A Java-Based Framework for Real-Time Control Systems

Alexander Blum, Vaclav Cechticky, Alessandro Pasetti, Walter Schaufelberger
Institut für Automatik
ETH-Zürich
Physikstrasse 3, CH-8092, Zürich
Switzerland

***Abstract* – This paper presents the Java version of the AOCS Framework. The AOCS Framework is an object-oriented software framework for real-time satellite control systems. It provides a set of design patterns, an adaptable architecture, and a set of configurable components that support the instantiation of satellite control applications. It was originally implemented in C++ but has now been ported to Java. The paper advocates the use of framework technology as the best way to promote software reuse in the control systems domain and discusses the precautions that must be taken to use this technology with Java as an implementation language in the presence of real-time constraints. It also presents two examples of instantiations of the AOCS Frameworks with two different Real-Time Java implementations.**

I. INTRODUCTION

This paper is concerned with the development of generic software architectures for real-time control systems. Such systems tend to have the same high-level structure which is illustrated in figure 1 (note that, contrary to a widespread misconception, the control system software contains much more than just implementation of control laws). Given the growing prominence of software-related costs in the total development costs of control systems, there is an obvious interest in considering whether this commonality of structure can be exploited to develop software assets that, being reusable across applications, can help contain software costs.

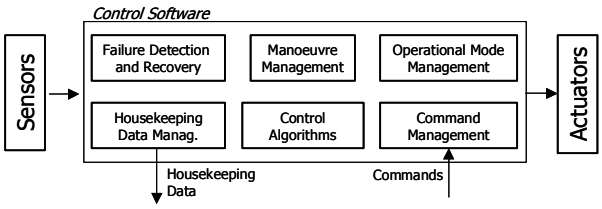

Fig. 1. Real-Time Control Systems

This paper presents an attempt to achieve this aim using framework technology. It describes one framework that we have developed for satellite control systems (the AOCS Framework). Framework technology is widely used in other fields. It has so far been shunned in the control domain primarily out of concerns for its compatibility with real-time constraints. The paper accordingly concentrates on the measures taken to ensure that the AOCS Framework is RT-compliant. The framework was initially developed in C++ but has now been ported to Java. In view of the recent interest of the control community for the Java language, the paper focuses on the latter version and in particular on two concrete implementations of the framework using two different real-time Java implementations, Jbed by Esmertec and PERC by NewMonics.

Use of framework technology is a means to contain software costs by increasing the level of software reuse. An alternative response to escalating software costs in the control community is the adoption of tools like Matlab's Real-Time Workshop that allows a control system to be designed in a GUI-based environment and to have the code implementing the design automatically generated by the tool itself. The last part of the paper describes the relationship between the Matlab and the framework approaches.

II. SOFTWARE FRAMEWORKS

Software reuse has long been practiced by control engineers but has traditionally been restricted to the *code level*: the reuse of individual routines or modules implementing recurring functions (e.g. matrix operations, common types of controllers, drivers for common units). Reuse however can take place at higher levels of abstraction. At *architectural level*, the software developer reuses a set of components defined by their external interfaces. The external interfaces of a set of components define how the components do or can interact. To reuse architecture therefore means to reuse the components and their mutual interactions. More recently, software engineers have recognized that reuse can take place at an even higher level of abstraction. *Design patterns* represent adaptable solutions to abstract design problems [3]. They encapsulate reusable design solutions and therefore allow reuse at the *abstract design level*.

Reuse can then occur at three levels: code reuse, architecture reuse, and design pattern reuse. We see software frameworks as a means to reuse software at all three levels thus maximizing the benefits of reuse [4]. More specifically, a software framework captures the commonalities of applications in a target domain and encapsulates them in three sets of constructs:

- Domain-specific design patterns
- Abstract interfaces
- Concrete components

The domain patterns offer reusable design solutions to recurring design problems in the framework domain. The abstract interfaces define the points of adaptability where the framework can be adapted to match the needs of specific

applications in its target domain. They also define the external interfaces of the framework components and hence define a reusable architecture. The concrete components implement some of the abstract interfaces and hence support the instantiation of the architecture.

