Self-Configuration of Vehicle Systems - Algorithms and Simulation

Michael Dinkel
BMW Group
Research and Technology
80992 München, Germany
Michael.Dinkel@bmw.de

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

Abstract

The engineering of automotive IT-systems is confronted with the challenge of increasing complexity which will no longer be manageable with the currently used approaches of embedded systems development. Heterogeneity and a variety of new types of applications will even increase the problem in future. In this paper we present a novel approach towards self-configuration of vehicle systems. We describe the control loop and outline how this enables autonomous management of application software in embedded distributed systems. We define the configuration problem as constraint satisfaction problem (CSP) and present simulation results of different algorithms. A mapping of algorithms to the different configuration contexts of vehicles is given.

1 Introduction

Handling complexity has evolved to the main challenge for automotive IT-systems engineering. In current upper class vehicles there are up to 70 heterogeneous embedded computers (ECU)[1] which perform different concurrent control, entertainment and assistance tasks in the system. Innovative applications in the automotive domain are driven by software, which will reach a level of 90 percent in the next decade [6]. A lot of new applications like lane departure warning, emergency braking or autonomous parking is going to reach product quality soon. Even today premium cars are equipped with more than 2,000 individual functions which are implemented in software [2]. More functionality is about to come as driver assistance and driver information systems have become a major differentiating factor for competitors.

Life-cycles become more and more important in vehicle environments since the different parts of an automobile have very different periods of product life. They range from about 15 years for the complete car over a few years for certain ECUs like navigation systems to just several weeks for short-lived applications or security updates for software. As the automotive market requires innovations in shorter cycles, new approaches like self-configuration have to be introduced to the automotive world of networked control units. Self-management technologies will definitely not reduce complexity, but they have high potential to make handling of complex systems easier and more predictable for development, maintenance and usage.

The term SELF-CONFIGURATION is used in this paper to describe the continuous process of autonomously determining and enforcing valid execution assignments between software components and platforms, such that the system goals in terms of available functionality are met in the face of a changing system. Figure 2 depicts the four steps of our configuration control loop which take care that up to date software is running in the system by comparing available applications with a desired application policy.

In this paper we outline a novel approach towards self-configuration for future automotive systems and compare different configuration algorithms. In Section 2 we present our target scenario including an overview of the self-knowledge and the system architecture for self-configuration. The self-configuration control loop is the topic of Section 3. We formulate the configuration problem as CSP[2] in Section 4 which is used for the configuration algorithms and their evaluation in Section 5. We conclude in Section 7.

[1] ECU - Embedded Control Unit

[2] CSP - Constraint Satisfaction Problem

2 Target Scenario

As not only functionality but also cost is a driving force in automotive engineering, current vehicle systems have been recognized to be much too error-prone and complex due to their heterogeneity. Todays static systems are ill-suited for management at runtime and self-configuration. Hence effort is directed towards a reduction of the number of different nodes and networks in the same system. Instead of ECUs in their current, very specialized design there will be two different types of nodes in future vehicle systems. On one side there will be light-weight sensors and actuators that might even be combined with mechanical units like electronic dampers. On the other side there will be multiple nodes with higher capacity and computing resources. These nodes are no longer used exclusively for one specific purpose. Instead, they become multifunctional *platforms* able to execute different applications concurrently.

Infrastructural software enables life-cycle management for these platforms and their applications. In our target environment software is separated from hardware by an abstraction layer. A small set of powerful *platforms* executes *software components*. The platforms are nodes which execute infrastructure software and are connected via a broadband network. Platforms may provide different *capabilities* to the software components running on them. *Applications* are defined according to [3] as a sets of communicating software components which provide coherent, user-perceivable functionality. Software components can be installed, removed or updated separately and have *functional requirements* (FRs) and *nonfunctional requirements* (NFRs) which are represented in the systems *self-knowledge* together with information about the platforms in the system, their capabilities and networks between the platforms.

Functional requirements describe the communication input data of a component. FRs are modeled as *InPorts* such that a configuration is only valid if a suitable *OutPort* exists for each InPort in the system. Nonfunctional requirements describe the dependencies which components have on their executing platform. As described in [3] NFRs can be used to capture resource requirements as well as arbitrary nonfunctional dependencies. A configuration is only valid if the platform which is assigned to a component provides sufficient capabilities for each NFR of the components located there.

The *Configuration Problem* is thus to determine a valid assignment of platforms to components such that both functional and nonfunctional requirements are fulfilled.

