THE ADAPTABILITY CHALLENGE FOR EMBEDDED CONTROL SYSTEM SOFTWARE

V. Cechticky[1], A. Pasetti[1,2], W. Schaufelberger[1]

[1]Institut für Automatik, ETH-Zürich, Physikstr. 3, Zürich, CH-8092
[2] P&P Software GmbH, Physikstr. 3, Zürich, CH-8092
cechti@control.ee.ethz.ch; pasetti@pnp-software.com; ws@control.ee.ethz.ch

Abstract: Software-related costs account for a growing share of total development costs for embedded control systems. In the control field, containment of software costs can be done either through the use of model-based tools (e.g. Matlab) or through a higher level of reuse. This paper argues that the second strategy is advantageous in the case of industrial control systems targeting niche markets where systems tend to be one-of-a-kind and where they can be organized in "families" of related applications. The paper then argues that progress in raising the level of software reuse in these fields depends on the adoption of better software adaptability techniques. The most promising such techniques are reviewed from the standpoint of control engineers.

Keywords: software reuse, adaptation, control systems, object-oriented programming, components

1. INTRODUCTION

In addressing the problem of how to contain software-related costs for embedded control systems, the first question to be answered is whether there is anything "special" about these systems that justifies treating them separately from other categories of applications. Our position is that this is indeed the case and that control applications pose two challenges to software designers that are specific to them and set them apart from other applications (both in the embedded and non-embedded world). The first one is the *Non-Functional Challenge.* Embedded control systems are typically subject to non-functional requirements (covering issues such as timing, dependability, availability, etc.). Their correct implementation requires the use of techniques that allow non-functional as well as functional aspects to be modelled. The second challenge is the *Variability Challenge.* Embedded control applications are often built as one-of-a-kind systems that can be seen as instances of families of related applications. Their efficient development requires the use of techniques that can model entire families rather than just individual applications.

The non-functional challenge has long been recognized. It is addressed by several research groups around the world (see the special issue of (Sastry, *et al*., 2003) for a survey) and very considerable progress has been made in developing techniques that handle non-functional, and in particular timing, problems. The variability challenge has instead long gone unnoticed. Its more recent recognition means that it is likely to be the major source of technical improvements in the near future. It is also likely to have the sharpest impact on development costs because the ability to exploit commonalities among different applications to reduce duplication of development has an obvious and clear impact on software costs at all levels (from design down to testing and servicing). In this paper, it is argued that one of the keys, or perhaps even *the* key, to addressing the variability challenge lies in the development and application of more effective adaptation techniques for control system software.

2. MODEL- VS. REUSE-DRIVEN APPROACH

Two approaches have emerged to tackle the variability problem in the embedded world: the *model-driven* and the *reuse-driven* approaches. With the former approach (see Figure 1), the application requirements are expressed in a modelling environment that is capable of automatically

generating the application code. The prototypical example of such an environment is the Matlab tool suite. The increase in efficiency arises from the fact that the software design and implementation phases are automated and that the control engineer can take direct control of the software development process without having to resort to the intermediary services of a software engineer.

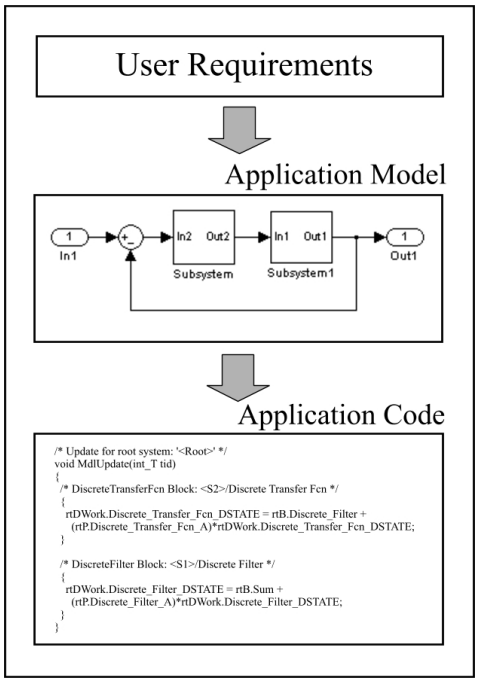

Fig. 1: Model-Driven Approach

In a reuse-driven approach instead (see Figure 2), the application code is built by configuring and composing a set of pre-defined software building blocks. The increase in efficiency now arises from the possibility of reusing existing software artifacts (modules, components, code fragments, etc.). Traditionally, the reuse-driven approach was implemented by developing libraries of reusable modules. More recently, software product families (Bosch, 2000) and software frameworks (Fayad, *et al*., 1999; Gamma, *et al*., 1995; Pasetti, 2002) have emerged as more convenient reuse vehicles that allow reuse to take place at architectural as well as at the code level.