Can framework technology be applied to real-time control systems? In a sense the answer is trivially positive since Real-Time Operating Systems or RTOS – a very successful instance of software reuse – are a form of framework. RTOS's are based on a small number of design patterns on how to handle concurrency. Concrete instances of RTOS's impose a specific architecture upon their client applications (they define interfaces to which these applications must conform) and they provide reusable and application-independent components that support the instantiation of that architecture. Thus, they contain all three elements listed above as characteristics of frameworks.

A RTOS however only covers a small subset of the functionalities normally implemented by a real-time control system. The question then naturally arises of whether the same framework-based approach can also be applied to the other functionalities found in these systems. Can, for instance, reusable software solutions be devised for the handling of sensors and actuators, for the implementation of control laws, for the implementation of failure detection checks and failure recovery actions, for the management of external commands, etc? The objective of our work is to show that this is indeed possible and that the benefits of an RTOS-like approach can be extended to most of the functionalities usually implemented by a real-time control system.

III. FRAMEWORKS FOR CONTROL SYSTEMS

Frameworks are primarily defined by how easily they can be adapted to match the needs of applications within their domain. Since design patterns offer encapsulation of adaptation mechanisms, we see them as the heart of a software framework. Hence, the first step in extending the framework approach to real-time control systems is the development of catalogues of design patterns for this type of systems. This requires the identification of recurring design problems in this field and the definition of abstract solutions for them. Some work has already been done in this area but it only covers concurrency and high-level architectural issues [2,5]. We have instead focused on design patterns addressing the software implementation of the following functional aspects of real-time control systems (see figure 1):

- Management of external units
- Management of commands
- Management of housekeeping data
- Management of operational mode
- Management of control algorithms
- Management of failure detection checks
- Management of failure recovery actions
- Management of manoeuvres

Our design patterns are presented in [4]. We believe that they provide a basis for the development of frameworks for real-time control systems. Two problems however arise when they are instantiated in concrete architectures and components. Firstly, the architecture and the components must allow integration with standard RTOS. Secondly, they must be compatible with real-time requirements. In order to explore these issues and to demonstrate the applicability of framework technology to real-time control systems, we have developed the *AOCS Framework*. This is an object-oriented framework for the Attitude and Orbit Control System (AOCS) of satellites.

IV. THE AOCS FRAMEWORK

The AOCS Framework is described in detail elsewhere [4]. Here its basic design is illustrated by means of an example. Consider the problem of providing a reusable solution for the management of control algorithms. An AOCS typically contains several control loops serving such diverse purposes as stabilizing the attitude of the satellite, stabilizing its orbital position, controlling the execution of slews, or managing the satellite internal angular momentum. The components implementing these control loops tend to implement the same flow of activities starting with the acquisition and filtering of measurements from sensors, continuing with the computation, and ending with the application of commands to actuators designed to counteract any deviation of the variable under control from its desired value. Despite such similarities, existing AOCS's do not have a unified management of closed-loop controllers which tends to be entrusted to a multiplicity of one-of-a-kind components that are each responsible for one single control loop.

The solution proposed by the AOCS Framework is based on the Controller Design Pattern of [4]. The pattern is illustrated in figure 2. It calls for the control algorithm to be encapsulated in a component that implements the abstract interface `Controllable`. This interface characterizes components that implement generic control algorithms (controllability in this context has thus nothing to do with the concept of controllability of control theory!). This interface defines the operations that can be performed upon a generic controller, regardless of the particular control algorithm it implements. For simplicity, only two such operations are shown. Operation `doControl` directs the component to acquire the sensor measurements, derive discrepancies with the current set-point, and compute and apply the commands for the actuators. Since closed-loop controllers can become unstable, operation `isStable` is provided to ask a controller to check its own stability (for instance, by performing heuristic checks on controller divergence).

