Efficient configuration management of automotive software

Cornelia Heinisch, Volker Feil, Martin Simons

Efficient configuration management of automotive software

Cornelia Heinisch

STZ Softwaretechnik

Volker Feil, Martin Simons

DaimlerChrysler

Abstract

The need for managing configurations of automotive software is growing significantly as a consequence of the continuously increasing volume of software in vehicles and due to the possibility to exchange software in vehicles over their lifetime.

In this paper, we discuss concepts for configuration selection, configuration verification, and configuration update and show an evolution for the step-wise introduction of these concepts to gain an efficient configuration management of automotive software. We propose a configuration model that defines automotive hardware and software releases and describes the dependencies between these releases. This configuration model forms the basis for an update management which enables the download and installation of new software releases into cars to maintain the car's software over its lifetime. We illustrate and discuss a prototypical realization, which takes the proposed concepts into account.

1 Introduction

The history of configuration management (CM) goes back to the early Fifties and has its roots in the aerospace industry. In the Apollo Space Program, thousands of changes were tracked using a defined configuration management process. At that time, the tracked changes belonged almost exclusively to hardware elements. For a long time, configuration management has been used also successfully for software. Elementary to Software Configuration Management (SCM) is the term configuration, where a configuration consists of configuration items (for example source code files, documents, executable system parts, etc.) and a tested configuration is called a baseline. SCM goes back to the mid Seventies and began with the introduction of the first SCM tools SCCS (Source Code Control System) [1] and Make [2]. Since that time the discipline of SCM has continuously evolved: RCS (Revision Control System) [3] replaced SCCS in 1985 and one year later RCS was extended to CVS (Concurrent Versions System) [4]. A configuration was created by labeling or marking a particular version of each file belonging to the system. Today, tools like Rational ClearCase [5] support the developers in SCM. The basic functionality offered by these tools is still the same as in the predecessor tools RCS and CVS.

The term software configuration management is used in diverse areas ([7], [8]). According to [6], the most relevant aspects in today's SCM tools are: version management, configuration selection, concurrent development, build management, release management, workspace management, and change management. In this paper, we consider software configuration management with respect to configuration selection, configuration verification, and configuration update, because we believe that these aspects are the most relevant to get an efficient configuration management of automotive software. For being efficient a configuration management must efficiently handle the increasing complexity of automotive software (see section 3.1). The key to success is to use hierarchical dependencies between configuration items for configuration selection, to define dependencies between

configuration items to verify the correctness of a configuration and to use a configuration update concept which allows to update individual system parts.

Section 2 introduces configuration selection (section 2.1), configuration verification (section 2.2), and configuration update (section 2.3) precisely and section 3 discusses these three configuration management concepts as parts of an efficient configuration management for automotive software. The subsection 3.1 describes a way from a completely implicit configuration management to a completely explicit configuration management and subsection 3.2 discusses a prototypical realization of an efficient configuration management of automotive software. An outlook towards an efficient configuration selection is given in subsection 3.3. The paper concludes with a summary in section 4.

2 Configuration management concepts

In this section we introduce concepts for configuration selection, configuration verification and configuration update.

2.1 Configuration selection

Configuration selection (also known as configuration construction) determines which configuration items belong to a configuration. A configuration item (CI) is a physical entity (source code file, header file, documentation, executable, subsystem, etc.) which is subject to configuration control. A CI can be an aggregation of other CIs, organized in a hierarchy. Any member of this hierarchy can exist in several versions, each being a separate CI. This means, a CI is a version of a node in the hierarchy.

Typically, modern SCM tools allow a hierarchical composition of SCM components to build up an overall system baseline, where a SCM component is a set of related files and directories (the CIs) that are versioned and baselined as a single unit [5]. A CI can be atomic, like documents and modules, or composed, like libraries and systems. According to [9], versions of a SCM component are also CIs at a higher level. A composed CI which forms an executable system part is called a runtime version. Depending on the complexity and on the fact whether a system under construction will be distributed or not, the system is fragmented into one or more SCM components. These SCM components can be composed to larger SCM components (subsystems) in a hierarchical structure. If a version of the overall software system meets the requirements of the customer it is released and called a software release. While the configuration management of the developers deals with the management of source code files, typically complete executable software systems are delivered to customers in the form of releases. These software systems contain executable components which have been generated from a configuration of source code files.

Figure 1 shows a hierarchical structure of CIs. The atomic CIs (A V1, ..., A V4, B V1, B V2, C V1, C V2, E V1, E V2, F V1, ..., F V3, G V1) are grouped in two SCM components D and H. A SCM component consists of a tuple of CIs.