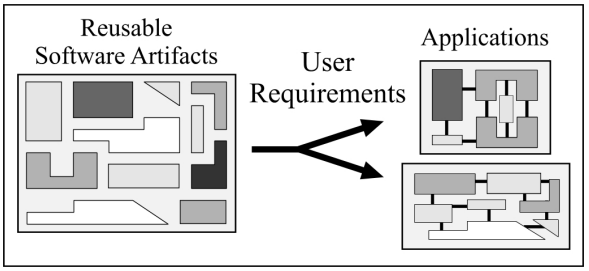

Fig. 2: Reuse-Driven Approach

In the control community there is a widespread belief that the model-driven approach has won the day and that tools like Matlab are on the verge of solving the software problem for control systems. In our view, this is a misconception. Model-driven tools owe their power to the fact that they offer high-level abstractions that are relevant to their users. Matlab, for instance, supports the direct expression of concepts like "transfer function", or "PID", or "state machine" that control engineers can directly use to express their needs (as opposed to having to code them in lower level languages such as C). The price paid for this power, however, is narrowness of focus. Model-driven tools are designed for use within a certain domain and their effectiveness declines very steeply as one moves away from that domain. The problem for control engineers is that their applications tend to be multi-domain. A complete control application does not simply cover implementation of control laws. In fact, in most cases, the implementation of control laws – the specific domain of Matlab – is only a small fraction of the total control software [1]. Most of the software normally is concerned with functionalities such as management of external sensors and actuators, management and generation of housekeeping data, management and processing of commands from some supervisory unit, implementation of failure detection and identification logic, implementation of failure recovery actions. Matlab-like tools are ill-suited to cover these functionalities (or, at any rate, they are not better suited than general-purpose languages). A reuse approach may then be more appropriate. The cost of developing reusable building blocks is lower than the cost of developing model-driven tools and a reuse-driven approach is therefore affordable even for niche products – as most industrial control systems are. The reusable blocks can moreover be more easily targeted to the specific needs of their users and can therefore more easily match the often idiosyncratic needs of control applications.

3. REUSE AND ADAPTABILITY

To reuse a software asset (a component, a fragment of code, a design model, etc.) means to use it in different operational contexts. In practice, different operational contexts will always impose different requirements on the reusable assets. Hence, effective reuse requires that the reusable assets be *adaptable* to different requirements. In this sense, adaptability is the key to reusability and the availability of software adaptability techniques is the necessary pre-condition for software reusability in domains like industrial control systems where there is a high degree of product variability and where individual products must be tailored to their operating environment.

A practical and effective reuse-driven approach must therefore take the form shown in Figure 3. The reusable assets are organized in a repository. The repository covers the need of a particular (and often

[1] We have experience with the development of software for satellite control system where, typically, control algorithms take 20-30% of the total software.

narrow) domain. Applications within the domain are constructed by selecting items from the *repository*, tailoring them to the needs of the application by passing them through an *adaptation phase*, and finally assembling them to create the target application (Cechticky, *et al*., 2003). The approach shown in Figure 3 is often called *product family approach* and is arguably the most successful way to achieve software reusability in an industrial context.