2.1 Self-Knowledge

The *system manager* component in our architecture in Figure 1 employs the concepts of functional, nonfunctional requirements, as well as capabilities in order to build up a model of the system [3]. All the entities in the system like software components and platforms are required to contain their own description and have to make it available to the knowledge base. The knowledge base represents and stores the systems static *self-knowledge*. It collects the information about applications and components on their installation and removes the descriptions after uninstallation on the removal of the applications from the component repository. Analogously the knowledge base gathers the information about the available platforms at the time they are first attached to the rest of the system and removes their descriptions when platforms are no longer available. This behavior enables evolution of the complete system including dynamic change in the system's self-knowledge.

The web ontology language (OWL)[1] has been chosen as a suitable format for encoding the descriptions of platforms and components. From the three sublanguages of OWL we used OWL DL (for Description Logic) as OWL DL supports the maximum expressiveness while retaining computational completeness [7]. OWL DL contains certain expressions which are not part of its lesser expressive brother OWL Lite. OWL is based on the Resource Description Framework (RDF) which provides an XML based format for data representation. One main advantage of RDF-based description is the uniqueness of its classes and the relations between them. The use of uniform resource identifiers (URI) for all its constructs ensures uniqueness. In this way it can be easily applied to the structures of vehicle development since the different parts of vehicles may be developed by different suppliers. Each supplier has the opportunity to specify his modules separately from the others. A few central definitions have to be made however, for example for common channel names, message types and names of capabilities and NFRs.

2.2 Architecture Overview

In this section we give a short overview of the architecture of the self-configuring aspects of our prototype system. The aim of our research is to hide complexity from the users of automotive computing systems but also to ease the life of workshop personnel and development engineers.

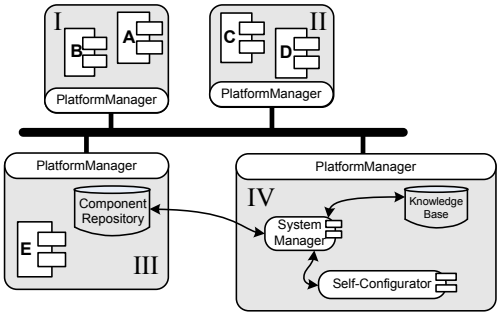

Figure 1. Architectural structure for self-configuration

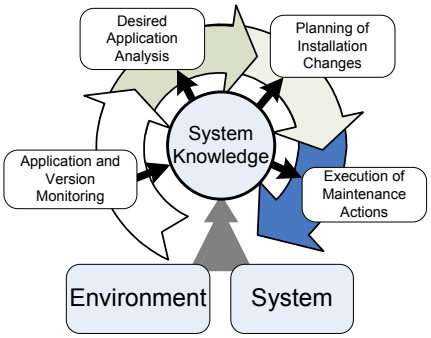

Figure 2. Self-Configuring Control Loop

Therefore we are concerned with the processes of installation, uninstallation and update of applications or single software components. Additionally, we address the evolution of such systems in terms of change in the hardware configurations like adding or removing platforms or peripherial devices attached to platforms. In our component-based system we have a *system manager* component that provides the functionality to install, update or remove application software on a system-wide basis. To accomplish this, the system manager uses the *knowledge base* to remember the systems self-knowledge. The *component repository* is used to store all software components that have been installed in the system. Additionally it caches those components that could not be installed in the current configuration. This centralized approach has been chosen to ease development; eliminating single points failure will be a subsequent, future task.

Figure 1 gives a schematic overview that encompasses four platforms and the application components `A-E`. Platform `IV` runs three components pivotal for the self-configuring system, the fourth one, the component repository is located on platform `III`. Even though three of these components are located at the same platform they could as well be distributed in the system as they use the same publish/subscribe messaging middleware [4] as all other application components.

The infrastructure of each platform contains a *platform manager* component which enables and manages local life-cycle processes like installation, removal, starting or stopping of components. The platform manager is also responsible for providing the platform description information to the system manager to be stored in the knowledge base. All platform managers are governed by the system manager which controls global life-cycle operations. The *self-configurator* uses the management functionality which the system manager provides to implement our self-configuring control loop (Figure 2). For simplicity reasons other parts of the infrastructure software and other hardware than platforms and the network have been omitted in Figure 1.