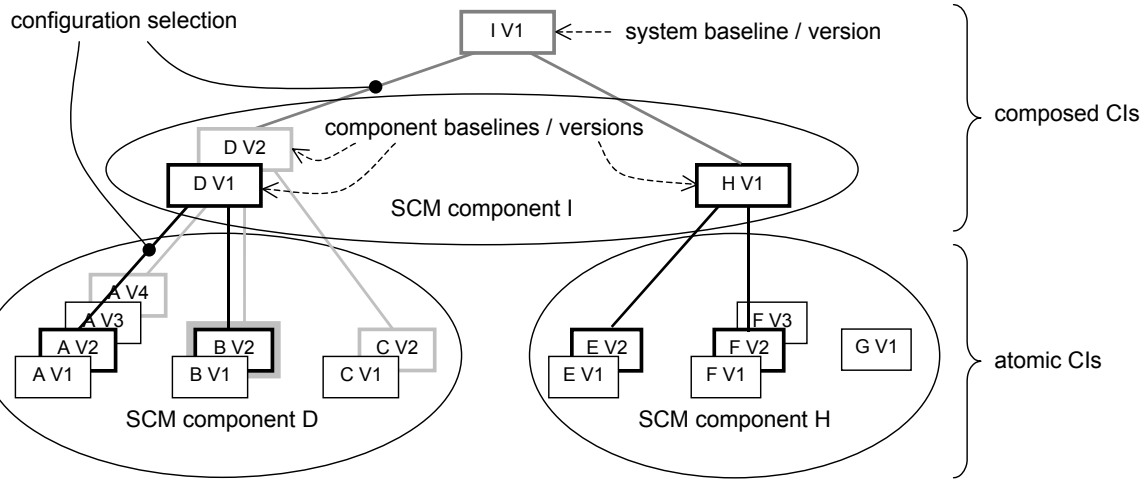

Figure 1: A system baseline made up of component baselines

For the SCM component D (A V1, ..., A V4, B V1, B V2, C V1, C V2), the component baselines D V1 (A V2, B V2) and D V2 (A V4, B V2, C V2) are defined, and for SCM component H (E V1, E V2, F V1, ..., F V3, G V1) the component baseline H V1 (E V2, F V2) is defined. Different versions of a SCM component (component baseline) are represented by different tuples. Two tuples are different, if they contain different combinations of CIs. A CI is a version of a node – in case of software a node is represented by a file. For example, the CIs (A V1, A V2, A V3, A V4) are different versions of a single file. Typically, a baseline contains just one distinct version of a file.

Configuration selection (baselining) is performed by considering dependencies in hierarchies between father nodes and child nodes.

One characteristic of defining system baselines – by selecting specific atomic CIs and specific component baselines – is that all CIs belonging to a specific configuration have to be selected manually – there are no rules, that verify whether the selected configuration really is a valid configuration that can be built, deployed, and installed, and which is running correctly. Therefore configuration selection (baselining) is still a tough job, despite good tool support.

2.2 Configuration verification

Configuration verification uses dependency descriptions (consistency criteria [10]) between composed CIs of a SCM component to verify the correctness of a configuration selection. Figure 2 shows dependencies used for configuration verification.

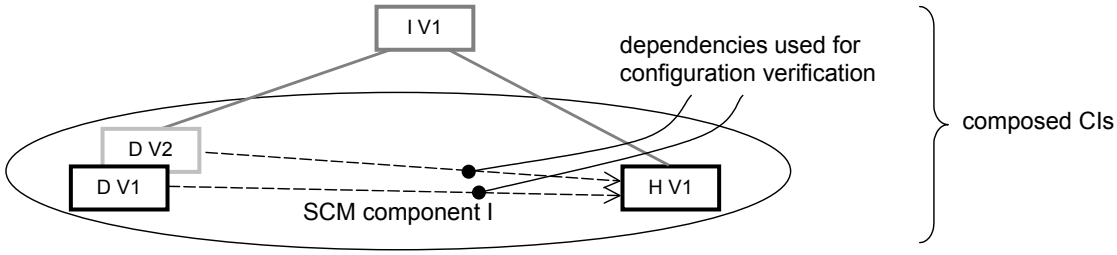

Figure 2: Dependencies used for configuration verification

Dependencies between composed CIs at the same level of a hierarchy, are used for configuration verification. These dependencies are used to verify whether a set of runtime versions forms a valid configuration. This verification can take place during configuration selection and after configuration selection, e.g., during system

installation or at system start-up. Having selected a specific CI the configuration verification can assist configuration selection by suggesting all CIs which can or should be added to the configuration.