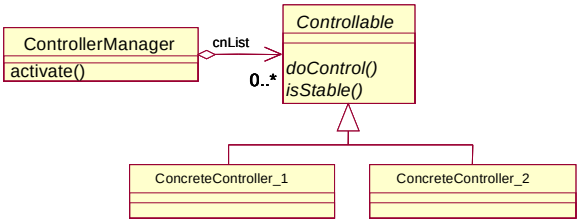

Fig. 2. The Controller Design Pattern

The introduction of the `Controllable` interface allows the definition of an application-independent `ControllerManager` component. This component maintains a list of components of type `Controllable` and is responsible for asking them to check their stability and, if stability is confirmed, to perform their allotted control action. Its key operation, `activate`, is implemented as follows:

```
Controllable* cnList[N_CN_OBJECTS];
 . . .
void activate()  {
 for (all controllers 'c' in cnList)
  if (c->isStable())
 c->doControl();
   else
 . . .     // error!
}
```

This operation would typically be called on a periodic basis by a scheduler. Its implementation is independent of the specific control algorithm and the controller manager then becomes an application-independent and hence reusable component. Reusability is a consequence of the presence of interface `Controllable`: the controller manager is application-independent because it sees concrete controllers only through this abstract interface `Controllable`. The AOCS Framework uses a similar approach to develop reusable solutions for all other functionalities implemented by a typical AOCS. In all cases, the objective is to factor out behaviour that is application-independent and to encapsulate it into reusable components. This is done by introducing an interface that separates the *management* of a functionality from its *implementation*. The management of the functionality is implemented in a reusable "functionality manager" component similar to the controller manager above. In order to separate the framework from the RTOS, the functionality managers are passive components but they expose `activate` operations that would normally be called by the RTOS.

Figure 3 shows the structure of an application instantiated from the AOCS Framework. Normally, the RTOS is the only part of a control application that is not application-specific. Now instead a set of functionality manager components have been introduced that capture the application-independent part of the application behaviour. Conceptually, it is as if the RTOS had been extended in a domain-specific manner. The AOCS Framework can thus be seen as the answer to the question: "If we had to develop an operating system only for AOCS applications, what would it look like?".

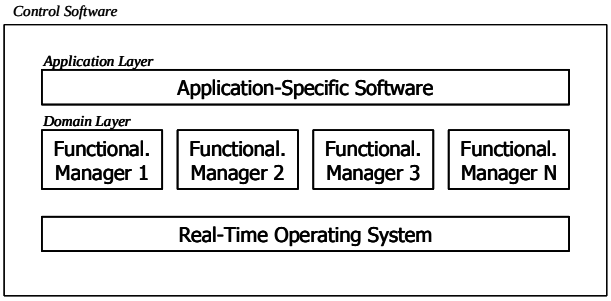

Fig.3. The Structure of AOCS Applications

V. REAL-TIME ASPECTS

Two real-time aspects of the AOCS Framework (and, indeed, of any framework for real-time control systems) must be considered. The framework must preserve the timing analysability of applications derived from it, and it must be independent of the underlying RTOS and of the scheduling policy selected by the user. The first requirement is important because control applications must normally be demonstrably schedulable and static timing analysis is essential to demonstrate schedulability. The second requirement is important because of the variety of RTOS's and scheduling policies in use in the AOCS field. Handling of real-time aspects depends on implementation language. The discussion below refers to the Java version of the AOCS Framework.

A. Timing Analysability

Timing analysability requires the execution time of any code segments to be statically predictable. This requirement has a repercussion at framework level insofar as some of the code in an AOCS application is inherited from the framework.

The AOCS framework safeguards timing analysability by avoiding constructs and operations that are not timing-analyzable. In practice, this led to the avoidance of exception handling and dynamic memory allocation. The former presented no particular problem as frameworks do not normally depend on exception handling. The latter was instead rather difficult to achieve. Most existing frameworks – and the Java language in general – are targeted at non-real-time applications and assume extensive use of dynamic object creation and destruction. Avoiding such operations required some rethinking of typical design patterns used in frameworks and sometimes resulted in slightly awkward constructs. The adopted policy is to create all objects during initialisation and never destroy them afterwards. This policy is very conservative but it is easy to enforce and guarantees that no dynamic memory allocation takes place during an application's operational phase.