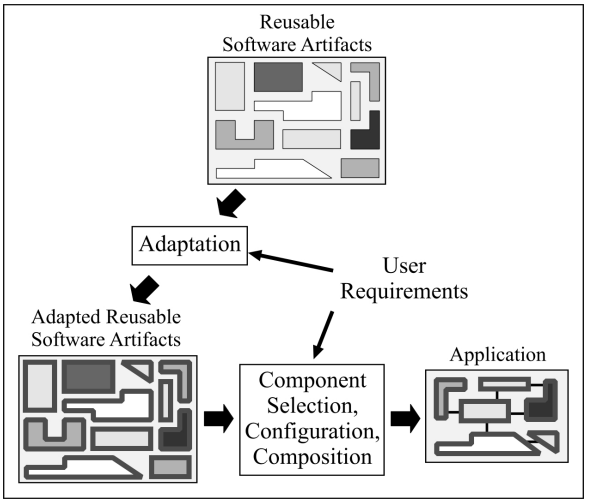

Fig. 3: Reuse through Adaptability

Software reuse is perhaps the oldest approach to the reduction of software costs and has often been tried in the past. Past attempts, however, had only mixed success primarily because they either ignored the adaptation phase shown in Figure 3 or because the state-of-the-practice techniques available at the time were not sufficiently powerful to model the variability in the target domain. As a result, the software of many – if not most – industrial control applications is still crafted by hand. This paper argues that this is unnecessary because recent advances in software engineering have brought very powerful adaptation techniques within the reach of mainstream applications. A family-based reuse-driven approach has therefore become possible and advantageous even for niche domains. The next sections of this paper give an overview of the main adaptability techniques and discuss their relevance to control applications. Two categories of adaptability techniques are recognized. The first one looks at techniques to *model* adaptability. The second one looks at techniques to *implement* adaptability mechanisms. Section 4 consider feature modelling that is the most prominent of the adaptability modelling techniques whereas Sections 5 and 6 consider object-oriented software frameworks and aspect oriented programming which are the most powerful adaptability techniques available at present. It should be stressed that all three technologies discussed in the next sections are mainstream technologies and sufficiently mature for use in an industrial context.

4. MODELLING ADAPTABILITY

One of practical obstacles to the adoption of a reuse-driven approach is the management of the reusable assets. On the one hand, one would like to have as rich a repository of reusable items as possible (to increase the coverage of the repository) but, on the other hand, a large number of reusable items makes it difficult for the user to select those that are relevant to his particular needs. There is a point where the cost of selecting the reusable items defeats the purpose of reuse. Feature modelling (Kang, *et al*., 1990) provides one way to address this problem.

In general, *feature models* (Beuche, *et al*., 2003; Cechticky, *et al*., 2004; Czarnecki and Eisenecker, 2000) are a means to model the variability and multiplicity of configurations of a certain system. In our context, they can be used to describe the features of the potential applications that can be instantiated from a repository of reusable software assets. Feature models are usually represented graphically as tree-like structures where each node represents a feature and each feature may be described by a set of sub-features represented as children nodes. Various conventions have been evolved to distinguish between mandatory features (features that must appear in all applications instantiated from the repository) and optional features (features that are present only in some applications instantiated from the repository). Limited facilities are also available to express constraints on the legal combinations of features.

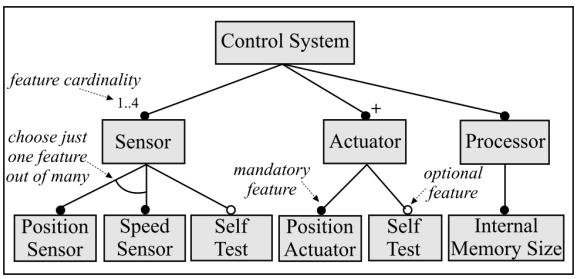

Fig. 4: Feature Model Example

Figure 4 shows an example of feature diagram representing a (much simplified) family of control systems. The diagram states that all control systems in the target domain have a single processor, which is characterized by its internal memory size, and have one to four sensors and one or more actuators. Sensors and actuators may have a self-test facility (optional feature). Sensors are either speed or position sensors whereas actuators can only be position actuators. In general, feature models like the one in Figure 4 can be built for each repository of reusable items and prospective users of the repository can then use the feature model to specify their application by ticking off the features they want. This specification process guarantees that the application will be within the domain of the repository.

A feature model approach is especially beneficial in the control domain where the control engineer is not necessarily a software expert. The use of the feature model allows him to make use of the repository of reusable software assets with only a limited understanding of its structure.

5. OBJECT-ORIENTED FRAMEWORKS

The concept of product family introduced above (see Figure 3) is very generic. In particular, it does not say anything about the nature of the reusable assets (are they components, subroutines, code fragments, or models?) or about their mutual relationships (are they completely independent of each other or are they embedded within some architecture?). In practice, there is now a consensus that repositories of reusable software assets can be effective only if some "structure" is imposed upon them. Software frameworks offer a particular way to organize the items in the reusable repository and hence provide precisely such a structure.

