

There are two possible scenarios for the Distributed Configuration to handle variables on the ECU. Either they are treated as a local configuration parameter, and only used within the applications on the same ECU, or global and distributed to many ECUs. Local configuration could be useful for specifying static variables, like different language files for a car's dashboard. Secondly, the parameters can be broadcasted both locally and globally. The Distributed Configuration is responsible for broadcasting the parameters on the CAN-bus. The parameters are then received by another Distributed Configuration component on the receiving ECU which stores the parameters locally, and may be used as described above through functional RTE calls.

During this phase the code used in previous phases where integrated into an AUTOSAR environment[3]. The AUTOSAR environment consist of three different components, namely a AUTOSAR configuration tool, a workbench with supplied basic software components in C, and a Testbench which simulates the CAN-bus used within a car. The Testbench displays all messages passed for debugging and it is possible to inject messages into the CAN-bus. We used the supplied configuration tool to create a basic software component, namely the distributed configuration. The component was configured to listen to the CAN-bus with the ability the send and receive 8 bytes of data in each frame. This gave us an AUTOSAR compliant XML-file which was used to generate the structure for the basic software component in C, together with the correct AUTOSAR interfaces.

By using the built in Testbench and injecting messages in the CAN-bus the implementation was tested with simulated publishers. The implementation consisted of one Application Software component and one Basic Software component. The application component was constructed with a simple control loop initializing one parameter with the distributed configuration. Once registered the application continues reading the value and operates as normal. The distributed configuration is listening for incoming parameters, and when a new set of parameters is received these are made available for the application component.

[3]Picea Evaluation Package 2.0, generously provided by Mecel AB

4.2 Learnings during implementation

There are several aspects to highlight which were derived during the development of the three phases. These are outlined here.

4.2.1 Subscription

A problem which emerged early in the first implementation is that of subscription. All software components that require external variables must register to those during subscription. A software component which depends on a set of variables before it can enter an operational state must be guaranteed that these are delivered. If a subscription is not instantiated appropriately, such that the variables are not being delivered, the software component may be left in an erroneous state. Three questions arise:

1. What if the distribution component taking the subscriptions is not yet initialized?
2. When is the first set of parameters distributed to all subscribers?
3. If subscription takes place in run-time, for how long is the publisher going to wait before distributing the initial set of parameters to the subscribers?

Possible solutions include immediate distribution on subscription, as well as the alternative to design the software component with an initial waiting time before entering an operational state. Since the publisher will have instant access to the parameters it could distribute these immediately upon subscription. This would imply the pattern is initially used as a client-server pattern.

Additionally, subscription in the first implementation is done by attaching a list of parameter identifiers that the software component is interested in. It is, therefore, necessary to have a discussion on what parameters are likely to change during run-time. For example, a car will not suddenly have four doors instead of five. Basically, a limit between how much should be known by each part of the pattern must be drawn, and such discussion is beyond the scope of this paper.

In addition to classes of parameters, to attain full subscription support in run-time it should be possible to change a subscription during the whole driving-cycle, i.e from unlocking the door to locking it again. Hence, a software component can, based on a certain set of parameters, choose to register for more parameters or change the current set of parameters.

4.2.2 Multiple publishers

There is also the question of whether multiple publisher should be allowed or not. It is unreasonable to assume that all variables required in a car can be provided from a central unit. Instead sensors are spread physically in the car, promoting the support for multiple publishers. That said, in the current implementation only one publisher is available. One could use a central message broker and allow multiple publishers to pass messages through this central point, however, that would mean the establishment of one single point of failure. Due to the nature of the CAN network, used to distribute data in a car, subscribing and/or publishing ECUs must be aware of which frames to use. If a frame is wanted or not is determined at design time. Usually, with a configuration tool during ECU configuration. The AUTOSAR specification states that a software component must not be aware of the CAN frames the ECU is registered to, this should be abstracted away between the basic software layer and the application layer.

To minimize complexity this study use one publisher distributing parameters using one CAN frame id.

4.2.3 Push vs Pull strategies

There are two scenarios possible as defined by Gamma et al. (1995); either the publisher pushes updates to the subscribers and these are responsible for keeping track of what changes has been made to the parameters that it is interested in. Alternatively, you let the publisher(s) know something about its subscribers. That means, when updates are to be sent out, the publisher knows which subscribers to update. In phase 1, but not phase 2 and 3, the publisher was aware of the subscribers. The publisher is able to tell how many subscribes, and what they subscribe to and one can therefore make informed decisions about which parameters to update.