Dependencies between atomic CIs (source code level) are not in the scope of this paper. It would be too complex to describe the dependencies between these atomic CIs (source code files, documentation, header files, etc.) to assist configuration selection of a component baseline. The only relevant dependencies between atomic CIs would be those which exist between configuration items located in different SCM components. But these dependencies can also be defined between the composed CIs themselves one level higher in the hierarchy than the atomic CIs.

We distinguish between three types of dependencies used for configuration verification:

- Direct dependencies between runtime versions
- Dependencies between runtime versions by using interfaces
- Dependencies between runtime versions by using contracts

We describe each kind of dependency description in the following subsections.

2.2.1 Direct Dependencies between runtime versions

Runtime versions are composed CIs which are executable system parts. For example, automotive flashware modules are runtime versions. A flashware module can be seen as a piece of code that can be written into the flash memory (a non-volatile memory device) of an ECU. A flashware module can also contain data like a characteristic curve which is required during the execution of an ECU application. Depending on the flash memory of an ECU and the structure of the ECU application, the software of an ECU can be divided into 1 up to n flashware modules. Today, automotive software is distributed over dozens of interconnected ECUs and is either located in ROM (Read-Only-Memory) or in flash memory. Software located in ROM is inseparably connected with the hardware. Consequently, the complete ECU is a composed CI and forms a runtime version.

For each runtime version (e.g. a version of a flashware module) it has to be defined explicitly which other runtime versions it requires to run (see Figure 3).

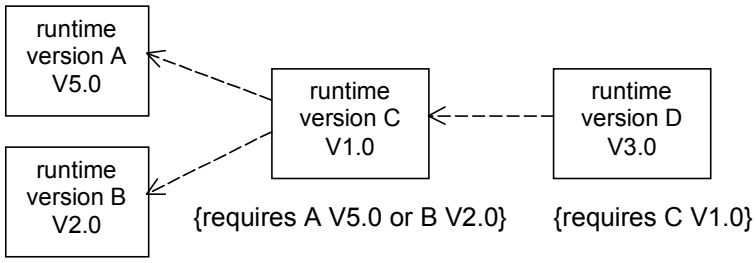

Figure 3: Dependency description between runtime versions

The challenge is to define for each runtime version – for an automotive example that means for each version of a flashware module – which dependencies with other runtime versions exist. This procedure is not only time-consuming but is also complex and very error-prone. An additional complexity is introduced by the need to express, if, for example, runtime version C V 1.0 requires A V 5.0 and B V 2.0 or runtime version C V 1.0 requires either A V 5.0 or B V 5.0. Figure 3 shows a possible additional notation to describe the restriction which states that runtime version A V 5.0 is a variant to runtime version B V 2.0 and therefore runtime version C V 1.0 does require just A V 5.0 or B V 2.0. Dependencies between runtime versions based on interfaces make the dependency description between CIs easier and allow a simple and unique notation of variants.

2.2.2 Dependencies between runtime versions by using interfaces

A runtime version can export an interface to indicate which services it provides and can import an interface to indicate which services it requires. An interface contains all call interfaces e.g. all object-oriented method declarations of that methods which are imported or exported, respectively. By using interfaces to describe dependencies between runtime versions it is not necessary that it is separately described by restrictions on which other runtime versions a runtime version is dependent – it is sufficient to declare which kind of runtime version, i.e. which interface it requires. A runtime version which defines what is required and what is provided by the usage of interfaces can be called a software component.

CORBA (Common Object Request Broker Architecture) [11] is a well-known example of a component-based system. The Object Management Group published the first standard for CORBA in 1991. In component-based systems, components publish their export interfaces (services) in a registry and use it to ask for an import service they require. Components typically run in the context of a framework, which may be distributed and which provides a runtime environment for software components. Another component framework which is more relevant for automotive applications is specified by OSGi (Open Service Gateway Initiative) [12]. Component frameworks as specified by CORBA or by OSGi use dependency descriptions based on interfaces to set up a complete running system. Nowadays, component frameworks can be found in cars of the luxury class. There, a powerful telematics ECU has the resources to integrate a component framework like OSGi. Software components that implement, for example, navigation based services can be loaded, installed, started, stopped and uninstalled dynamically.

A flashware module – which is the typical representation of a runtime version considering automotive software – contains only application code and data and does not contain information about the services it provides and the services it requires. But this information can be added for the purpose of configuration management in the form of an additional information to the CI representing a flashware module. The additional information shows the dependency on an (import) interface or the implementation of an (export) interface.