Like all modern frameworks, the AOCS framework relies extensively on dynamic binding to model application adaptability. At first sight dynamic binding might seem to be incompatible with code execution predictability because of the impossibility of statically associating a definite piece of code to a particular method call. However, in an embedded control system there is no dynamic class loading and hence the number of methods that *might* be called is finite (and often small). It is therefore always possible to determine worst-case execution times and use this estimate in the schedulability analysis.

The heavy reliance of the framework on design patterns poses a more serious problem since some design patterns have a recursive structure. This makes static timing analysis impossible. In such cases, timing predictability can only be retained by adding meta-information to the source code that specifies the maximum depth of the recursion. For this purpose, the AOCS Framework documentation identifies all use of recursive patterns and explains how upper bounds on the depth of recursion can be computed.

B. RTOS and Scheduling Independence

The only interface between the AOCS Framework and the RTOS is the `activate` operation offered by the functionality managers. The functionality managers are designed not to make any assumptions about the frequency with which this operation is called. Whenever they are activated, the functionality managers take whatever action is appropriate at the time they are called without regard to how recently they were last called in the past or to how soon they may expect to be called again in the future. This insulates them – and hence the framework – from the AOCS scheduling policy. Obviously, the components that implement the AOCS functionalities and that customize the functionality managers will rely on their methods being invoked according to some timing pattern but these components are application-specific and are defined and configured by the application developer at application-level. Their dependency on a particular scheduling policy therefore does not carry over to the framework.

As an example consider the controller functionality described above. A typical concrete controller component might implement a digital filter which needs to have its internal state propagated at regular intervals. The controller component will therefore have a built-in assumption about the frequency with which some of its methods must be called. No such assumption, however, applies to the controller manager component that sees the concrete controllers through an abstract interface whose operations do not imply any timing requirements. The framework architecture therefore achieves independence from the scheduling policy of the AOCS application by confining scheduling assumptions to the application-specific components.

It should finally be noted that some scheduling policies require synchronization mechanisms to coordinate access to shared resources. The AOCS Framework takes a conservative approach and regards any public method as potentially giving access to a shared resource and therefore as in need of an access synchronization mechanism. In the Java spirit, this is provided by marking all framework methods as `synchronized`.

VI. IMPLEMENTATION ASPECTS

This paper refers to the Java version of the AOCS Framework. In recent years, many have proposed that the use of Java be extended to real-time systems, including control systems. The theoretical case for doing so is discussed in several publications, see [1] for an overview. One of our objectives was to use the AOCS Framework as a concrete opportunity to test the maturity of Java implementations for real-time applications.

The Java language is a natural choice for implementing software frameworks. It is fully object-oriented and offers a construct to represent abstract interfaces that play a crucial role in framework design. Additionally, it offers built-in multitasking and networking support and its wide user base leads to lower development costs. The presence of automatic documentation facilities is another asset of Java, which is in domains – like control systems – where often there are demanding documentation requirements.

The use of Java for a real-time control system places constraints on the application code and on the run-time system. Both must guarantee predictability of timing behaviour. The constraints on the application code were discussed in the previous section. The constraints on the run-time system chiefly relate to the enforcement of deterministic schedulable policies, garbage collection and synchronization mechanisms. Such constraints are not mandated by the Java language specifications but a number of suppliers provide Java run-times that claim to be RT-compliant. Among them there are Jbed by Esmertec and PERC by NewMonics which are the system we used in the tests described below.

Jbed by Esmertec [14] offers a Java Virtual Machine (JVM) running directly on the hardware that implements a subset of the Java 1.2 libraries and adds some real-time guarantees. The most important are: a priority-based scheduling of Java threads, priority inheritance mechanism to avoid unbounded priority inversion, and an incremental and preemptable garbage collector running on a low priority background task. PERC by NewMonics [13] runs on top of a conventional RTOS. It consists of a set of development tools and a Java runtime platform for real-time software development. The latter implements a subset of the Java 1.3 libraries. It provides round robin scheduling with priority inheritance for tasks with the same priority. PERC has a preemptable garbage collector running as a low priority background task that can be optionally switched off. PERC is generally considered to be the most mature real-time Java implementation currently on the market.