In the CAN-bus network the messages are broadcasted to every ECU, and it is not possible to specify a receiver for the frame. This suggests that a push methodology should be opted for, and as a result subscribers keeps track of which parameters to update on publication.

4.2.4 Register parameters of interest

These learnings are primarily drawn from phase 2 and 3 where all parameters that are going to be used by a Software Component needs to be registered with the Distributed configuration during the initialization sequence. This is so that the Distributed Configuration can keep track of what parameters it listen to. The registration can take place anytime during the driving cycle but if the registration is not done during initialization, parameters published earlier could be missed. This is not a problem if publication is cyclical, although timing is likely still an issue.

All components with run-time variability needs an initialization state where it register parameters of interest to the publisher. Until publication is complete, the component works with a set of default values. These may have to be specified in design time. The use of default values will ensure the stability of the system even if the broadcast would take longer then expected. As soon as the real value is assigned by the publisher the old values will be replaced in the Distributed Configuration's cache.

There is a complete separation between the publisher and subscriber through the broker. Hence, the subscriber has no knowledge of where the parameters are coming from and the publisher has no knowledge of which components are using the parameters it distributes.

4.2.5 Local and Global configuration files

The prototype was designed to be location independent, meaning that the publisher could be implemented on the same ECU or any other ECU with the distributed configuration implemented. When the publisher is located on the same ECU, the parameters could also be limited to distribute locally within the ECU or published globally. Since the software application component is independent on the publisher, it will operate in exactly the same way no matter where the publisher is located.

Table 3: Quality scenarios learnt during implementation

Identifier	Description
L1	Low coupling - publisher and subscribers separate
L2	Parameter identification
L3	Initial effort
L4	Scaleability
L5	When subscription takes place
L6	Multiple publishers
L7	Changing subscription
L8	Push data vs. Pull data
L9	Initial waiting time
L10	Full distribution vs. Individual distribution
L11	Transparent storage
L12	Only store relevant parameters per ECU

4.2.6 Parameters in an AUTOSAR environment

When using parameters within AUTOSAR, the size restrictions to a maximum of eight bytes in the CAN protocol will have to be taken into consideration. Looking at how Volvo Cars are using the parameters show that the limit of eight bytes is not enough for some parameter values. Instead these have to be split over multiple frames and a method for maintaining data integrity has to be added. This ensures correctness and quality for message delivery.

Also the parameters have to be identified in a unified matter. Looking at the configuration system at Volvo they identify the parameters with their position in the frame. Using this way is easy when dealing with a fixed number of parameters and the length of each parameters is bound. However, in an more general implementation in AUTOSAR this could raise a problem. A possible solution tested in the implementation is to use separators between parameters and instead look for the parameter number as identifier. The other implementation discussed is using a dictionary implementation, using a key-value storage. This could prove useful since the order and number of parameters sent would not affect the output. This would also give the possibility to resend a single parameter without sending all parameters again. The implicit drawback would then be the extra number of bits needed for sending identifiers within each frame, and the limitation of 8 bytes would make this even more complex.

4.3 Summary

After each phase in the implementation, time was taken to reflect over the lessons learnt during the implementation and these where discussed together with the Software Architect from Volvo Cars. To be able to have a discussion on the previous data we provide a summary in the form of a few words. These are summarized in table 3 and highlight the important aspects from each lesson learnt during the implementation.

5 Analysis

In this section we will discuss and analyze the data gathered during the implementation against related work to provide a solid base for understanding why run-time variability is needed and what is needed when implementing this in the AUTOSAR standard. We start with discussing the quality attributes defined in the quality