3 Self-Configuring Control Loop

In this section we present an overview of the self-configuration control loop and point out how the configuration algorithms fit into phase four. For self-configuration in vehicle systems we are concerned with whether the correct and up-to-date versions of applications are installed in the system. Development and maintenance of software may lead to updated and error corrected versions of software components. The hardware layout of the system may change due to adding or removing platforms to and from the system. The knowledge base forms the basis for reasoning in the control loop. The information stored in the knowledge base alone is not sufficient for the reasoning tasks since additional information about newly available software components and updates is needed. Hence the *self-configurator* (Figure 1) periodically fetches XML-encoded data about the availability of new software versions from an Internet server. By comparing this data with a goal policy (*desired applications list*) it can determine which applications to update. In the following we present the four phases:

Monitoring For self-configuration we assume that successfully installed software components do not change their state (`installed` / `not_installed`) unless they are uninstalled or the platform that hosted them disconnects from the system. So subject to change is given by the release of newer versions of the installed applications, by changing the desired applications list in the goals policy file, and by changing capabilities of the system as a result of adding or removing

platforms. The system manager gets informed by the system managers and can figure out whether platforms left or whether new platforms joined the system. So the self-configurator can easily monitor the available capabilities and can determine whether the current configuration is still valid. It furthermore checks whether all elements in the set of current applications are installed and whether there are applications installed that have been removed from the desired applications list. The self-configurator also checks whether new versions of the currently installed applications are available. If one of these conditions is true the analysis phase is started.

Analysis The analysis phase is closely coupled with monitoring since some of the tasks done in the monitoring phase already provide analysis results. For newer versions of applications than the currently installed ones we have to figure out what is new about them. Applications are made up of software components, so out-dating may be due to newer versions of one or more software components. The result of the analysis phase is a set of components to install and a set of components to remove from the system.

Planning Planning the actions to reach the target state where all necessary applications are installed in their desired versions works hand in hand with the analysis phase. The outcome of the planning phase is a list of declarative instructions what to do next, like "`install component x to platform y`". The process of creating these instructions is guided by a set of principles. Each component can only be installed if:

- its non-functional requirements are fulfilled by its destination platform
- its functional requirements are either fulfilled by the current system or will be fulfilled after the current installation process. This may happen when two components in a publisher-subscriber relationship are installed at the same time.

The planning phase is where the subsequently presented configuration algorithms come to work (Section 5). The algorithms have to find valid assignments and to produce a plan of actions.

Execution Once the change actions have been planned the system manager can execute the instructions in the previously arranged order. The system manager handles components and instructions on a generic level since there may be multiple different types of platforms in a heterogeneous vehicle system. Platforms and components are addressed via their RDF identifiers as specified in the knowledge base.

4 The Configuration Problem as CSP

The *configuration problem* we described in Section 2 can be modeled as a constraint satisfaction problem (CSP) as done in [8]. In the following we consider the configuration problem mainly with the focus on nonfunctional requirements. Consider variables in the set $KOMP = \{k_1, k_2, ..., k_k\}$ which represent software components and have to be assigned a value from the set of platforms $PLAT = \{p_1, p_2, ..., p_p\}$ such that the constraints in the set $\mathcal{CS}$ are fulfilled. Each platform provides capabilities $\mathcal{C} = \{c_1, c_2, ..., c_c\}$ and each component requires a certain amount of them. In order to formulate the nonfunctional requirements as constraint $NFRs \in \mathcal{CS}$ we define the function $C_i^a(x) : \mathbb{N} \longrightarrow \mathbb{N}$ that returns the amount of capability c_i which is *available* at platform p_x. Accordingly the function $C_i^r(y) : \mathbb{N} \longrightarrow \mathbb{N}$ returns the amount of capability c_i which is *required* by component k_y. The constraints of the configuration problem are given by the set $\mathcal{CS}$ that represents the application requirements. Nonfunctional requirements are expressed by the predicate $NFRs$ as follows:

$$\forall i \in \{1, 2, \ldots, c\} \wedge \forall x \in \{1, 2, \ldots, p\} \wedge \forall y \in \{1, 2, \ldots, k\}$$
$$NFRs = true \Leftrightarrow \sum_{k_y = p_x} C_i^r(y) \leq C_i^a(x) \qquad (1)$$

A configuration is only valid if the constraint $NFRs$ is true. A similar constraint can be formulated for formulated for functional requirements.

5 Configuration Algorithms

In order to solve the configuration problem we investigated two different algorithms in a prototypical implementation. We soon realized that the prototype was much too small to gain significant results. Consider that a system with 100 components and 10 platforms already has 10^{100} possible configurations. We designed a simulation environment which is able to simulate real-life scenarios for installation and reconfiguration. As the configuration problem is $\mathcal{NP}$-hard we used heuristics to gain sensible runtime behavior.

