Embedded Software Market Transformation Through Reusable Frameworks

Wolfgang Pree, Alessandro Pasetti

Software Research Lab
University of Constance, D-78457 Constance, Germany
firstname.lastname@uni-konstanz.de
www.SoftwareResearch.net

Object-oriented frameworks are a software reuse technology that fosters reuse of entire architectures and which has made software reuse a reality in many domain areas. Like other advanced software techniques, however, framework technology has seldom been used in the embedded domain. This paper argues that its application to embedded (control) systems is technically feasible and liable to bring to them the same benefits it has already brought to other domains. The description of a prototype framework for satellite control systems corroborates the argument. It is then argued that software frameworks, when combined with other enabling technologies, have the potential to standardize various aspects of embedded software and to transform the embedded sytems market.

1 Project culture versus product culture

Classical software development strategies do not focus on the reusability of software components. Bertrand Meyer [Mey89] remarks that “object-orientedness is not just a programming style but the implementation of a certain view of what software should be—this view implies profound rethinking of the software process.” Thus we might discern between two views or cultures:

– The conventional culture is *project-based*: the classical software life-cycle or some of its variations have the goal of solving *one* certain problem. The primary question addressed in the analysis and design phases is “What must the system do?”, followed by a stepwise refinement of a system’s functions.
– The preconditions of a *product culture* are object-oriented development techniques, in particular object-oriented frameworks. This culture yields not only software systems that solve *one* certain problem, but reusable software components that are useful for a potentially large number of applications.

A set of ready-made reusable/adaptable software components also influences the system specification. In contrast to the project-based culture, where a software system is developed to satisfy specific needs, good frameworks typically capture the existing “best practices” in a domain. For example, the SAP system represents a framework that—although developed with a non-object-oriented language—standardizes significant portions of how companies conduct business, covering areas such as accounting, human resource management, production planning and manufacturing. The SAP framework can be adapted and fine-tuned to the specific needs of companies. Though the effort required to adapt SAP to a specific company is significant and some adaptations are not feasible, the defacto domain standardization is part of SAP’s success story.

In other words, instead of slavishly adhering to the user’s or customer’s requests in the realm of a conventional custom-made system construction process, the system specification inherent in a framework most likely provides a somewhat different functionality (for example, without some nice-to-have features), which can be created by means of existing framework components. The customer has the choice between a custom-made system implemented (almost) from scratch with significantly more effort and cost, and a system built out of ready-made components by adapting a framework. The framework has the additional advantage that the quality of the resulting system in terms of reliability will likely be higher than the custom-made system. This is particularly true if the framework has been thoroughly tested and/or has already been reused several times. Some offers might be hard to refuse.

From an organizational persepective, successful application of framework technology implies a more integrated approach to software development that breaks down the traditional barrier between the application and the software specialists. The way how SAP systems are adapted to specific needs corroborates this change in the development process.

Other domains where a standarization through frameworks has lead to a product culture are graphical user interfaces and Internet applications. Fayad et al. [Fay99] present successful frameworks in various domains. We see no reason why a standardization of real-time embedded systems software should not be feasible.

2 Object-oriented frameworks for reuse and flexibility

Over the past decade object technology has gained widespread use in software development. Overall, three essential concepts comprise object technology: information hiding, inheritance/polymorphism and dynamic binding. The mixing ratio of these ingredients defines the flavors of object technology. Object-based systems stress information hiding. Object-oriented programming is often described as *programming by difference* as it adds inheritance and dynamic binding to the object-based paradigm. Nevertheless, the object-based and object-oriented paradigms do not automatically enhance reusability and exentsibility. The coupling of components[1] often manifests in the source code. Figure 1 schematically outlines the problem. If the right hand component should work with another component, its source code has to be changed.

[1] software component:= a piece of software with a programming interface; ideally deployable as unit. Classes and modules are examples of software components.

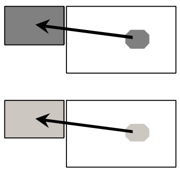

Fig. 1. Components with glue code inside.