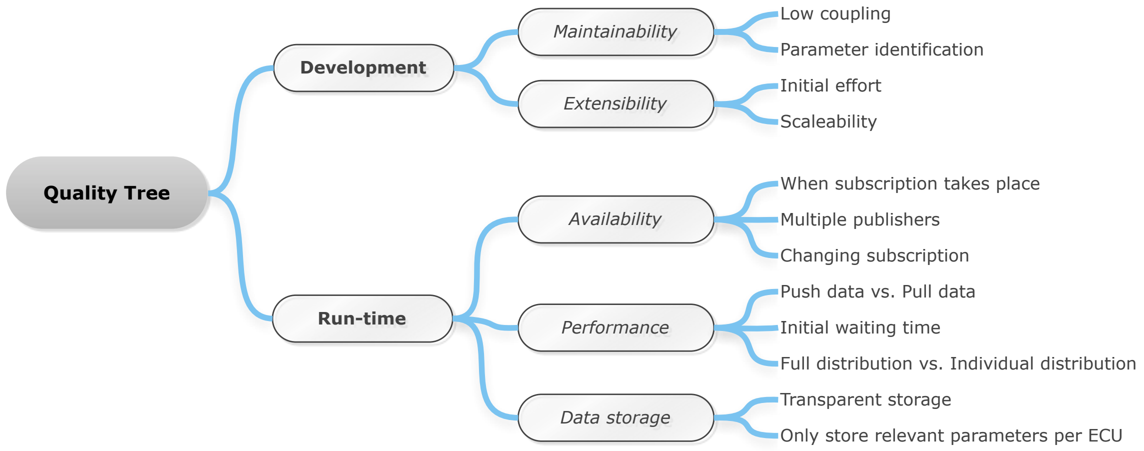

Figure 4: Quality tree for publisher-subscriber pattern

tree, moreover looking at how this affects the automotive industry's variability catalogue. Finally, we go through the limitations for this study and give suggestions where future research could be conducted.

5.1 Qualities of publisher-subscriber pattern as run-time variation mechanism

In figure 4 our findings from the three phases have been categorized into what Fritsch et al. (2002) calls a quality tree. From the left, the two categories *Development* and *Run-time* are constructs from the original model. Development concerns quality attributes which emerges during implementation, like maintenance or extensibility. Run-time, on the other hand, contains quality attributes which emerges when the product line is being used, such as availability or memory consumption. The level below development and run-time, for example performance, are quality attributes based on our interpretations of the characteristics outlined in section 4.2. This tree provides us with an overview of the run-time variability implementation. The leaf nodes in the quality tree describe strengths and possible weaknesses in the implementation. As an exploratory case study, these leaf nodes represents areas for further exploration. Basically, they are problems or challenges identified throughout the implementation. Below we discuss how these are significant for a production ready implementation.

5.1.1 Maintainability

The responsibility between the different components have to be clearly separated, no component should depend on another. In this case application software components are the subscribers, decoupling them completely from the publishers, using the distributed configuration component. By registering all parameters with the distributed configuration the software component does not need to know when or from whom the parameters are coming. In the same way the distributed configuration does not need to know who is registering the parameters. This is crucial from an AUTOSAR perspective, which uses a layered architecture, since calls must be made top-down. Hence, the distributed configuration does not need to know any details about the software component or how many subscribers are attached to it. The interesting parameters are stored in a cache belonging to the distributed configuration. Likewise, when new parameters are published these parameters will be updated in the cache.

Parameter identification is not addressed by this study. For a more detailed discussion, see section 5.4.

5.1.2 Extensibility

Developing software which satisfy variability is, depending on strategy used, a costly endeavor (Bachmann & Bass 2001). For example, constructing a code generator for particular configurations is both time and cost consuming.

The initial effort for developing the distributed configuration is high due to the complexity of the component. But, in comparison, adapting a software component to become a subscriber is easy as it only requires the use of functional calls through the AUTOSAR RTE. However, since it is possible to re-use this component in all ECUs of a car the initial effort may be neglected. On top of that, at least one publisher has to be provided to distribute data. Thereafter, it is possible for as many software components as required to make use of the information once they have registered with the distributed configuration. Additional components can be added, for example, with a software upgrade, at later times without affecting existing software components.

5.1.3 Availability

Availability contains, perhaps the most important, findings in this case study as they affect the entire system. In no particular order, we pose the following questions:

- How should subscriptions be managed if the distributed configuration is not ready or unable to accept subscriptions, for example, during start-up?
- If there are multiple publishers available in the system, how are identical or similar messages handled?
- Again, in a multiple publisher set-up, should it be possible to define priorities between them, and how is this handled by the distributed configuration components?