5.1 Backtracking Algorithms

Backtracking Algorithms [9] model the CSP as a search tree where in each node of the tree a variable $y \in KOMP$ is assigned to a value $x \in Plat$ such that the constraints can be fulfilled. The algorithm performs a depth-first search through the tree. If the constraints cannot be fulfilled, the algorithm has to go

back to the previous level (backtracking) and reassign different values. The backtracking algorithm has found a solution for the configuration problem if a leaf node of the search tree is reached. In the worst case the algorithm has to check all possible combinations (all nodes of the tree). Thus a *modified worst-fit heuristic*, developed in [5], has been conceived which chooses the most constraining assignment first such that it leaves as much freedom for the remaining assignments as possible.

5.2 Iterative Repair Algorithms

The second class of algorithms are *iterative repair algorithms* [9] which try to solve the configuration problem locally. These algorithms start with an arbitrary assignment even if it is invalid in the sense of the configuration problem. We used the *min-conflict heuristic* which uses a greedy strategy, it tries to minimize the number of conflicts locally and assumes that this will also lead to a globally optimal solution. The algorithm could be significantly improved by including a local minima avoidance strategy.

6 Simulation of Configurations

Our simulation environment makes use of the same architecture and knowledge representation with OWL as our prototypical implementation. We were especially interested in reconfiguration scenarios where a previously valid system becomes invalid and a new configuration has to be found. Reconfigurations may happen at runtime of future vehicles due to updates or new installation of applications and have tight real-time constraints.

6.1 Simulation Setup

Figures 3 and 4 present an example for the simulation setup. The numbers of platforms and components are parameters for our simulation environment. The simulation setup first creates the required number of platforms which provide a certain amount of the capabilities *CPU*, *RAM* and *persistent memory*.

The capabilities CPU rate, RAM size and persistent memory for each platform are initialized with values scattered randomly around a certain expected value such that each newly generated system has slightly different values.

In order to be able to implement a certain average load threshold for all platforms we use the following

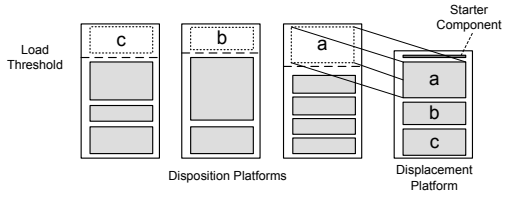

Figure 3. Configuration test setup, step 1

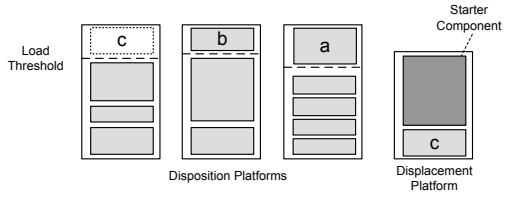

Figure 4. Configuration test setup, step 2

procedure: (1) assign components randomly to platforms, (2) randomly distribute the available resources (CPU, RAM, Memory) up to the desired load threshold. This procedure results in a system with a valid assignment for the disposition platforms and the average load is regarded while having different NFRs for the different resources. For a complete test setup one additional platform, the *displacement platform*, is created afterwards. The displacement platform is assigned to additional components. As depicted in Figure 3 the number of components on the displacement platform equals the number of previous platforms. The NFRs for the components on the displacement platform are arranged in a way that each component fits onto at least one platform of the disposition platforms. So the displacement platform could be completely cleared of components so far. The sum of NFRs of these components determines the size of the capabilities of the displacement platform. Additionally one further component is created which is be able to change its NFRs, the *starter component*. Initially the starter component has a zero consumption for its NFRs but this is changed at the beginning of each simulation run, as in Figure 3.

To start a configuration we simply increase the resource NFRs of the starter component such that a certain number of components has to be removed from the displacement platform. In the example in Figure 4 two components have to be removed. The number of components to relocate can be specified as a further parameter for each simulation run. As each component from the displacement platform can be assigned a platform of the disposition platforms it is guaranteed that a valid assignment for all components exists.