Go the extra mile—define abstractions! Object-oriented frameworks form a special breed of object-oriented systems with extensibility as key characteristic [Font01]. The precondition for achieving extensibility and as a consequence plug&work composition capabilities are domain abstractions. They become abstract classes or Java/C# interfaces. Conceptually, domain abstractions form plugs. Polymorphism and dynamic binding allow a compositional adaptation of these components without having to change the source code. Figure 2 illustrates the difference to the previous solution: It depends on the dynamic type of the plugged in component which implementation of m1() is executed. The source code which calls m1() remains unchanged. This call-back-style of programming manifests in function/procedure parameters in conventional languages. Object-oriented languages provide polymorphism and dynamic binding for that purpose. As several methods can be grouped in object-oriented abstractions, the composition becomes more convenient if an object-oriented language is used.

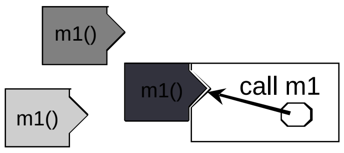

Fig. 2. Plug standardization through domain abstractions.

A framework deserves the attribute well-designed if it offers the domain-specific plugs, often referred to as variation points, to achieve the desired flexibility for adaptations. Well-designed frameworks also predefine most of the overall architecture, that is, the composition and interaction of its components. Applications built on top of a framework reuse not only source code but architecture design—which we consider as an important characteristic of a framework.

Pros and cons of frameworks

Besides the fact that reuse of architecture design amounts to a standardization of the application structure, frameworks offer further advantages. Adapting a framework to produce specific applications implies a significant reduction in the size of the source code that has to be written by the application programmer. Mature frameworks allow a reduction of up to 90% [Wei89, Fay99] compared to software written with the support of a conventional function library.

More good news is that framework-centered software development is not restricted to specific domains. Actually, frameworks are well-suited for almost any commercial and technical domain as sketched in Section 1.

The bad news is that framework development requires an extraordinary development effort. The costs to develop a framework are significantly higher (in the order of three to four times) compared to the development of a specific application: If similar applications already exist, they have to be studied carefully to come up with an appropriate generic semifinished system—the framework for the particular domain. Framework development requires a thorough domain understanding. Adaptations of the resulting framework lead to an iterative redesign. So frameworks represent a long-term investment that pays off only if similar applications are developed again and again in a given domain.

Framework development and reuse is at odds with the current project culture that tries to optimize the development of specific software solutions instead of generic ones. We refer to the discussion of organizational issues by Goldberg and Rubin [Gol95].

Finally, it takes considerable effort to learn and understand a framework. The application developer cannot use a framework without knowing the basics of the functionality and interactions the framework provides. This makes essential to not only provide a high-quality framework, but also a straightforward way to learn it. Socalled cookbooks document frameworks and support their adaptations. They contain a set of recipes that provide step-by-step guidelines for typical adaptations together with sample source code. Visual, interactive composition tools may further support the configuration process.

Software frameworks and embedded systems

Despite having proven its worth in various domains, framework technology is still largely shunned in the embedded world. To some extent, this is just one aspect of the wider problem of the technological backwardness of embedded software. Embedded software projects, when seen from the point of view of mainstream computer science, often appear to be taking place in a time warp: the language of choice remains C, the dominant architectural paradigm is only now shifting from procedural to object-based, software engineering tools are seldom used. Embedded systems additionally tend to be mass-produced which creates a financial incentive to limit their resources, and embedded software is consequently severely CPU- and memory-limited. The fact that object-oriented software requires some extra resources, is one reason why object-oriented frameworks have been avoided so far.

To penetrate the embedded world, framework technology must overcome at least this technological hurdle. This will require more powerful hardware and a greater willingness to see software as the main source of added value in embedded systems. Both conditions are gradually coming to be realized. On the one hand, the continuing decline in hardware costs is bringing more and more processing power and memory resources within the budgetary envelope of embedded software projects while, on the other hand, ever expanding consumer expectations are putting increasing demands on the software. Satisfying these demands will eventually force the adoption of the same kind of technology prevalent in other domains and frameworks will undoubtedly play an important role in shortening development times and in improving product flexibility.

