Knowledge Growing Old in Reconfiguration Context

Ingo Kreuz
DaimlerChrysler AG
Research and Technology
HPC T721
D-70546 Stuttgart, Germany
ingo.kreuz@daimlerchrysler.com

Dieter Roller
University of Stuttgart
Graphical Engineering Systems Dept.
Breitwiesenstraße 20-22
D-70565 Stuttgart, Germany
roller@informatik.uni-stuttgart.de

Abstract:

A long time span often lies between the production (initial configuration) and the reconfiguration of a technical system. During that time new components are invented and related knowledge changes. The goal of this examination is to use the same knowledge base over years. One common way to solve this problem is to integrate a versioning system into the knowledge base. This article however examines a more natural and "dynamic" way of knowledge growing old. This mechanism will be integrated into a reconfiguring system for electronic systems in cars, making their extensions or repair more individual, more flexible and more actual thus increasing quality.

Introduction

Electronic systems of current cars consist of a vast amount of hard- and software modules with multiple dependencies. One function is typically spread over multiple software modules residing in multiple hardware units. In automobile context these hardware units are called ECUs (Electronic Control Units). They are interconnected through communication-buses and are responsible for executing the software modules which control all sensors and actuators connected to the ECU. To give an example, imagine pressing a button located at the driver's door to adjust the outer mirror located on the passenger's side. The button is connected to the ECU next to it, i.e. the ECU located inside the driver's door. This ECU initiates the command to start the appropriate actuator to the ECU located next to the outer mirror on the co-driver's side over a communication bus. At least two software modules are involved: One for reading the sensor inside the driver's side ECU and one for starting the motor residing in the co-driver's side ECU.

The following two scenarios underline the need for a reconfiguring system:

If a faulty module is to be replaced and an identical spare part is currently not or no longer available, a substitute has to be found. This substitute must meet all dependencies of the given system. A reconfiguring system can help finding an appropriate part even if subsequent changes are necessary. In the above example the mirror on the passenger's side could have been mechanically damaged. The smallest changeable part is the whole mirror unit consisting of motors and position sensors together with the mirror itself. The only available part in this example is a mechanically identical unit including a newer version of the position sensors, let's say with a higher resolution. Today this spare could not have been used because the software inside the ECU does not fit the signals from the sensors. The reconfiguring system instead would identify the software module responsible for position control to be changed as a subsequent step and possibly other necessary subsequent changes.

The second scenario deals with upgrades of systems. If the owner of a system wants a new functionality added, some new modules (hard- and software) have to be added or other have to be changed. To meet all dependencies subsequent changes are likely. As an example, a customer wishes to have the mirrors turned automatically to see the ground as he or she switches to reverse gear like in the new S-Class of Mercedes/DaimlerChrysler. Even if this function was not invented when the car was produced the wanted function might be added using a reconfiguring system identifying all necessary changes.

Note: Current methods for complex non-predestined changes include telling the customer to buy a new car including the wanted functionality.

However laws exist in many countries, which state a lot of expensive tests need to be performed subsequent to

changes in a car. Only after these laws are relaxed, the approach described below can be used for individual cars. But service departments can use the mechanism at least for building repair- or upgrade-packages certified once.

Reconfiguring System

The reconfiguring system basically consists of two parts. A knowledge base holds all the necessary knowledge about

- available components together with their possible parameters (hard- and software)
- dependencies between components
- strategies to exchange parts successfully
- what modules are actually on stock

We plan to use an object oriented approach to model the knowledge, like this has be done with KONWERK in the PROKON project (see [Günter et al. 1995]).

The second part is formed by an inference machine dealing with the knowledge. At this time we use the KONWERK – kernel because it provides good possibilities to add own strategies. On the other hand a pure resource based strategy is appropriate because the above mentioned dependencies mainly have their origins in the given resources of the modules and can therefore be modelled in a very adequate way (see [Heinrich et al. 1996]).

Exact configuration Onboard

To reconfigure a system, knowledge is needed about the modules currently built into the system. We have defined the structure and contents of an XML-document holding the necessary information. It will be stored inside the car and we call it the Exact Configuration Onboard (ECO, see [Kreuz et al. 1999]).