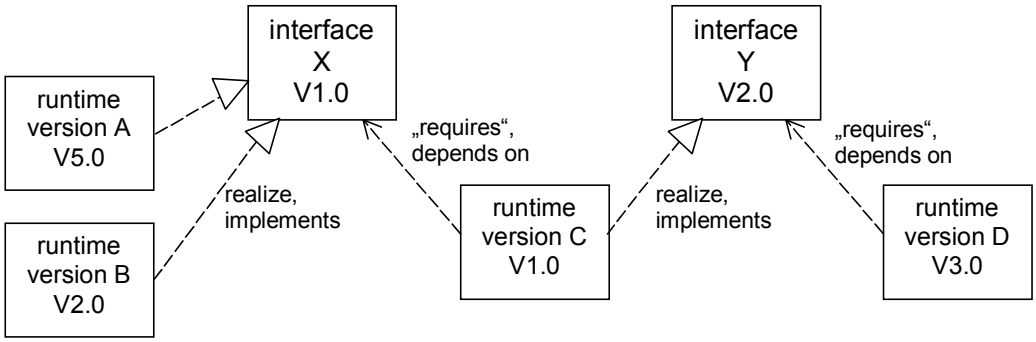

Figure 4: Dependency descriptions based on interfaces

Figure 4 shows a runtime version C which depends on (requires) a runtime version which implements interface X, version 1.0. Two runtime versions (A and B) implement interface X, version 1.0. Compared to an dependency description based on runtime versions this way of describing dependencies is much easier, because no knowledge about the specific runtime versions A and B is necessary. A further advantage: if a runtime version is added which also implements interface X, version 1.0, no additional dependency description has to be added to runtime version C.

The limitation of an interface is, that it does not allow to describe in which order the methods have to be called, nor does it allow to describe which timing constraints should be granted. This limitation can be put behind by using contracts to describe dependencies between runtime versions.

2.2.3 Dependencies between runtime versions using contracts

Contracts provide mechanisms to describe behavioral aspects with the aid of assertions, i.e. pre- and post-conditions and invariants (Design by Contract) ([13], [14]). Additional approaches exist to add synchronization and quality of service aspects to contracts [15]. A contract defines the rights and duties between the involved parties. A component providing a service guarantees that the post-conditions will be hold if the pre-conditions are fulfilled.

A pre-condition is a condition under which a call of a method is allowed. A post-condition describes the state of the system after the call of a method. The post-condition guarantees that after executing a method the state conforms to certain characteristics, provided that the pre-conditions have been hold before calling the method. Invariants describe certain properties which hold for all methods of a class and not only for individual methods.

If the service user guarantees that the pre-conditions will be fulfilled, the service provider guarantees, that the post-conditions will be fulfilled after the service is executed. Then a contract explicitly defines under which conditions a component works correctly. A service user should test in advance, if it is able to fulfill the pre-conditions of the service contract. If this is true, the contract is valid and a valid configuration of components is found.

2.3 Configuration update

Configuration update aims to update an installed configuration (a software release) with a new configuration. The first step of updating an out-of-date configuration is to calculate the difference to an up-to-date configuration [16] and to provide instructions for performing the update from the out-of-date configuration to the up-to-date configuration. Consider the following example: software release A with the CIs a, b, c, d, and f is installed in a car and should be updated to configuration B with the CIs b, c, e, and g. The update instructions can be as follows: Delete a, d and f and install e and g, where e may be a new version of d. This additional information can be added to the update instructions and is especially useful in the case where some user defined data of the CI d to be deleted should be transferred to the new version of d which is configuration item e.

The update instructions together with the new CIs (in our example e and g) can be packed into an update package. The update package then contains everything required to update a specific installed configuration to a new configuration.

The second task of configuration update is to perform the update instructions. This can either be done during non-operation (see section 2.3.1) or during operation (see section 2.3.2) of a system. Configuration update while a system is running is also known as dynamic change management and is additionally used for the migration of entities at runtime. In contrast to a runtime version which represents executable code such as flashware modules or software components an entity exists only at runtime. Typically an entity is instantiated from a runtime version. For example an entity (object, instance) is instantiated from a class file which represents a runtime version.

2.3.1 Updating configurations only during non-operation of the system

A configuration of a system can be updated safely, if the system is non-operating. The update package containing the new CIs and the instructions is unpacked and the instructions are executed. Before the update instructions are executed it has to be ensured, that the currently installed configuration is identical with the configuration the update package is generated for. Additionally it is important, that all update instructions are executed completely and correctly to gain a valid configuration and a correctly running system. An example of updating configurations during non-operation can be found in automotive industry. Flashware modules can be exchanged during the vehicle's lifetime e.g. in order to remove errors. These flashware modules can be only exchanged in a workshop during a vehicle diagnosis session. Then, the car is non-driving and therefore its systems are not normally operating but vehicle diagnosis can take place in this special system mode.