We tested the framework approach by using the AOCS Framework to instantiate a controller for a “Swing Mass Model” or SMM (see figure 4). This is a laboratory equipment consisting of two rotating disks connected to

each other and to a load with a torsional spring. A DC motor drives the one of the disks. The goal of the controller is to control the speed and position of the other disk. The instantiated application includes: two operational modes, failure detection checks on the main system variables, autonomous failure recovery actions autonomously executed upon detection of failures, provision of housekeeping data, processing of external commands, capability to execute speed profiles. It is therefore representative of a full control system application as conceptualised in figure 1.

Fig. 4. The Swing-Mass Laboratory Model

The application was first run on the Jbed Java run-time and then on the PERC Java run-time. Different real-time architectures were used for each case. In the Jbed case, the control period was 1 second and four threads were used of which two were real-time. The first real-time thread collects and processes sensors measurement, performs failure detection and failure recovery actions, calculates control laws. The second real-time thread sends data to actuators. The two remaining non real-time threads maintain a communication between the prototype framework application and a ground station by sending telemetry data and receiving telecommands. In the PERC case, the control period was 0.2s and five threads were used of which three were real-time. Essentially the first real-time thread implemented for Jbed platform was split into two real-time tasks, thus the first real-time only collects and process sensors measurement, and the second real-time thread performs failure detection and failure recovery actions, calculates control laws. The hardware platforms were also different: a PowerPC 823@66Mhz processor in the Jbed case and a Pentium3@1GHz in the PERC case. It is noteworthy that, despite these differences, the framework was instantiated in the same manner in both cases which demonstrates the claim made in the previous section about its independence from the RTOS and scheduling policy.

PERC allows easy integration of Java code with C code and we used C routines for implementing the low level drivers and for implementing the controller. In the Jbed case, the entire implementation was done in Java.

Use of Java is often resisted on grounds of excessive CPU and memory overheads. Our experience is that CPU-related concerns are exaggerated. Even on the slow PowerPC processor, total execution time for the real-time part of the application was only 27 ms per cycle. On the Pentium processor it was around 1 ms. These very good performances are primarily due to the fact both Jbed and PERC use ahead-of-time compilation which brings Java performance close to of that of compiled languages. The execution times where computed by taking the worst value in a large series of measurements. The measurements for both real-time Java platforms are summarized in table 1.

Tasks	*Jbed*	*PERC*
Real-time task 1	10 – 26 ms	71.5 us
Real-time task 2		333.4 us
Real-time task 3	1 ms	26.8 us
Telemetry thread	10 – 17 ms	800 us
Telecommand thread	1 ms	50 us

Table 1. The application timing requirements

Memory footprint might instead be a concern. In the Jbed case, total memory occupation was 522 KBytes (data+code) of which less than 200 KBytes were for the Java run-time system. In the PERC case, the application required 1.5 MBytes but the Java run-time required 9MBytes. Note that the first implementation of the AOCS framework was in C++. A prototype application was developed for ERC32 SPARC processor running at 14MHz and using RTEMS operating system. A size of the AOCS prototype application was around 500Kbytes, which is comparable to the Java prototype applications running either on Jbed platform [6].

Perhaps more worryingly, we discovered that the Jbed implementation of periodic threads is incorrect in that it introduces a drift that is equivalent to a bias error on the period. Jbed provides an atypical solution in comparison with other real-time Java Virtual Machines; runs directly on the hardware without underlying real-time operating system, hence Jbed has an integrated scheduler that is responsible for the management of real-time tasks and non real-time threads. Jbed's scheduler is being reset every scheduling period to reschedule dynamically tasks execution. However this operation adds certain computation overhead that results in the period drift.

As discussed in the next section, we believe that the framework approach should be combined with the Matlab autocoding approach. We demonstrated the feasibility of doing so by using Matlab to synthesize the controller for the SMM. The code implementing the control algorithm was then generated from the real-time workshop. The AOCS Framework offers a wrapper for Matlab-generated autocoded routine and this was used to integrate the control algorithm in the application instantiated from the framework. However, Matlab can only generate C code and integration with the Java code of the framework requires an efficient implementation Java's native interface which only seems to be available on the PERC platform.