The first question is primarily about fault-tolerance and down to the real-time specification of the system. As, for example, more subscriptions will take place when the car is starting one may question if the system is able to initialize the distributed configuration before all software components starts sending subscription requests.

While this may partly be a question of performance, it is also down to prioritizing the resources. We have discussed the option of operating on default values, implying that the component needs to be carefully designed with regard to this. For example, how long can a software component operate on default values? This is closely related to the function that the component is to perform. An alternative is to avoid using run-time variation for safety critical components. As further elaborated in section 5.4, real-time constraints, which in the automotive industry is linked to safety, are not a part of this study.

The Testbench provided in the AUTOSAR environment did not allow us to test multiple publishers in a distributed setting since it does not support more than one ECU simulations. Although it is theoretically possible to use multiple publishers (Gamma et al. 1995), we do not have sufficient data to provide any conclusive insights on how this should be tackled in AUTOSAR.

5.1.4 Performance

Within the car industry, safety is paramount. There are real-time requirements with hard deadlines that has to be guaranteed. For example, an airbag system must respond when the car is in a crash. There is simply no room for alternatives. One issue that we discovered with the publisher-subscriber pattern is closely related to subscription and can be summarized in the following question: "when is the first set of parameters distributed?" As mentioned above, one may choose to not attach any software components to this type of run-time variation mechanism.

Furthermore, if a software component registers in the middle of, or just after, a set of parameters has been distributed a parameter value could be missed. And, on the other side, if the publisher is waiting for too long before distributing the first set of parameters other parts of the system might be negatively affected. One possible solution from a publisher perspective would be to resend all parameters within an interval to guarantee that all software components eventually receives their correct values. Until then, the components will run using its default value. Possible side effects of such implementation have to be considered.

Based on the characteristics of the CAN-network, we suggest adopting a push technique for the publisher. The ECUs need to be configured to listen to CAN frames designated to carry publisher-subscriber parameters. But beyond that configuration, the software components need and should not know about any frame specifics. Thus, distributing the data is managed entirely at lower levels in the architecture. This is further confirmed by the ability for publisher to directly publish data to the CAN-network, basically side-tracking the distributed configuration. Although, that would defeat the purpose of the distributed configuration.

5.1.5 Data storage

The software components does not need to know where and how the parameters are stored. In the quality tree we call this *transparent storage*, meaning that the publisher of the data could actually be next-doors to the subscriber, or on an entirely different ECU. In the latter case, the parameter is passed to all distributed configuration components over the CAN-network.

Moreover, the publisher is completely responsible for reading the parameters. For example, these may be read from sensors or a configuration file. How the parameters are stored is decided by the individual implementation of the publisher and could easily be adapted by each manufacturer. Consequently, as Gamma et al. (1995) explains, the implementation is provided with further extensibility via the broker, or the distributed configuration as we call it. Take note though, that we also give it a larger responsibility compared to Gamma et al. This is closely related to maintainability for de-coupling reasons, but with this quality attribute as it concerns where data originates from.

5.1.6 Core arguments

We argue for a publisher-subscriber to be successful in an AUTOSAR environment it will deviate from the traditional pattern as defined by Gamma et al. (1995). This is primarily due to the added responsibility to the broker and the methodology chosen to distribute parameters. The latter may affect the real-time properties of the system. Moreover, we recognize the usefullness of providing a transparency of where data is stored - an issue coupled with maintainability. With respect to the AUTOSAR architecture, good maintainability and extensibility are relatively easy to retain since the architecture provides clear guidelines for separation of concerns. Finally, we also warrant for careful management of subscriptions. Which, if not done properly, will affect the performance and the availability of the publish-subscriber implementation. Based on our prototype, we consider these aspects vital for a successful, production ready implementation.

5.2 Automotive industry

The motivation for introducing variability in AUTOSAR is threesome. First, it is to establish a common language to enable suppliers and manufacturers to work closely together. Second, it is to avoid redundancy between artifacts. And thirdly, it provides a basis for basic software product lines in which, for example, a supplier can support more options than that delivered to a manufacturer (AUTOSAR 2010*b*). Previously, variability was not standardized and instead managed internally by each manufacturer. This also caused issues when communicating with suppliers as each party would have a different understanding of when and where, for example, variation points were introduced. Therefore, in AUTOSAR, variant handling is mostly about documentation at architectural levels, the meta-model is considered a documentation artifact too, and the actual implementation is left to tool vendors and suppliers.