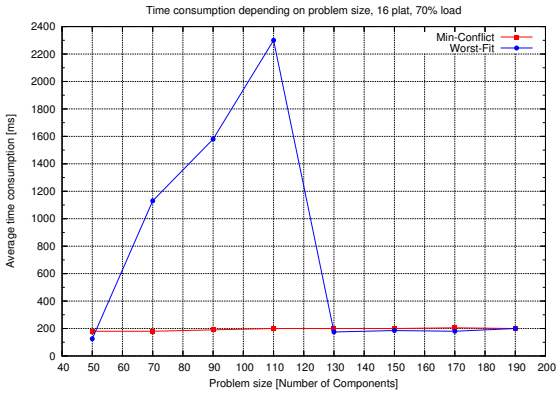

Figure 5. Time Consumption of Reconfigurations

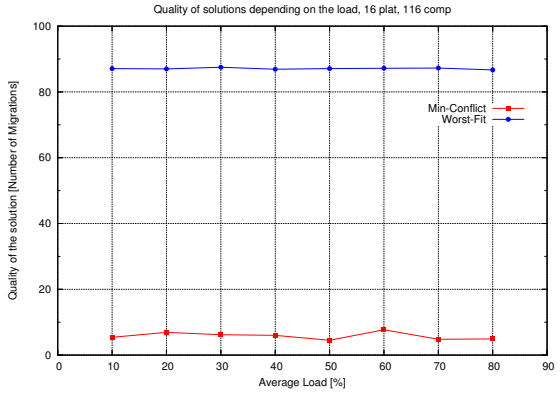

Figure 6. Quality of Solutions

6.2 Simulation Results

Some of the simulation results are displayed in the figures 5 and 6.

Figure 5 depicts a comparison of the runtime behavior of the two algorithms depending on the number of components involved, ranging from 50 to 190. The presented values are average values over 100 simulation runs each. The measurements confirmed our assumption that the runtime of the worst-fit search grows with the number of components while the min-conflict algorithm works nearly independent from the problem size. The results of worst-fit in the last four values are due to the simulation setup and show how the worst-fit algorithm works. With rising numbers of components and a constant number of platforms the components on the disposition platforms become smaller such that the components on the displacement platform become biggest in terms of resource usage at a certain point. The worst fit algorithm assigns the biggest components first and can thus find solutions quickly as it does not need to perform a backtracking.

In Figure 6 we present the quality of the configurations which were constructed by the two algorithms. We measured the quality in the number of component migrations needed more than the known solution depending on the average platform load. We used a scenario with 16 platforms and 116 components. Here the differences between the two algorithms become obvious. As the min-conflict search is a backtracking algorithm it reassigns all components to new platforms which is often unnecessary and unwanted for a reconfiguration. The local-repair algorithm with min-conflict heuristic is able to find solutions with a constantly better quality as it leaves most of the available assignments unaltered.

7 Conclusions

In our simulations we carried out multiple different runs with the aim of determining "the optimal" configuration algorithm. From the simulation results we conclude that the min-conflict algorithm is well suited for dynamic *reconfiguration contexts*, its nearly constant runtime behavior even with growing number of components provides a good solution for time-critical situations. Additionally, as the min-conflict algorithm starts with a pre-assigned system, produces only few change operations which fits well for reconfigurations. In contrast the the backtracking can be used in *clean configuration contexts* for setting up a new vehicle system, as the worst-fit sometimes produced valid assignments after a longer runtime when the min-conflict search produced none in the same time.

References

[1] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language - reference. Technical report, World Wide Web Consortium, Februar 2004.

[2] M. Broy. The 'Grand Challenge' in Informatics: Engineering Software-Intensive Systems. *IEEE Software*, 39(10):72–80, October 2006.

[3] M. Dinkel and U. Baumgarten. Modeling nonfunctional Requirements: a Basis for dynamic Systems Management. In *SEAS '05: Proceedings of the Second International Workshop on Software Engineering for Automotive Systems*, pages 1–8, New York, NY, USA, 2005. ACM Press.

[4] M. Dinkel and D. Fengler. Unified Communication in Heterogeneous Automotive Control Systems. In *Proceedings of the 3rd International Workshop on Intelligent Transportation*, pages 21–26, Hamburg, Germany,, March 2006. Hamburg University of Technology.

[5] T. Gotovac. Design and Implementation of a Self-Descriptive Component Model for Distributed Self-Organising Component Systems, Supervisor: M. Dinkel. Master's thesis, Technische Universität Darmstadt, March 2006.

[6] R. Maier, G. Bauer, G. Stöger, and S. Polenda. Time-triggered Architecture: A Consistent Computing Platform. *IEEE Micro*, 22:36–45, July/August 2002.

[7] D. L. McGuinness and F. van Harmelen. OWL web ontology language - overview. Technical report, World Wide Web Consortium, February 2004.

[8] A. Mühling. Self-Configuration in Dynamic Distributed Systems, Supervisor: M. Dinkel. Bachelor's thesis, Technische Universität München - Fakultät für Informatik, August 2006.

[9] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition edition, 2003.