VII. DISCUSSION

Four "lessons learnt" emerge from the work described here. First, framework technology can and should be used in the control domain. Its use in a real-time context requires some deviations from established practice but is clearly possible. It is also beneficial because real-time control systems exhibit a fairly stable structure that can be captured by a framework. Our experience is that about half the classes in a final application can be provided by the framework. The development of the remaining classes is facilitated by the need to adhere to the framework design patterns and abstract interfaces.

Second, Java is becoming a realistic option for real-time control systems but caution is required in selecting the run-time system. Of the two systems we tested, only one (PERC) proved satisfactory. Note that recently have appeared on the market two new Java Virtual Maschine implementations, JTime by TimeSys [10] and Aero JVM developed by a consortium lead by Astrium SaS [11]. Both are compliant with Real-Time Specification for Java produced by The Real-Time for Java Expert Group [15]. Presumably provide a better solution than Jbed and PERC, such as a real-time garbage collector, a full control over real-time and normal threads, and direct access to any memory areas, etc.

Third, the chief advantage of using Java for control system programming is that the presence of the Java Virtual Machine (JVM) blurs the distinction between the desktop and the embedded environment. One reason why control software development is hard is that much of it must be done on not-very-friendly embedded targets. When Java is used, however, the JVM insulates the application from the underlying platform and from the RTOS. This allows nearly all of the development to be done on a convenient desktop environment in the knowledge that behaviour – including much scheduling-related behaviour – is the same as on the remote target. Our experience certainly confirms that Java's much vaunted Write-Once-Run-Anywhere claim holds for control systems, too, and that it can yield the same benefits in this domain as it does in others.

Finally, the relationship of the framework-based approach advocated here to autocoding tools like Matlab's Real-Time Workshop must be considered. Both approaches aim to contain software costs. The Matlab approach is more ambitious because it tries to avoid manual intervention in the software generation process. However, Matlab can only model a fraction of a control system. It excels at modelling control algorithms but the software of a control system is dominated by functions like unit management, command processing, housekeeping data generation, failure detection and failure recovery, and other functions for which Matlab provides no specific abstractions and which it is consequently unable to model effectively. In our domain of interest (satellite control systems), for instance, the implementation of control algorithms typically takes only about 20-30% of the total software.

A framework is ideally suited to modelling the non-algorithmic part of the control software and for this reason we see it as complementary to a Matlab-based approach. Indeed, the AOCS Framework offers a component that can act as a wrapper for Matlab-generated code thus allowing to combine the advantages of both approaches.

Our long-term vision of how control software should be developed is sketched in figure 5. The final application is obtained by configuring and assembling components. Some of these components are provided by a framework, some are wrappers for Matlab-generated code, and some are developed manually (but must comply with the framework design patterns and abstract interfaces). The work described in this paper concentrated on the framework element of this vision (lightly shaded in the figure). On-going work instead aims at building a GUI-based composition environment where the control engineer can configure and assemble a full application exclusively through graphical means with no need to write any code. This type of environment will eventually extend the Matlab autocoding approach to the entire software of a control system.

The development of the AOCS Framework was supported by the European Space Agency under contract 13776/99/NL/MV[1]. The AOCS Framework is publicly available from the project web site at: control.ee.ethz.ch/~pasetti/RealTimeJavaFramework. The on-going work on extending its approach as shown in figure 5 is documented at: control.ee.ethz.ch/~pasetti/Automated FrameworkInstantiation.