2.3.2 Dynamic change management

If there is a dependency between runtime versions as described in section 2.2.1, 2.2.2, and 2.2.3 the entities of these runtime versions can interact at runtime. Changing one interacting entity can lead to inconsistencies to all entities that are involved in the interaction. Examples for changes are the migration of an entity to a new environment (e.g. to balance load) as well as the update of an entity. For instance, if the entity is realized as a software component that consists of a set of objects, it can be dynamically updated by exchanging some old objects with new ones. However, what shall happen with the local variables of an obsolete method that are stored in the execution stack of the runtime environment? Which instruction shall be executed, if the instruction that is pointed from the instruction pointer of the runtime environment is no more existing? Familiar dynamic update management approaches ([17], [18]) avoid the dealing with these questions, because they prevent the changing of an interacting entity. Then, this update management is responsible for the monitoring of this prevention: Only operable entities are allowed to interact. Only non-interacting entities are allowed to become non-operable in order to get changed.

The concepts of dynamic change management (that includes dynamic update management) are relevant for managing long running, non-interruptible programs. For instance, such kind of programs run in satellites. However, for automotive software the update management in non-operational state is more interesting, because the vehicle's software running is interrupted at every drive break.

3 Concepts for configuration management of automotive software

In this section we apply the introduced basic concepts for configuration selection, configuration verification and configuration update to the management of automotive software and show a way to an efficient configuration management. The next subsection describes an evolution from an implicit to an explicit configuration management of automotive software.

3.1 From implicit to explicit configuration management

For managing the configurations of small systems, which consist of only a few software modules, it is sufficient to consider points in time where old runtime versions are replaced by new runtime versions. For example, if there is a new runtime version replacing an old one with the same functionality it is deployed to a specific server and made available for end-of-line production. Henceforth, the new runtime version is used in end-of-line production. Such a point-in-time-related configuration selection has the following consequence: Every time, once new runtime versions are deployed, vehicles with identical functionality will have a different configuration from vehicles manufactured before this point in time. If the number of configuration items to be managed is not very high and if no configuration update is performed over the lifecycle of a car, such an implicit configuration selection will be sufficient (see the bottom left quadrant in Figure 5).

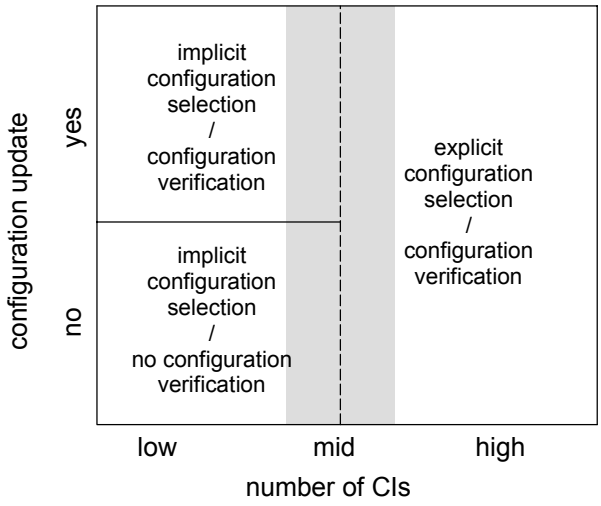

Figure 5: Relevance of configuration verification and configuration selection

If software is exchangeable over the lifecycle of a car - for example in workshops where some flashware modules but also whole ECUs have to be exchanged because of damage, corrective maintenance or because of new functionality is ordered - it has to be verified which flashware modules/ECUs have to be exchanged additionally to get a valid configuration again. Consider the following example: In the case an ECU is damaged the complete ECU will be exchanged. If the old version of the ECU is no longer available an ECU with a newer version will be mounted. In this case it has to be ensured that the software within this ECU is still compatible with the software in the other ECUs. To solve this problem dependencies between runtime versions can be introduced. These dependencies describe which runtime version requires which other runtime versions and can be used in the workshops to verify if a configuration is still valid when a runtime version should be added or exchanged. By this way configuration verification is introduced using direct dependencies between runtime versions (see the upper left quadrant in Figure 5). To make the dependency description between runtime versions easier and more reliable interfaces or contracts can be introduced between runtime versions.