In the case of a framework approach, the items in the repository are abstract interfaces (namely definitions of abstract services that the applications must provide) and components providing default implementations for those interfaces (Blum, *et al.*, 2003). The interfaces, taken together, define an architectural skeleton that is shared by all applications instantiated from the framework. The chief virtue of a framework is thus its ability to raise the level of reuse from that of mere code fragments to that of an entire architecture. In predefining an architecture optimised for applications in their domain, software frameworks go beyond subroutine or class libraries because they make available not just individual modules but also the relationships between them. Subroutine and class libraries, on the other hand, are generic artifacts that can be used in a large variety of applications whereas frameworks are targeted at a specific – and often narrow – domain. They aim at depth rather than breadth of reuse.

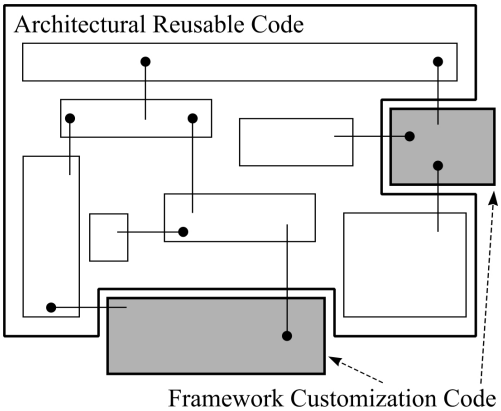

Fig. 5: Software Framework Structure

The typical representation of a software framework is shown in Figure 5. The unshaded area represents the architectural backbone shared by all applications in the framework domain. The framework captures this architectural backbone and makes it available to application developers who adapt it to their needs by plugging into the framework components that implement the application-specific behaviours (the darker boxes in the figure).

Since a software framework exists primarily to be adapted, its quality depends essentially on the ease with which the artifacts it offers can be adapted to match the needs of its users. Software frameworks are usually categorized on the basis of the adaptation technology they use. Virtually all frameworks built in recent years are object-oriented in the sense that they use *inheritance* (Figure 6) and *object composition through abstract coupling* (Figure 7) as their chief adaptation techniques. In the former case, the behaviour of a reusable component is tuned by extending it through inheritance. In the latter case, it is tuned by letting the component delegate the variable part of its behaviour to an external component that is characterized through an abstract interface.

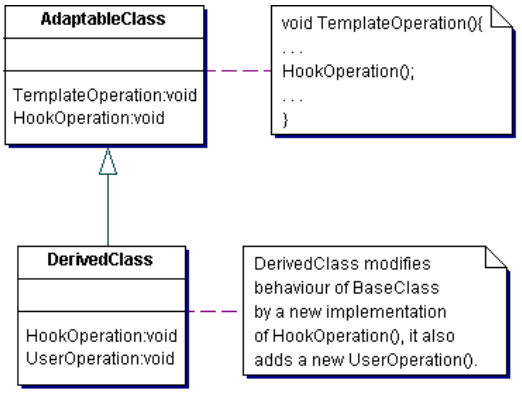

Fig. 6: Component Adaptation through Inheritance

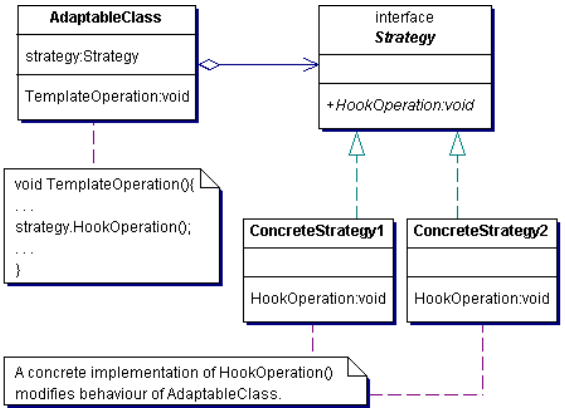

Fig. 7: Component Adaptation through Composition

From the point of view of a user – and in particular a user in the control domain – the main point about these adaptation techniques is that they allow the behaviour of a reusable component to be adapted without touching its source code. Control applications are often mission-critical and their software must normally undergo some kind of

qualification process. The fact that adaptation can be achieved without touching the source code of the reusable component means that the component can be qualified only once and can then be reused without having to be re-qualified. In a sense, object-orientation allows the qualification process as well as the code of a component to be reused. Given the cost of software qualification processes, this is an important advantage.