As part of our commitment to demonstrating the applicability of framework technology to embedded applications, we have designed and developed a prototype software framework for satellite control systems. Satellite systems are representative candidates because on-board processor resources are undergoing a rapid expansion with the traditional 16-bits CISC processors being replaced by 32-bits SPARC processors while, on the demand side, there is growing pressure to extend the capabilities of the on-board software. We think that similar trends are in evidence in other embedded fields and we believe that software frameworks represent an effective way to exploit the extra hardware resources now becoming available to meet the new user demands.

The prototype satellite framework was developed in cooperation with the European Space Agency by an interdisciplinary team that included both satellite and software expertise. Our experience certainly confirms the generally held view that this combination of software and domain skills was essential to the success of the project. Thus, we think that the satellite framework project is representative of future framework projects for embedded systems both from a technological and from an organizational point of view.

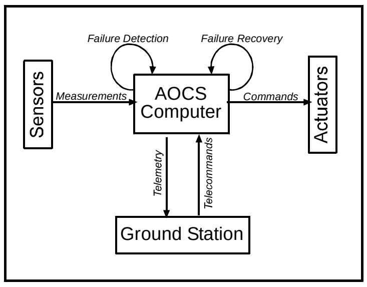

Fig. 3. General structure of an AOCS

3 Case study: a framework for satellite control systems

The Attitude and Orbit Control System (AOCS) is usually the most complex of the subsystems on a satellite. Its main function is to process the measurements of a set of attitude and orbit sensors to generate commands for a set of actuators that are designed to stabilize the satellite’s orbital position and attitude. Additionally, the AOCS must respond to commands from a ground center and must generate housekeeping data for it. Finally, like most embedded systems, the AOCS must have a high degree of autonomy and must be capable of performing failure detection tests upon itself and of carrying out emergency recovery actions in response to detected failures. The AOCS is therefore a typical embedded control system and its structure is schematically shown in figure 3.

Although the AOCS framework was initially aimed at the AOCS domain, its applicability is in fact wider because, as became clear during its development, the framework is best seen as a collection of independent modules, the so-called *functionality managers,* that can be used either together or in isolation from each other and most of which are suitable for control systems other than the AOCS.

The AOCS framework is described in detail in [Pas01b] and on the Web site [Pas01c], which includes the freely available source code. [Pas01a] provides a summary of the framework components. Below, an overview of the principles that inspired its design is given together with a discussion of the problems that are specific to the embedded character of the target domain.

Frameworks as domain-specific operating system extensions

The typical structure of an embedded system is as shown in figure 4(a). The two intermediate layers represent the reusable part of the software whereas the top layer is application-specific and is typically re-developed from scratch for each new application. The operating system is reusable because it packages functionalities that are common to most embedded systems. The device drivers are instead reusable because they are specific to the (reusable) devices whose interface they encapsulate. Both the operating system and the device drivers are constituted by components (typically available as binary entities) that are characterized by the services they offer to their clients. They are intended to be delivered as off-the-shelf items and to be configured for use in a particular application at run-time.

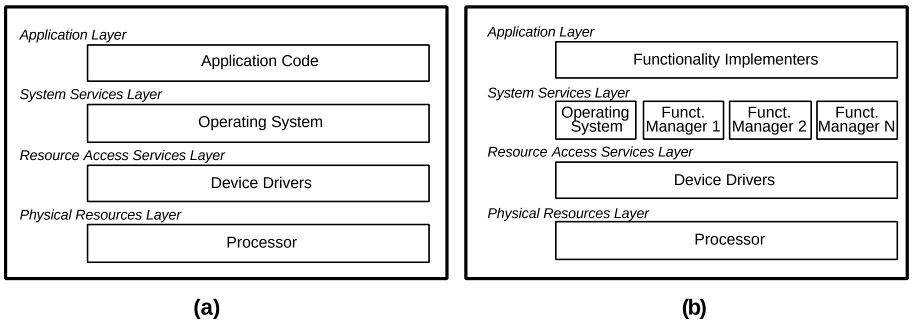

Fig. 4. Conventional (a) versus framework-based (b) structure of a control system.

The operating system represents a framework because the configuration of an operating system during application initialization can be seen as an extension of the operating system itself through the call-back style of programming. Components typically register with the operating system by passing to it pointers to functions they expose and which the operating system – which retains overall control of the main thread of execution – calls when appropriate. The scheduler, for instance, which is one of the core components of an operating system, is configured by passing to it the entry points of the tasks that it must schedule.