A further step is to introduce an explicit configuration selection. That means, valid software releases are defined explicitly. By this step the number of configurations which can be built into a car can be limited and a configuration verification has only to be executed during the configuration selection, when a new software release is defined. The number of distinct configurations built into vehicles can be limited, because only predefined software releases are allowed to be built or installed in a car and consequently only updates to predefined software and hardware releases are allowed. Possible other configurations which may be also valid considering the dependency descriptions are not allowed. In very large and distributed systems it is highly recommendable to reduce the delivered configurations by an explicit configuration selection in order to reduce possible sources of errors (see the right quadrant in Figure 5).

The explicit definition of software releases is also advantageous to perform updates. In the case it is known which software release is installed and to which software release an update should be performed, the difference between the installed configuration and the new configuration can be calculated and an update package can be automatically generated. This makes configuration update very efficient, because no configuration verification is required to perform an update of a configuration. If different parties or the customers themselves are allowed to install software on a single platform, it is not possible to easy determine which software release is installed. But if only one authority is allowed to manipulate a system it can be known in advance or easy determined which software release is inside a car.

The continuously increasing software volume in vehicles and the possibility to exchange software in vehicles over their lifetime requires an explicit configuration selection and the definition of dependencies based on interfaces or contracts to assist configuration selection. Furthermore the explicit configuration selection enables an efficient configuration update.

3.2 Prototypical realization of configuration management

In this section we discuss a configuration model in which automotive hardware and software releases are defined. This configuration model is used in a prototype to realize an efficient update management for automotive software.

3.2.1 Configuration model for automotive software

In section 2.1 we only discussed software-related CI (e.g. runtime versions). In this section we additionally introduce ECUs as hardware-related CI. A configuration of software-related CIs which is released is called a software release and a configuration of hardware-related CIs which is released is called a hardware release. In this section we introduce a configuration model in the form of an Entity Relationship Model (ERM). This configuration model defines automotive software and hardware releases (see Figure 6).

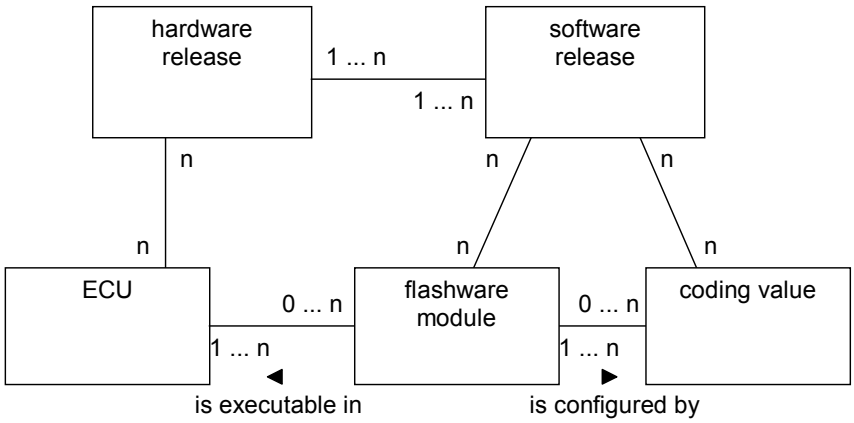

Figure 6: ERM for the definition of automotive software and hardware releases

Specific to automotive software is that the software is either located in ROM or in flash memory. As discussed in section 2.2.1, software located in ROM is inseparably connected with the hardware. This is the reason why in Figure 6 an ECU hosts 0 up to n flashware modules. Zero holds where the ECU software is located in ROM and 1 up to n holds where the software is located in flash memory. Vice versa one flashware module is executable in 1 up to n ECUs. Typically a flashware module can only be executed in exactly that hardware it is compiled for. But if there are two devices with changes in hardware that do not influence the controller architecture it is possible to have two distinct ECUs which are able to execute the same flashware modules.

Coding values are used to activate or deactivate a functionality which is implemented in a flashware module. That means if a specific functionality is not ordered by a customer it might just be deactivated by a specific coding value. These coding values are used, for example, in end-of-line programming or in workshops to adjust the ECU software to reflect the ordered functionality. A flashware module can be configured by 0 up to n coding values and vice versa a coding value can be used for the configuration of 1 up to n flashware modules.

A hardware release consists of n ECUs and a software release consists of flashware modules and corresponding coding values. The relationship between software releases and hardware releases describes which software release is executable on which hardware release. This is a relevant information for updating automotive software without exchanging hardware. This means a remote update of a software release is only possible, if the new software release is executable at the same hardware.

The discussed configuration model explicitly defines software and hardware releases and forms the basis for an update management which is implemented in a prototype and discussed in the next section.