Starting the Reconfiguring Process

In [Crow et al. 1994] reconfiguration is used to obtain a FDIR[1] system. For this reason they tried to combine diagnosis and reconfiguration in systems containing standby spares. We do not presume standby spares and do not restrict ourselves to changing parts only after diagnosis, but want to use the reconfiguring system also for additions to a system.

The following starting points for reconfiguration have to be considered in our context:

First point is a part that needs to be changed (for example as the last step in a FDIR process chain). The user of the system identifies the part and the reconfiguring system removes it from the actual configuration derived from the ECO. The configuring system now detects that the system is no more complete. Completeness in this context means that the system meets the specification given by the set of functions previously included in the system and that all dependencies are valid.

The second starting point leads to the same situation: If a new function shall be added, the system is also no more complete. The incompleteness now is a consequence from an altered specification, i.e. the set of functions previously included in the system plus the newly wanted function.

In both cases the reconfiguring system tries to add a component providing the missing functionality as a composition step. It might even select the same sort of part the user has removed before, if it is still available and has not to be ordered. But it may also "decide" to select a newer version of the part or a totally different part that is currently on stock depending on the strategy and optimization rules actually used. To insure all dependencies the use of subsequent reconfiguration (see next section) is likely.

A very interesting third situation arises, if "rules" (dependencies) for the system have changed. This might happen for example because new dependencies were detected and added to the knowledge base by development departments afterwards. Also moving the system to another country having different laws might be the reason. In this case the reconfiguring system can help by determining if all dependencies are still valid and initiate subsequent reconfiguration if not.

Subsequent Reconfiguration

There are several possibilities for the reconfiguration process. In [Crow et. al. 1994] reconfiguration is modeled as an analogy to the model based diagnosis paradigm formalized by Reiter [Reiter 1987].

This article however is focused on a mechanism that tries to use as much as possible from the well researched "classical" configuration paradigm. This mechanism consists of adding components to obtain a wanted functionality and decomposes the system if a conflict pair is detected. This is very similar to the basic configuration method with backtracking in configuring systems like those described in [Günter et al. 1995] or [Heinrich et al. 1996]. The difference to normal backtracking is, that not only those parts that were previously added by the algorithm can be removed but also those, that were built into the system before the algorithm started. This is similar to "repair" described in [Günter et al. 1995].

Two phases can be distinguished as parts of the algorithm:

[1] FDIR: Fault Detection, Identification and Reconfiguration

1. *composition phase*
   The system tries to find a component that provides at least one of the wanted functionality. If more than one wanted functionality is missing, the reconfigurator has to decide which component is to be added next. This might be the component providing the highest number of missing functions or the component causing the least number of conflict pairs for example. This phase ends when conflict pairs occur or when the system is complete, relative to a given specification.

2. *"backtracking" or decomposition phase*
   The system removes one components that is part of a conflict pair or group. The addition of one component in phase one can cause multiple conflicts, so this phase basically consists of a loop
   - identifying the components that are part of at least one conflict pair or group
   - choosing the next component to be removed, hence meeting some optimization criteria (strategy). For example it might be useful to remove those components first, that are involved in the most conflict pairs.
   - removing the identified component

This phase ends when there are no more conflict pairs.

Phase 1 and 2 are applied in a loop that ends when the system is complete relative to a given specification after phase 1.

The Role of Versions

As mentioned in the beginning, there is one thing to be remarked in technical systems, if they are to be serviced over a longer time span: Due to proceedings in development new components are invented, normally having some advantages compared to those built into an older system. If a component of the older system is to be changed, it may be the case, that only newer versions of the part exist as spares. Normally a newer version replaces an older version completely and often holding all old versions as spares is more expensive for a company than accepting some subsequent changes using the new version in older technical systems, too. In many cases this is even cheaper for a company if the subsequent changes are made cost transparent to customers, because storing spares is that expensive.

The reconfiguring system described above can help to manage versions of components in two ways:

First off it can test if any conflicts occur using the newer version and also it helps to solve the conflicts by finding alternative components providing the same functionality.

Secondly it can identify all parts that should be replaced by a new version (maybe even if they still work) if a constraint is added to the knowledge base saying that the older versioned part is a conflict by itself.

These two aspects are fulfilled without any extra programming by the algorithm described in the above section. The component of the new version is to be added to the knowledge base as an available part and the older version has to either be removed from the knowledge base or be marked as unavailable.