We have demonstrated how run-time variability can be encompassed in an industry standard, acknowledging that pre-build variation patterns are not enough to encompass the growing customization demands. Implementing the publisher-subscriber pattern is one way of supporting run-time variability in AUTOSAR. Furthermore, the implementation fully complies to the current specification in the sense that is written as standalone software components. No structural changes are required to the specification, thereby we enable manufacturers to add a publisher-subscriber pattern for run-time variability as a complement to the specification. In other words, remaining compliant with the 4.0 specification and retaining backwards compatibility.

There is no support at all for run-time variability in the AUTOSAR 4.0 specification. This may require an extension of the meta-model and at least one additional pattern specification to be added, i.e a description of how the pattern is implemented and used. Moreover, another binding time, that of run-time, must be added. Since a variation mechanism bound at run-time requires all variations to be implemented and provided with the object code (Bachmann & Bass 2001), it is also suggested that necessary constraints and checks are added to avoid missing implementations. For example, one wants to avoid that the software enters states requiring parameters that is not provided by a publisher. The latter applies primarily to tool support, but if possible this should be considered also when extending the meta-model. Note, that in this study we will not go into further detail about how, or consider if, the standard should be extended. We merely note that these areas have to be investigated to support run-time variability.

Broy (2006) notes that better re-use of software is preferred within the automotive industry. We cannot entirely confirm that a publisher-subscriber pattern will increase re-use. The pattern, however, provides a mechanism to share and distribute configuration parameters and other variables between ECUs and software components during run-time. As a methodology, the pattern can operate on any set of configuration parameters depending on the implementation of the publisher. Therefore, in-line with Weiss et al. (2009), run-time variability support increases the ability to support after-market services. The choice of variation mechanism partly depends on the ability to interact with the product once it has been deployed (Bachmann & Clements 2005). Our case supports reconfiguration once the product, a car in this case, has left the factory. Making it possible for service garages, car dealers and 3rd party suppliers to update a part of, or the entire, configuration at a later time. Furthermore, entire sub-systems can be added to make use of existing configuration variables that are passed via the distribution configuration as long as the new sub-system subscribes to them. This significantly improves extensibility on the after-market, as there is no need for changes to the already deployed systems. Therefore working in the same way as a plug-in architecture. Moreover, as long as the core is maintained, there will always be backward compatibility for the deployed systems. Meaning, 3rd party and after-market components can be added and removed at any time without affecting the base. Due to real-time constraints, however, at some point the CAN network may become saturated. Managing this type of fault-tolerance is part of the limitations of this study.

5.3 Extending the variability catalogue

In contrast to the method for self-organizing software proposed by Weiss et al. (2009) who moves towards moving functionality between controllers, we take a different approach. The publisher-subscriber implementation described above originates from a specification for distribution of configuration parameters. In essence, two characteristics particularly distinguishes our research from Weiss et al (2009)'s, namely data structure complexity and compliance to a standard. First off, our implementation only supports data up to 8 bytes, which amounts to the carrying capacity of CAN frames. Since the distributor determines which parameters are relevant to the subscribing software components it limits, or rather influences, the type of data sent. We recognize two approaches to managing data integrity, either let the distributor control this or leave this responsibility to the subscribing software component. The latter will allow for greater flexibility of data structures passed. The second characteristic, compliance to a standard, is extensively covered in our case. Weiss et al (2009)'s work recognizes AUTOSAR but does not attempt to follow it for their implementation, whereas we do. While the work of both groups are far from production ready implementations they argue for run-time variability and contribute, in their own aspect, with the difficulties that have to be addressed, both by AUTOSAR and by the individual manufacturer, before applying it in practice.