3.2.2 Prototype

We implemented a prototype which enables the download and installation of new software releases into cars to maintain the car's software over its lifetime. The architecture of the prototype allows the transport of update packages to the car over various media as shown in Figure 7: Transport may occur by wireless links such as GSM, GPRS, UMTS, or DAB directly out of the infrastructure, by wired links in a workshop, by wireless links such as Bluetooth or WLAN from a flashware access point (maybe also in a workshop), or via portable media such as CD-ROM or USB memory stick. The main components at the vehicle side – the Flashware-Reprogramming-Controller and the Installation-Configuration-Controller run within an OSGi-Service-Gateway (a compact component framework). The Flashware-Reprogramming-Controller contains the (re)programming logic to install flashware modules in ECUs connected to the various subnetworks and the Installation-Configuration-Controller is the vehicle side part of the update management. This component is responsible for executing the update instructions of an downloaded update package and for controlling the installation procedure. We present in [19] a comprehensive description of those components and the architecture at the vehicle side.

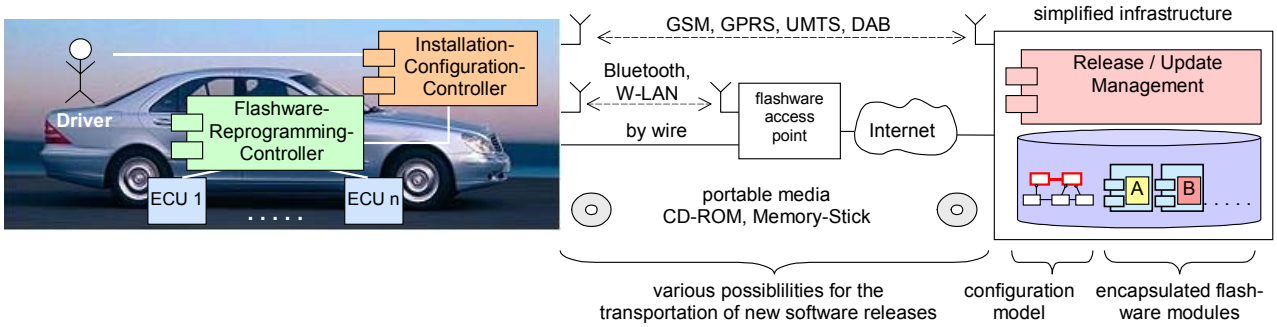

Figure 7: Main components of a software download architecture

The release management has to protocol which software release is installed in which vehicle and the update management has to calculate the difference between a new software release which has been assigned to a vehicle and the currently installed software release. An update package is generated consisting of flashware modules encapsulated in container constructs. In our prototype we used OSGi bundles as container constructs. These containers will be transferred to the vehicle, unpacked and the content – the flashware modules – will be installed in the destination ECUs.

3.3 Outlook: Towards an efficient configuration selection

For explicit configuration selection of large or/and distributed systems hierarchies are typically introduced according to section 2.1. If we consider distributed systems, the hierarchies are build by physical aspects. That means runtime versions (e.g. flashware modules) are composed to subsystems (e.g. ECUs), these subsystems are composed to larger subsystems (e.g. all ECUs connected to a vehicle sub-network). This cascading principle continues until the complete system is assembled. By introducing such hierarchies the complexity of configuration selection can be reduced significantly and each subsystem can be tested separately. This principle cannot be assigned to configuration management of automotive software straightforwardly.

Configuration management of automotive software has to consider that customers can choose within each production series from a large amount of optional equipment (the so-called features). Different features such as radar-assisted cruise control, heated front seats, etc., lead to different software releases. Consider the following example: When ordering a car, a customer has the option to equip the chosen vehicle with 0 up to 20 optional equipment features such as radar-assisted cruise control, heated front seats, Keyless-Go (the car can be unlocked without a key by carrying a chip card in the pocket and can be started by pressing a button), Teleaid (automatic call for help when an accident happens), etc.. Therefore, our customer has 2^{20} (more than one million) possibilities to

order a new car of a given production series. This also implies that more than one million software releases have to be defined, tested, and maintained.

In addition to variants which can be selected by the customer, there are further variants to consider, for example country variants. In the case of country variants, the look and feel (e. g. for a navigation system) as well as the hardware used can change dependent on the delivery market.

A customer selectable feature may require various CIs in different subsystems. As a consequence, for each subsystem containing a CI which is required for the feature, a variant has to be defined. If a lot of optional equipment features require CIs located in different subsystems a lot of variants of each subsystem have to be defined and tested. By this way the advantages of decomposing a system hierarchically into subsystems decrease and it seems to be more suitable to define software releases for the individual features.

In future work, we will tackle the challenging question whether runtime versions of automotive software should be decomposed into subsystems in a hierarchical manner or whether software releases for individual features should be defined or whether a combination of both is most suitable.