6. ASPECT-BASED ADAPTABILITY

As indicated in Section 1, one of the key problems in the design of control system software is the presence of non-functional requirements covering issues such as timing, reliability, observability, testability, and so forth. Thus, in the control domain, adaptability techniques must also cover adaptation in the non-functional aspects of the behaviour of reusable components. The object-oriented techniques outlined in the previous section are unfortunately inadequate in this respect because they can only be used to tune the functional part of the behaviour of a component. The lack of tools to model non-functional adaptability was one of the prime causes of the low level of reuse in the control domain. Recently, Aspect Oriented Programming (AOP) has emerged as a remedy for this problem.

Aspect oriented programming (AOSA, 2004; Birrer, *et al*., 2004) is a software paradigm that promotes the application of the principle of separation of concerns to the non-functional aspects of a software system. At the most basic level, aspect oriented techniques can be seen as a means to perform automatic transformations of some base source code. An aspect oriented environment consists of two primary items: an *aspect language* and an *aspect weaver*. The aspect language allows the non-functional aspects to be specified and encapsulated in self-contained modules. The aspect weaver is a compiler-like tool that reads an aspect program and projects the changes it specifies onto some base code. This process is illustrated in Figure 8.

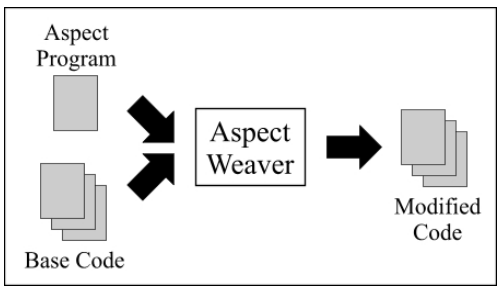

Fig. 8: Aspect Oriented Programming Environment

An example may illustrate the relevance of AOP to control systems. Control applications are normally subject to real-time constraints. Unfortunately, even within the same domain, there is usually no single real-time architecture that is adopted by all control applications. In particular, some applications privilege simplicity over efficiency and opt for a non-preemptive scheduling approach whereas others use some form of preemptive scheduling in order to optimize the use of CPU resources. This lack of a single real-time model poses a problem for reuse-driven approach because the real-time model has an impact on the implementation of the reusable components. Consider, for instance, the problem of synchronization code. If a non-preemptive real-time architecture is assumed, then there is no need to have synchronization code in the framework components. If, on the other hand, preemptive scheduling is allowed, then the framework components must be endowed with synchronization mechanisms that ensures that they can are accessed in mutual exclusion.

The traditional solution to this problem is for the reusable components to make the worst case assumption and implement synchronization mechanisms. This forces some users to carry a great deal of excess baggage and to suffer the associated memory occupation and execution efficiency penalties. If the implementation language is C/C++ (a common choice in the control domain), there is the additional difficulty that real-time facilities are not provided within the language and hence their implementation depends on the operating system. This means that the reusable components become dependent on a particular choice of operating system.

The alternative solution is based on the use of AOP techniques. In this case, the reusable components are developed without any regard to the real-time model (they only implement the functional part of the application behaviour) and the real-time model is specified separately in an *aspect component*. The user can then choose the functional and aspect components independently from each other and can generate the deployable component (containing both the functional and the real-time behaviour) by merging them using an aspect weaver.

AOP is a recent technique and its support tools are targeted at desktop applications. In particular, they are difficult to use in a context – such as that of many control domain – where the software has to undergo a qualification process. This is due to the fact that the merging between the base code and the aspect program is normally done at the level of object code. This would make the qualification of the modified code very difficult (because one would have to qualify code for which no source code is available). In recent work, however, we have addressed this problem and have developed an aspect oriented environment that operates at the level of source code and that therefore is well-suited to qualifiable applications programming (Birrer, *et al*., 2004).