While there are numerous case studies and observations for variation mechanisms during pre- and post-build, there are, to our knowledge, few on run-time variation. We successfully provide a case where the publisher-subscriber pattern is used as a variation mechanism. In comparison to product architecture derivation, compiler switches, and binary linking, and `if`-statements for run-time variability (Bachmann & Bass 2001, Svahnberg et al. 2001), the publisher-subscriber pattern extends the variability spectrum. The pattern, as implemented, will not bind the variant until the parameters is used by the software component. Moreover, it is easily extended with new software components if the distributer is supplied as an integral part of the core system. The initial development effort before implementing new software components that follow the pattern is confined to defining the initialization. In addition, the publisher-subscriber pattern illustrates a more complex case for managing variability that is not limited to one software component. From an architectural perspective this is important if multiple variations must be accounted for. As Bachmann & Bass (2001) explains, variation points described in an architecture must be linked to the functional or non-functional requirement instantiating them. Since the publisher-subscriber pattern will affect a system architecturally (see figure 3), it is unclear how the variation point, which could be multiple, are linked to requirements. Further studies are required to examine this.

In addition to above mentioned studies, this study provides a case for how variant handling is used in large consumer-oriented systems. There are several studies on variation management in large systems but with relatively few deployments (Svahnberg et al. 2001, Jaring & Bosch 2002, Keepence & Mannion 1999, Brownsword & Clements 1996), on the other side, there are few studies on how organizations delivering products to end-users manage variation (Svahnberg et al. 2001). Examples of the type of system that has been investigated is military naval system with few deployments. Another example is the mobile phone industry, which are produced in thousands, and are not varied on a per-customer basis. The case we are looking at is where each of the 400 000 cars produced by Volvo is adapted to suit the customer's wants and needs. This calls for a wider selection of variation mechanisms.

When working within software product lines the ability to reuse software is important and introducing variability gives a the needed flexibility for the system to adapt (Jaring & Bosch 2002). However, implementing run-time variability does not only give extra flexibility it also increases the constraints on the system. For every runtime variation, the system needs to include code for handling both cases, even if only one variation is performed. This leads to an increased size of the software and extra complexity is introduced.

In anytime variability as defined by Van Der Hoek (2004) the variation point does not need to be decided in design time. This concept could also be adapted within an AUTOSAR environment, for example the tools used to create AUTOSAR compliant code. However, this do not effect the AUTOSAR standard merely the way the standard is used. Since AUTOSAR 4.0 variability has been available but it is still limited. This prevents the possibility of anytime variability, however, one could postulate that if runtime variability is introduced within the AUTOSAR standard the anytime variability mechanisms could be introduced in the configuration tools to support an even broader concept of variability. This gives the designer or developer even more flexibility, and instead of bounding the variation points early in the design, it can easily be changed depending on current settings by exporting a new version with different variation patterns, hence supporting anytime variability.

5.4 Study limitations

As in all research there is no time to follow-up on every aspect. In this section we discuss the limitations imposed on this study. They are all, in their own aspect, relevant to run-time variability in AUTOSAR, but have not been evaluated or tested thoroughly. A limitation in scope provides a targeted contribution, consequently these issues are suggested for future research.

The proposed structure requires all software components that need a distributed parameter to register their interest at startup. However, there is no guarantee that this will occur within a specified time frame. In section 5.1.4 we outlined the issue of performance with the publisher-subscriber pattern. As the goal was to establish whether or not it was possible to add run-time variability to an industry standard, and not to evaluate the performance issues of such implementation, performance requirements were omitted.

One also need to evaluate the data structure for the parameters. In our research we identified two possible solutions, however, no proper testing has been conducted to be able to elicit possible advantages and drawbacks with either solution. Using a position-based structure, for example, parameter number 15 describes type of doors mounted on the car, allows us to send more parameters in a shorter period of time. However, resending a single parameter is not possible in such setup since the whole list of parameters need to be sent to maintain the position structure. Using a dictionary approach gives the flexibility to only send the parameters needed at that moment. This may imply other difficulties though, for example, meeting real time constraints and separation of concerns.

This article looked at a small portion of the automotive industry, namely passenger cars, and more precisely from Volvo Cars perspective. It is likely that the needs for other manufacturers and other areas of the automotive industry are different. For example, requirements for a truck will prioritize different quality attributes, affecting the choices of variation patterns and their implementations in other ways. Before suggesting run-time variability to be included in the AUTOSAR standard, further studies on other manufacturers should be conducted.