4 Summary

In this paper, we discussed configuration management with respect to configuration selection, configuration verification and configuration update. Explicit configuration selection reduces the number of configurations used in the cars due to the limited number of releases. Configuration verification uses either explicit or implicit dependencies between runtime versions, where implicit dependencies are specified by interfaces or contracts. Configuration verification is required if the number of configuration items is very high or a configuration update is performed during the lifecycle of a vehicle. If configuration selection is performed explicitly it is sufficient to execute a configuration verification during configuration selection. Moreover, configuration update profits from an explicit configuration selection - it can be performed efficiently by calculating the difference of the currently installed and the new software release. Update packages containing update instructions and runtime versions to be installed are generated automatically in order to be loaded into the car. These update packages will be installed by executing the associated update instructions. An OSGi-based prototype implementing such an efficient configuration update for automotive software has been discussed. Software and hardware releases are defined in a configuration model and update packages consisting of OSGi-bundles are generated automatically. OSGi-bundles are loaded into the car and the flashware modules encapsulated in the bundles are installed in the destination ECUs.

5 Acknowledgement

We would like to thank the master students Ralf Lohrmann, Qinghua Cheng and Volker Kugler for valuable discussions as well as our colleagues from Mercedes-Benz PKW Development and the Center for Diagnosis and Flash Technologies and in particular Prof. Dr. Wolfgang Rosenstiel of the University Tübingen and Prof. Dr. Joachim Goll of the University of Applied Science Esslingen.

6 Contact

Cornelia Heinisch: Cornelia.Heinisch@stz-softwaretechnik.de

Volker Feil: Volker.Feil@daimlerchrysler.com

Martin Simons: Martin.Simons@daimlerchrysler.com

7 References

[1] M. J. Rochkind, “The Source Code Control System”, IEEE Trans. Software Eng., vol. 1, no. 4, pp. 364-370, 1975

[2] S. I. Feldmann, “Make – A Program for Maintaining Computer Programs”, Software Practice and Experience 9, 255-265, 1979

[3] W. F. Tichy, “RCS A System for Version Control”, Software – Practice and Experience, 1991

[4] B. Berliner, “CVS II: Parallelizing Software Development”, Proceedings of the USENIX Winter Technical Conference, 1990

[5] B. A. White, “Software Configuration Management Strategies and Rational ClearCase”, Addison-Wesley, ISBN: 0-201-60478-7, 2000

[6] A. P. Dahlqvist, U. Asklund, I. Crnkovic u. a. “Product Data Management and Software Configuration Management – Similarities and Differences”, Association of Swedish Engineering Industries, 2001

[7] K. Frühauf, A. Zeller, “Software Configuration Management: State of the Art, State of the Practice”, 9th International Symposium System Configuration Management, Toulouse, France, 1999

[8] S. A. MacKay, “The State of the Art in Concurrent, Distributed Configuration Management”, Software Configuration Management: Selected Papers SCM-4 and SCM-5, Seattle, WA, April, J. Estublier ed., LNCS (1005), Springer-Verlag, pp. 180-194, 1995

[9] L. Bendix, A. Dattalo, F. Vitali, “Software Configuration Management in Software and Hypermedia Engineering: A Survey”, "Handbook of Software Engineering and Knowledge Engineering", page 523 – 548, 2001

[10] J. Estublier, “Software Configuration Management: A Roadmap”, ICSE – Future of SE Track, 2000

[11] http://www.omg.org

[12] http://www.osgi.org

[13] B. Meyer, “Object-Oriented Software Construction”, Second Edition, Prentice Hall, 1997

[14] N. Tran, C. Mingins, D. Abramson, “Managed Assertions for Component Contracts”, IDPT June 2003

[15] A. Beugnard, J Jézéquel, N. Plouzeau, D. Watkins, “Making Component Contracts Aware”, IEEE Computer, July 1999

[16] A. Hoek, D. Heimbigner, A. L. Wolf, “Versioned Software Architecture”, ISAW3, pages 73 – 76, 1998

[17] X. Chen: “Extending RMI to support dynamic reconfiguration of distributed systems”, Proceedings of the 22nd International Conference on Distributed Computing Systems, ICDCS 2002, Vienna, Austria, 2002

[18] J. Kramer, J. Magee: “The evolving philosophers problem: Dynamic change management”, IEEE Transactions on Software Engineering, vol. 16, no. 11, pp. 1293-1306, 1990

[19] C. Heinisch, M. Simons, “Loading flashware from external interfaces such as CD-ROM or W-LAN and programming ECUs by an on-board SW-component”, SAE World Conference, March 2004