When looking at figure 4(a), it is natural to ask whether the operating system really represents the highest possible level of reuse in embedded systems or whether there are commonalities beyond those covered by it that can be encapsulated in reusable and reconfigurable components. If one considers embedded systems in general, the answer to this question is no. There is simply too much variation in the generic domain “embedded systems”. However, as one narrows the focus to particular categories of embedded systems, it becomes possible to identify functionalities that are common to groups of related applications and which can become the basis for domain-specific frameworks.

So far we cannot define these categories. In our project, we concentrated on satellite control systems and we tried to identify whether there were other OS-like functionalities that were specific to this domain and that might allow the OS to be extended in a domain-specific manner. In other words, the question we asked was: if we were to design an operating system *only for satellite control systems*, how would we do it? Which functions would this OS cover and which components would it offer to AOCS developers?

The OS achieves reusability by separating the *implementation* of a functionality from its *management*. Thus, for instance, the scheduler sees the tasks it schedules as abstract entities and it is exclusively responsible for deciding when they should be initialized, executed, suspended, etc. The scheduler is application-independent because it performs all these operations in a manner that is independent of how the tasks are implemented. In this sense, task management is separated from task implementation.

Analysis of the AOCS domain showed that there are several functionalities in an AOCS for which it is possible to have an analogous separation between management of the functionality and its implementation. In such cases, it then becomes possible to construct components that encapsulate the functionality management and that are completely application-independent. We have called such components the *functionality managers*. The AOCS framework offers managers for the following functionalities: telecommand and telemetry handling, failure detection, failure recovery, closed-loop controlling, as well as maneuvers and unit management.

The structure of an AOCS application instantiated from the framework then becomes as shown in figure 4(b). Comparison to figure 4(a) shows that the operating system has now been extended in a domain-specific fashion. It is worth pointing out that the functionality managers can be deployed independently of each other. There is another analogy with operating systems which are often offered as a bundle of modules of which only those needed by a particular application have to be installed.

The real-time dimension

Satellite control systems—like many other embedded systems—are subject to hard real-time constraints. Framework technology has seldom been applied to hard real-time systems. This is partly for the same reasons why it has not been applied to embedded systems in general, but also because of the special problems inherent to real-timeness. Software frameworks rely heavily on dynamic binding as a behavior adaptation mechanism and dynamic binding makes static analysis of the timing properties of an application harder to perform. Additionally, many of the classical design patterns that are commonly used to model framework adaptability are typically implemented using dynamic memory allocation which is not used in real-time applications or use recursion which is again incompatible with static timing analysis.

The position adopted in the AOCS framework project is that since embedded systems are "closed" (all the components making up an application are known at compile time), dynamic binding does not preclude timing analysis because it is always possible to put an upper bound on the execution time of a call to a dynamically bound procedure [Pas99].

As mentioned above, the AOCS framework can be seen as a collection of design patterns that have been adapted to the special needs of the AOCS domain. In many cases, this adaptation takes the form of making the pattern compatible with the real-time requirements of AOCS applications either by removing the need for dynamic memory allocation or by showing how the depth of recursion can be bounded using semantic information.

Conventional frameworks are designed to have control of application execution [Mat99]. This is a problem in the AOCS domain – and indeed in most embedded systems – where applications often differ for the scheduling policy that they adopt. The framework therefore has to be insulated from scheduling aspects. This is done by turning the functionality managers into *active components*, namely components that offer an entry point to a generic scheduler that is assumed external to the framework itself. Decoupling from the scheduling policy is achieved by designing the functionality managers in such a way that they need not make any assumptions about the frequency with which they are activated, about the source of the activation call, or about the relative ordering in which they are activated.

This decision to leave scheduling aspects outside the framework can also be justified by noting that there already exist excellent reusable solutions to the scheduling problem, provided by commercial operating systems. Hence, a framework should concentrate on addressing issues that are not yet covered, while providing an interface to the components that address the scheduling problem.

All of the solutions we have implemented in the AOCS framework to tackle the real-time dimension could, at least in principle, be carried over to other embedded domains and therefore we believe that our project demonstrates that the framework approach can be successfully applied to this class of embedded systems.