7. SUMMARY AND APPLICATIONS

This paper has presented the case for a reuse-driven approach to the development of the software for control systems. This case is especially strong in

niche domains where applications tend to be one-of-a-kind and where applications can be organized in "families" of related applications. Under such conditions, the development of model-driven tools can be too expensive and the use of commercial model-driven tools such as Matlab is inadequate to cover all the needs of the family. A reuse-driven approach can then be advantageous. The paper then argued that reusability at the software level depends crucially on adaptability and it presented three techniques that foster the use or the development of adaptable software assets. Feature modelling techniques allow adaptability to be modelled from the point of view of the user. They provide a tool which allows a control engineer to specify his application in terms of the features offered by the reusable software asserts. Object-oriented frameworks raise the level of reuse from the traditional one of mere code fragments to the level of an entire architecture with correspondingly greater cost savings. Object-oriented techniques in particular allow components to be adapted without modifying their source code. This is a valuable feature in a field like control engineering where modifying source code is a very expensive process due to qualification requirements. Finally, aspect oriented techniques allow adaptability with respect to the non-functional properties that so often characterize control systems.

The above techniques have been developed in the last decade or so in academic or research settings but they are now mature enough to be considered in industrial applications. We have recently tested their maturity with the development of the OBS Framework (P&P Software, 2004) which uses all three techniques discussed in this paper. The range of applications that can be instantiated from the framework is described by a feature model. The framework is built as a set of components that use both inheritance and object composition to be adapted to match the needs of target applications. The framework components only encapsulate functional behaviour. Adaptation with respect to timing requirements is done through aspect programs. Finally, the OBS Framework offers facilities to integrate Matlab-generated code. It thus demonstrates that the two alternatives presented in Section 1 – the model-driven and the reuse-driven approaches – are not mutually exclusive.

REFERENCES

AOSA (2004). Homepage of the Aspect-Oriented Software Development. http://aosd.net/

Beuche, D., H. Papajewski and W. Schröder-Preikschat (2003). Variability Management with Feature Models. In: *Proceedings of the Software Variability Management Workshop*, p. 72-83.

Birrer, I., V. Cechticky, A. Pasetti and O. Rohlik (2004). Implementing Adaptability in Embedded Software through Aspect Oriented Programming. In: *Proceedings of International IEEE Conference Mechatronics & Robotics '04*, p. 85-90. Sascha Eysoldt Verlag.

Blum, A., V. Cechticky, A. Pasetti and W. Schaufelberger (2003). A Java-Based Framework for Real-Time Control Systems. In: *Proceedings of 9th IEEE International Conference on Emerging Technologies and Factory Automation*, p. 447-453. IEEE and UNINOVA.

Bosch, J. (2000). *Design and Use of Software Architectures – Adopting and evolving a product-line approach*. Addison-Wesley Professional.

Cechticky, V., P. Chevalley, A. Pasetti and W. Schaufelberger (2003). A Generative Approach to Framework Instantiation. In: *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, p. 267-286. LNCS Vol. 2830, Springer-Verlag.

Cechticky, V., A. Pasetti, O. Rohlik and W. Schaufelberger (2004). XML-Based Feature Modelling. In: *Software Reuse: Methods, Techniques and Tools: 8th International Conference*, p. 101-114. LNCS Vol. 3107, Springer-Verlag.

Czarnecki, K. and Eisenecker (2000). *Generative Programming – Methods, Tools, and Applications*, Addison-Wesley.

Fayad, M.E., D.C. Schmidt and R.E. Johnson (1999). *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, John Wiley & Sons.

Gamma, E., R. Helm, R. Johnson and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Kang, K.C., S.G. Cohen, J.A. Hess, W.E. Novak and A.S. Peterson (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. In: *Technical Report No. CMU/SEI-90-TR-21*, Software Engineering Institute, Carnegie Mellon.

Pasetti, A. (2002). *Software Framework and Embedded Control Systems*, LNCS Vol. 2231, Springer-Verlag.

P&P Software (2004). The OBS Framework. http://www.pnp-software.com/ObsFramework/

Sastry, S., J. Sztipanovits, R. Bajcsy and H. Gill (2003). Special Issue on Modeling and Design of Embedded Software. *Proceedings of the IEEE*, **Vol. 91 No. 1**.