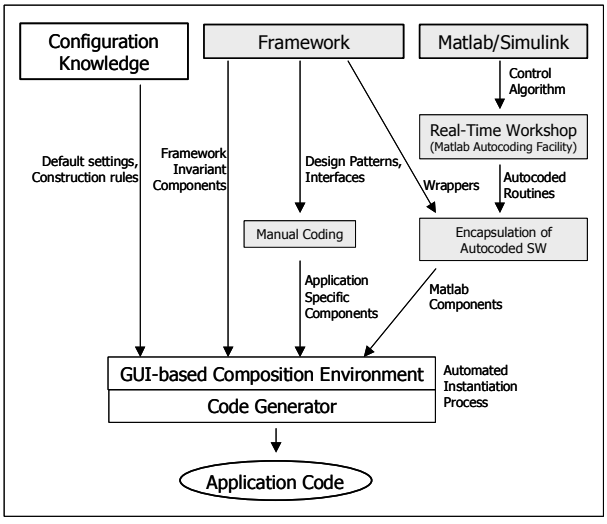

Fig. 5. The Automated Instantiation Environment

VIII. CONCLUSIONS

The work presented in this paper addressed two issues. The first was the definition of a set of design patterns specifically aimed at embedded control systems and the

[1] The views expressed in this paper are those of its authors only. They do not in any way commit the European Space Agency or reflect official European Space Agency thinking.

development of a software framework instantiating them. The second aspect was the use of the Java language as implementation technology for the framework. The use of software frameworks has been investigated for nearly a decade [2,3,5,7] and it is certainly our experience that software frameworks can bring to embedded control systems the same benefits that they have brought to other disciplines. Similar results have also been reported in domains close to the control domain. Two examples are a flexible software framework for real-time motion robot control OROCOS [8] and a software framework for robot motion planning, coordinate transformation and control which is also implemented in Java [9]. On the language issue, our starting point was the realization that there are currently several well-established software solutions for Java for real-time systems. Prominent examples are like PERC, JamaicaVM, JTime, and the Aero JVM [10,11,12,13]. Of these, at least two (JTime, Aero JVM) are compliant with the specifications for a real-time version of Java backed by Sun Microsystems [15]. Our experience is that some at least of these solutions are mature for embedded control systems provided that certain precautions are taken at implementation level.

IX. REFERENCES

[1] B. Brosgol and B. Dobbing, "Can Java Meet Its Real-Time Deadlines?", *Ada-Europe*, LNCS 2043, 2001, pp. 68-87.

[2] B.P. Douglass, *Real-Time Design Patterns*, Addison-Wesley, Massachusetts; 2002.

[3] E. Gamma and *et al.*, *Design Patterns – Elements of Reusable Object Oriented Software*, Addison-Wesley, Massachusetts; 1995.

[4] A. Pasetti, *Software Frameworks and Embedded Control Systems*, LNCS Vol. 2231, Springer-Verlag; 2002.

[5] J. Zalewski, "Real-Time Software Architectures and Design Patterns: Fundamental Concepts and Their Consequences", *Annual Reviews in Control*, 25, 2001, pp. 133-146.

[6] A. Pasetti and *et al.*, "An Object-Oriented Component-Based Framework for On-Board Software", *in Proceedings of the 2001 DAta Systems In Aerospace Conference*, http://www.eurospace.org/

[7] M. Fayad, R. Johnson, and D.C. Schmidt, *Implementing Application Frameworks: Object-Oriented Frameworks at Work*, Wiley & Sons, New York, 1999.

[8] H. Bruyninckx and P. Soetens, "Generic real-time infrastructure for signal acquisition, generation and processing", *Fourth Real-Time Linux Workshop*, December 2002, Boston.

[9] M. Honegger, "A Java-Based Framework for the Designof Robot Controllers", *in Proceedings of the 2002 Embedded Systems in Mechatronics Conference*,October 2002, Winterthur, Switzerland, pp.59-65

[10] JTime, http://www.timesys.com/

[11] F. de Bruin and *et al.*, "A Standard Java Virtual Machine for Real-Time Embedded Systems", *in Proceedings of the 2003 Data Systems In Aerospace Conference*, June 2003, Prague, Czech Republic. http://www.aero-project.org/

[12] JamaicaVM, http://www.aicas.com/

[13] PERC, http://www.newmonics.com/

[14] Jbed, http://www.esmertec.com/

[15] Real-Time Java Expert Group, "The Real-Time Specification for Java", http://www.rtj.org/