The framework and autocoding tools

Control engineers are becoming more and more accustomed to using environments such as Matlab that offers tools to design controllers and to simulate their operation. (We expect Matlab to prevail in the future. The discussion below applies equally well to similar tools, in particular XMath). Increasingly, such environments come with autocoding facilities that allow code to be generated that implements the controller defined by the user. Since the AOCS framework targets a control domain, the question naturally arises as to whether the approach it promotes to the development of an application is alternative, complementary to, or just different from, that promoted by autocoding tools [Pas99].

The main point to note in this respect is that the Matlab tool is aimed at facilitating the synthesis of control laws and that the implementation of the control algorithms usually represents a small part of the total application code. In the AOCS domain, it is probably around 20-30%. The remainder of the code covers tasks such as data handling or failure detection and recovery for which the autocoding tools do not offer any specific abstractions. Although, technically, it would be possible to generate an entire AOCS from Matlab, this approach would not foster reuse since understanding a complex Matlab model can be as difficult as understanding a complex piece of code. Reusing it is as hard as reusing poorly structured code. The AOCS framework, on the other hand, is specifically designed to be portable across projects and to have a structure that facilitates maintainability, reusability and understandability.

On the other hand, the ease of designing control algorithms in Matlab and the convenience of being able to generate code that is guaranteed to match the models that were verified by simulation is too valuable to give up. The AOCS framework therefore takes the view that the framework and the autocoding approach are complementary. The framework defines the overall application architecture but it also offers wrappers for code generated by Matlab. The intention is that Matlab covers the design and implementation of the control algorithms whereas the framework provides the architectural skeleton for the application and covers all other aspects of an AOCS application.

4 Towards a product culture—the way ahead

The AOCS framework project is only the beginning of mid- to long-term research and development activities to improve the software process for embedded control systems. The objective is that control engineers can develop new applications within a particular domain with a minimum amount of manual coding by adapting semifinished frameworks mainly through visual/interactive composition (see figure 5). Conceptually, two stages can be identified in the application development process: the definition of the *logical architecture* and the definition of the *physical architecture*. The logical architecture covers the realization of the functional aspects whereas the physical architecture covers scheduling, communication and distribution issues.

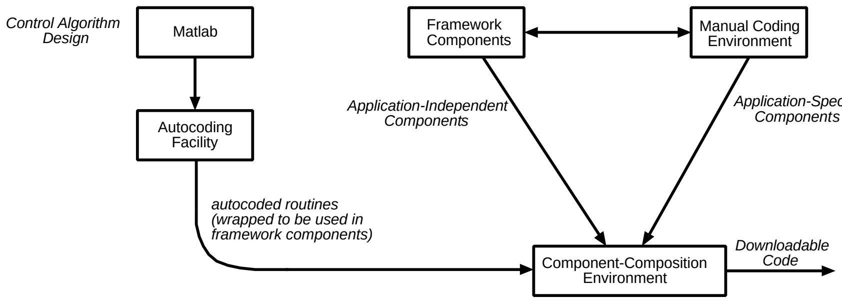
Control Algorithm Design
Matlab
Autocoding Facility
autocoded routines (wrapped to be used in framework components)
Framework Components
Manual Coding Environment
Application-Independent Components
Application-Specific Components
Component-Composition Environment
Downloadable Code

Fig. 5. Development of embedded control systems out of framework components.

The first actor in the definition process for the logical architecture is a tool such as Matlab which allows the control engineers to define and validate by simulation the control algorithms. An autocoding facility generates the corresponding code. The second actor in the definition of the logical architecture is an adequate framework providing a configurable architectural skeleton for the application and a set of default components representing common implementations of the functionalities. Inevitably, some manual development of components is required. Compliance with the framework facilitates the development of the new components and it preserves the architectural integrity and high-level uniformity of the application by ensuring that the similar problems in different parts of the application receive similar solutions.

The components then need to be configured to form a complete application. This process is typically done by manually writing the code to glue together the components, but could be automated through visual/interactive tools analogous to commonly used GUI editors.