Age coming into Play

Versions of components are not the only thing that change over years. Knowledge about dependencies or (re-) configuration strategies change, too.

From a standpoint based on human experience it is often good to prefer things and knowledge that are new and therefore up to date. As a result human brain is able to forget things "automatically" if they were not used for a long time. For the same reason more recent knowledge is preferred to old fashioned experiences by humans.

For our reconfiguration process having aged knowledge can also be an improvement (see below). Additionally it would be more natural, if both components and knowledge would continuously and automatically change instead of releasing a new version for the item at one time.

Improving the Knowledge Base

One advantage of having aged knowledge is that very old knowledge that was not used for a certain time can be removed from the knowledge base automatically. This keeps the knowledge base small and the configuration engine fast as a consequence. This process is analogous to the behavior of human brain mentioned above.

Of course there might occur special situations when old knowledge is needed. For example if one tries to rebuild an old-timer car with original old parts. For this reason it is useful to simply use backups of the knowledge base enabling to go back to whatever date is wanted.

Improving the Configuration Process

A second and maybe more important issue for having knowledge growing old is to speed up the configuration process. Learning from the behaviour of human brain, it seems very probable that knowledge, that helped very often to find a solution and knowledge that is very much up to date, helps best to solve an actual problem. As a consequence it is very near at hand to use this kind of knowledge first. Knowledge from that quality is furthermore called *actual*.

In detail this means for the (re-) configuration that

- actual components are preferred
- actual strategies are preferred
- actual dependencies are tested first

to speed up the process.

If a case based paradigm is used, determining actuality also helps to find solutions that are not as conservative as they would be following the original paradigm. The problem of being conservative is described in [Cunis et al. 1991] as a result of having the main reason for decisions that something has be done before in a similar way.

Actuality /Age of Knowledge

At this time the following suggestion is given to determine the actuality of knowledge. It is derived from the above thoughts about knowledge growing old in human brain and is consistent to every day experience:

$$\text{actuality} = \frac{\#\text{useful}}{\text{age}}$$

where

#useful is the number of times, when the knowledge have led to a solution or a component has been part of a solution

age stands for the number of time units used for the system (for example hours, days, weeks or years) since the knowledge / component was stored into the knowledge base originally

Without any limitations we can initialize the value of actuality for an item with 1 by initializing *#useful* as well as *age* with 1.

For the reason that only knowledge of the same kind is to be compared by actuality, no constants are required in the above formula as long as the same measurement is used for something's age: Actuality is a relative number which increases based on how often the knowledge has been useful and it decreases with age as expected. If an item is not used for a long time span, actuality of this item approaches 0 and can be removed.

Nevertheless it seems practical to allow weights for #useful relative to the age of an item. The following definition for actuality allows to influence the significance of age relative to #useful by introducing a constant c:

$$\text{actuality} = \frac{\#\text{useful}^c}{\text{age}}$$

where

$c \in]0,1]$ to increase the significance of age

$c \in [1,\infty[$ to increase the significance of #useful

Note: c is the exponent because if c was a multiplier, there would be no effect to the weight of #useful *relative* to age.

To be able to calculate the actuality, #useful and the date of birth are to be stored for all knowledge and components residing in a knowledge base.

The constant c can be varied by the user for each class of items in a knowledge base. The actuality of items is only comparable, if the same constant c was used for all of them.

The following three figures show actuality for different values for constant c.