6 Conclusion

This paper set out to improve software product line manufacturing of unique products in large-scale settings. In this exploratory case study we provided contributions on three levels. First, and on a practical level, implementing run-time variability is possible while remaining compliant to AUTOSAR 4.0 using a publisher-subscriber pattern. And, we identified where changes to the standard is required to support the variation mechanism. As a consequence, we argued that run-time variability improves a manufacturer's capacity to support after-market services, 3rd party integration and increase software re-use. Second, from a developer perspective, we showed that the traditional publisher-subscriber pattern have to be adjusted for the real-time constraints existing in the automotive industry. In addition, transparent storage enables low-coupling and easier scaleability, but real-time constraints and subscription management needs further consideration. Finally, and on a general level, we demonstrated that the publisher-subscriber pattern can be used as a run-time variation mechanism. In conjunction with Weiss et al. (2009)'s paper on self-organization of software, we strengthen the support for run-time variability and argue that it is a crucial component when developing large numbers of unique products derived from one software product line.

Acknowledgements

The authors would like to thank: Ulrik Eklund at Volvo Cars for his valuable contributions to this research, Mecel AB for providing the development environment and technical insights on AUTOSAR used to develop the prototype, and Carl Magnus Olsson for his continuous thoughtful guiding of this thesis work.

References

AUTOSAR (2010*a*), 'AUTomotive Open System ARchitecture 4.0', [Last accessed: 2010-03-16].
URL: *http://www.autosar.org*

AUTOSAR (2010*b*), Generic structure template, 3.0.0 4.0, AUTOSAR.

Bachmann, F. & Bass, L. (2001), 'Managing variability in software architectures', *ACM SIGSOFT Software Engineering Notes* **26**(3), 126–132.

Bachmann, F. & Clements, P. (2005), 'Variability in software product lines', *Software Engineering Institute, Pittsburgh, USA* .

Bosch, J. (2000), *Design and use of software architectures: adopting and evolving a product-line approach*, Addison-Wesley Professional.

Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J. & Pohl, K. (2002), 'Variability issues in software product lines', *Lecture Notes in Computer Science* pp. 13–21.

Brownsword, L. & Clements, P. (1996), 'A case study in successful product line development', *Software Engineering Institute (SEI) Technical Report, CMU/SEI-96-TR-016. Carnegie Mellon University, Pittsburgh, PA* .

Broy, M. (2006), Challenges in automotive software engineering, *in* 'Proceedings of the 28th international conference on Software engineering', ACM, p. 42.

Creswell, J. (2008), *Research design: Qualitative, quantitative, and mixed methods approaches*, 3 edn, Sage Pubns.

Fritsch, C., Lehn, A. & Strohm, T. (2002), Evaluating variability implementation mechanisms, *in* 'Proceedings of the Second International Workshop on Product Line Engineering-The Early Steps: Planning, Modeling, and Managing (PLEES'02)', Citeseer.

Fürst, S. (2009), Autosar - a worldwide standard is on the road, Technical report, BMW Group.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design patterns: elements of reusable object-oriented software*, Addison-wesley Reading, MA.

ISO 11898-1 (2003), *Controller area network (CAN) – Part 1: Data link layer and physical signalling*.

Jaring, M. & Bosch, J. (2002), 'Representing variability in software product lines: A case study', *Software Product Lines* pp. 219–245.

Keepence, B. & Mannion, M. (1999), 'Using patterns to model variability in product families', *IEEE software* **16**(4), 102–108.

Klein, H. & Myers, M. (1999), 'A set of principles for conducting and evaluating interpretive field studies in information systems', *MIS quarterly* **23**(1), 67–93.

Spencer, H. (1992), ifdef considered harmful, or portability experience with c news, *in* 'In Proc. Summer'92 USENIX Conference', pp. 185–197.

Svahnberg, M., Van Gurp, J. & Bosch, J. (2001), On the notion of variability in software product lines, *in* 'Proceedings of the Working IEEE/IFIP Conference on Software Architecture', IEEE Computer Society Washington, DC, USA, p. 45.

Van Der Hoek, A. (2004), 'Design-time product line architectures for any-time variability', *Science of computer programming* **53**(3), 285–304.

Walsham, G. (1995), 'Interpretive case studies in is research: nature and method', *European journal of information systems* **4**(2), 74–81.

Weiss, G., Zeller, M., Eilers, D. & Knorr, R. (2009), 'Towards Self-organization in Automotive Embedded Systems', *Autonomic and Trusted Computing* pp. 32–46.