The final step in the application development process is the imposition of a physical architecture onto the logical architecture. This means that tasks must be allocated to processors in the distributed case or to processes in the single-processor case. Furthermore, scheduling policies must be defined and the schedulability must be checked. The communication infrastructure must be put in place to allow the tasks to exchange information among themselves and with the external world. We believe that logical and physical architectures should be kept separate and should be definable independently of each other. Within the object-oriented paradigm, this is possible if a middleware is available that sustains the “illusion of local action” [Ast01] by allowing components to interact as if they resided in the same address space. The logical architecture is defined in this fictitious but uniform space and the successive definition of a physical architecture leads to its splitting into intercommunicating subspaces possibly located on different nodes in a distributed network. Distribution and communication issues should again be handled automatically by dedicated tools.

Realizing the vision of figure 5 requires significant work. Nevertheless, the main planks of the direction outlined above are already in place. In particular, Matlab is a mature commercial product with autocoding facilities. It represents a defacto standard in the embedded community that is tested and widely applied. The AOCS framework provides wrappers for the generated Matlab code. The areas where work needs to concentrate in the future are: the generalization of the AOCS framework to embedded control systems, the development of visual/interactive composition tools, and the automation of the instantiation of logical architecture to the physical architecture of an embedded control application. For each of these areas, we are trying to provide proof-of-concepts solutions that are entirely built on existing technology.

A set of frameworks for control systems

The AOCS framework is targeted at satellite control systems. Although the structure of such systems is representative of that of generic embedded control systems, to be usable in other contexts, the AOCS framework needs to be modified and extended. This can be a gradual process resulting probably in a set of frameworks for different categories of control systems. Quite likely, additional abstractions have to be added and specific components have to be created. As the AOCS framework is made up of independently deployable functionality managers its extension can occur naturally.

An interesting aspect is whether the AOCS framework imposes a certain overall architecture or favors a real-time programming paradigm. It is unclear whether a focus on the time-triggered paradigm, which is considered superior for saftey-critcial applications [Kop97] would imply a different framework design.

The analogous question has to be raised for heavily distributed embedded systems as opposed to centralized architectures. This concerns, for example, the handling of external sensors. At present, the framework assumes “dumb” external units (sensors and actuators) of the kind currently used in satellite systems. These are conceptually simple data acquisition devices with little or no internal processing capabilities. The trend is towards “intelligent” units which are processor-based and capable to interact with the central controller in complex ways. In the case of the AOCS, for instance, GPS receivers and star trackers with autonomous star pattern recognition capabilities are becoming common. They are endowed with software that rivals in complexity the software of the central AOCS computer. New concepts need to be developed to handle the interaction between a central computer and these intelligent peripherals.

A related problem concerns the definition of the boundaries of the control software. Once units start having their own software it is no longer clear whether one should consider the control software as separate from that of the external units. It may be more sensible to regard the entire control system as a distributed system and, in the spirit of figure 5, to design it as an integrated whole at the logical level and to leave its partitioning over central computer and external units to the second stage where the physical architecture of the system is defined.

Developing a visual/interactive composition tool

JavaBeans [Sun01] form the basis of component composition in Java development environments. The AOCS framework is currently being ported from C++ to Java and tested on an embedded target using a real-time version of Java. As part of the porting, all framework components are being turned into JavaBeans.

The advantage of transforming the framework into a set of beans and of modelling its instantiation as a sequence of bean operations is that it becomes possible to visually/interactively manipulate them in off-the-shelf Java development tools. For complex framework adaptations specific bean configuration editors might have to be provided.

Mapping the logical architecture to the physical architecture

The implementation of the logical architecture upon a physical platform requires ideally a description of the physical characteristics of the platform and a tool that can automatically map the logical architecture to the system at hand. An important property of such a tool should be its ability to take into account the real-time nature of most embedded control systems.

As already mentioned above, our philosophy is to reuse existing tools wherever possible and integrate them within the approach we are proposing. In this case, we found that Giotto [Hen00, Hen01] matches most of our requirements. Giotto can be seen as a middleware that offers a tool-supported design methodology for implementing embedded control systems on platforms of possibly distributed sensors, actuators, CPU's, and networks. It is specifically intended to decouple the software design (the logical functionality) from implementation concerns (scheduling, communication, and mapping to physical resources) which makes it compatible with our distinction between logical and physical architectures.