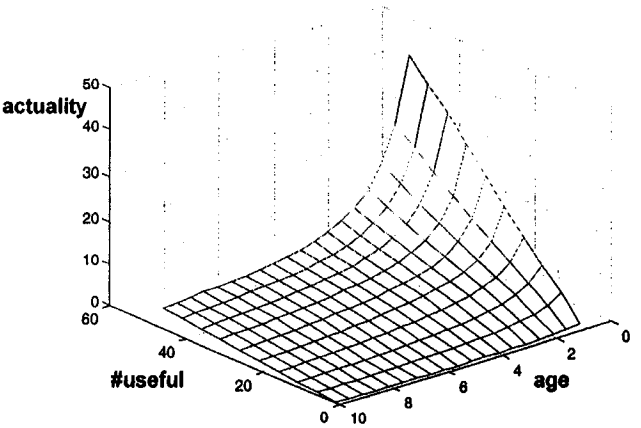

figure 1 actuality calculated for c=1

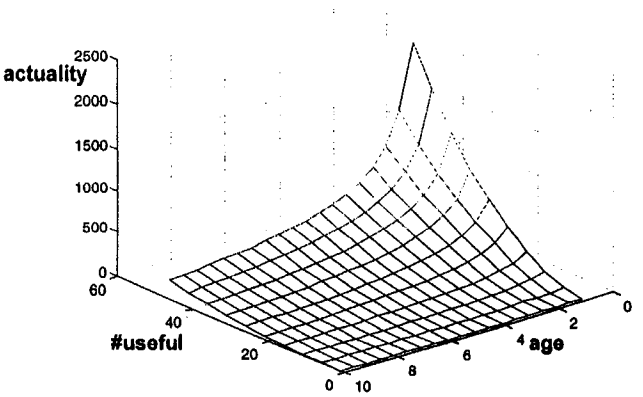

figure 2 actuality calculated for c=2

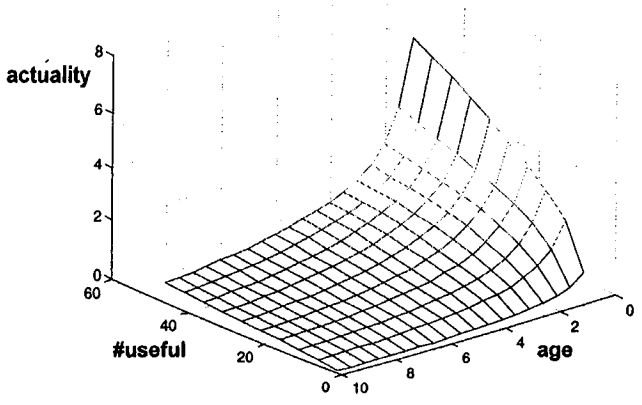

figure 3 actuality calculated for c=0.5

Summary and Conclusion

This paper described the use of a reconfiguring system for electronic systems of vehicles in consequence to supplementing new functions to or exchanging components of a vehicle. An algorithm for reconfiguration using techniques of well researched configuration was presented, age/actuality of knowledge was introduced and improvements resulting from having age/actuality for knowledge were discussed.

As a next step it is planned to implement a prototype using these mechanisms proving the mentioned advantages. This prototype will also enable fine tuning for the use of "actuality" by finding appropriate values for the mentioned constant c.

References

[Crow et al. 1994] Judy Crow and John Rushby: *Model-Based Reconfiguration: Diagnosis and Recovery*, Computer Science Laboratory, SRI International 333 Ravenswood Avenue, Menlo Park, CA 94025 USA, NASA Contractor Report 4596, May 1994

[Cunis et al. 1991] Roman Cunis, Andreas Günter, Helmut Strecker: *Das Plakon-Buch. Ein Expertensystemkern für Planungs- und Konfigurierungsaufgaben in technischen Domänen.* Springer-Verlag, Berlin Heidelberg, 1991, pp. 131

[Günter 1995] Andreas Günter (ed.): *Wissensbasiertes Konfigurieren. Ergebnisse aus dem Projekt PROKON.* Infix-Verlag, Sankt Augustin, 1995.

[Heinrich et al. 1996] Michael Heinrich, Ernst Jüngst: The Resource-Based Paradigm: *Configuring Technical Systems from Modular Components* in AAAI-96 Fall Sympos. Series: Configuration. MIT, Cambridge, MA, November 9-11, 1996, p. 19-27.

[Kreuz et al. 1998] Ingo Kreuz, Thomas Forchert, Dieter Roller: *ICON. Intelligent Configuring System* in Dieter Roller (ed.) *Proceedings of the 31th ISATA, Volume "Automotive Electronics and New Products"*, Düsseldorf Trade Fair, Croydon, England, 1998, p. 219ff..

[Kreuz et al. 1999] Ingo Kreuz, Ulrike Bremer: *Exact Configuration Onboard. Onboard Documentation of Electrical and Electronical Systems consisting of ECUs, Data Buses and Software*, ERA conference 1999, Coventry. To Appear, p. 5.2.1 ff

[Reiter 1987] Raymond Reiter: *A theory of diagnosis from first principles.* Artificial Intelligence, 32(1): 57-95, April 1987.