In order to be used for the skechted purpose, Giotto must be supplemented with a middleware layer to support the illusion of local action. We assess existing middleware such as DCOM or CORBA and its real-time extension as too complex and thus as unsuitable for real-time applications. Instead we propose a lean middleware concept that we call the delegate object mechanism [Bro01]. This concept is tailored to distributed real-time control system and is designed to be implemented upon a Giotto infrastructure.

We believe that the Giotto toolset combined with the delegate object mechanism automates the mapping of the functional architecture to the physical architecture, generating as an output the executable components ready to be downloaded on the embedded target.

5 Conclusions

Table 1 sketches possible means for automating the development of embedded systems. We view framework technology as key factor for coming up with reusable components for the functional aspects aside of control algorithms. A lean middleware, such as the delegate object mechanism together with Giotto could automate the mapping to a physical system and provide timing determinism. Due to the fact that the development of significant portions of embedded control system can be automated as sketched in the previous sections, we expect a thorough transformation of the embedded software market over the next decades similar to the transformation of the commercial software market through SAP since the 1970s.

Table 1. Means for automating embedded software development.

Logical System	Control Algorithms	Automation Approach
	20-30% of the logical system	reuse of models
		autocoding tools
	Management	
	70-80% of the logical system	reuse of framework components
Physical System	**Timing Determinism**	formal methods and tools
	Distribution	middleware based on formal methods and tools

References

[Ast01] M. Astley, D.C. Sturman, and G.A. Agha, *Object-based Middleware*, sidebar in Customizable Middleware for Modular Distributed Software, Communications of the ACM, Vol. 44, No. 5, May 2001.

[Bro01] T. Brown, A. Pasetti, W. Pree, T. Henzinger, C. Kirsch, *A Reusable and platform-independent framework for distributed control systems*, Proceedings of the 20-th Digital Avionics Systems Conference, Daytona Beach, FL, 14-18 October 2001

[Fay99] M. Fayad, D. Schmidt, R. Johnson, *Building Application Frameworks*, Wiley Computing Publishing, 1999

[Font01] M. Fontoura, W. Pree, B. Rumpe. The UML-F Profile for Framework Architectures. Addison-Wesley/Pearson Education, 2001.

[Gol95] A. Goldberg and K. Rubin. *Succeeding with Objects: Decision Frameworks for Project Management.* Addison-Wesley, 1995.

[Hen00] T.A. Henzinger, B. Horowitz, and C.M. Kirsch, *Giotto: A Time-triggered Language for Embedded Programming*, Technical report: UCB//CSD-00-1121, University of California, Berkeley, 2000.

[Hen01] T.A. Henzinger, B. Horowitz, and C.M. Kirsch, *Embedded Control Systems Development with Giotto*, Proceedings of LCTES 2001, ACM SIGPLAN Notices, June 2001.

[Kop97] H. Koptez *Real-Time Systems : Design Principles for Distributed Embedded Applications*, Kluwer Academic Publisher

[Mat99] M. Mattsson, J. Bosch, *Composition Problems, Causes, and Solutions,* p.467-487, in [Fay99]

[Mey89] The New Culture of Software Development: Reflections on the Practice of Object-Oriented Design; in Proceedings of Tools Europe 89, Paris, France.

[Pas99] A. Pasetti, W. Pree, *A Component Framework for Satellite On-Board Software,* Proceedings of the 18-th Digital Avionics Systems Conference, St. Louis (USA), Oct. 99

[Pas01a] A. Pasetti, *et al, An Object-Oriented Component-Based Framework for On-Board Software*, Proceedings of the Data Systems In Aerospace Conference, Nice, May 2001,

[Pas01b] A. Pasetti, *A Software Framework for Satellite Control Systems – Methodology and Development*, PhD Thesis, University of Konstanz, 2001, to be published in the Springer-Verlag LNCS monograph series.

[Pas01c] http://www.SoftwareResearch.net/AocsFrameworkProject/ProjectOverview.html

[Sun01] http://java.sun.com/products/javabeans/

[Wei89] Weinand A., Gamma E. and Marty R. Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming,* **10**(2), Springer Verlag, 1989.