Software Component Services for Embedded Real-Time Systems

Frank Lüders and Daniel Flemström
Dept. of Computer Science and Electronics
Mälardalen University
Box 883, SE-721 23 Västerås, Sweden
{frank.luders|daniel.flemstrom}@mdh.se

Anders Wall
ABB Corporate Research
Forskargränd 8
SE-721 78 Västerås, Sweden
anders.wall@se.abb.com

ABSTRACT
The use of software component models has become popular during the last decade, in particular in the development of software for desktop applications and distributed information systems. Such models have not been as popular in the domain of embedded real-time systems, presumably because of the special requirements such systems have to meet. There is a considerable amount of research on component models for embedded real-time systems, or even narrower application domains, which focuses on source code components and statically configured systems. This paper explores an alternative approach by laying the groundwork for a component model based on binary components and targeting the broader domain of embedded real-time systems. The work is inspired by component models for the desktop and information systems domain in the sense that a basic component model is extended with a set of services for the targeted application domain.

1. INTRODUCTION
The use of software component models has become increasingly popular during the last decade. The most widespread use of such models has occurred in the development of software for desktop applications and distributed information systems. The most popular component models include JavaBeans [6] and ActiveX [5] for desktop applications and Enterprise JavaBeans [15] and COM+ [18] for distributed information systems. In addition to basic standards for naming, interfacing, binding, etc., these models also define standardized sets of run-time services oriented towards the application domains they target.

Unlike for desktop applications and distributed information systems, there has been no widespread use of software component models in the domain of real-time and embedded systems. It is generally assumed that this is due to the special requirements such systems have to meet, in particular with respect to timing predictability and limited use of resources such as memory and CPU time. Much research has therefore been directed towards defining new component models for real-time and embedded systems, typically focusing on relatively small and statically configured systems. Most of the published research proposes models based on source code components. Typically, these models target relatively narrow application domains. Examples of published models include the Koala component model for consumer electronics [24], PECOS for industrial field devices [8], and SaveCCM for vehicle control systems [9].

The focus on statically configured systems of source code components is motivated by efficiency as well as by the possibility of ensuring predictable behavior through source code analysis and white-box testing. A potential liability of using source code components is that application developers rely on component properties that may be inferred from the source code but are not guaranteed by component specifications. Thus, a system may break if a component is updated with a new version that does not have the same inferred properties, although the component specifications are compatible. The restriction to static configuration is increasingly at odds with requirements for flexibility, adaptiveness, etc. The development of component models for relatively narrow application domains is motivated by the desire to optimize systems for attributes considered particularly important for those domains. Typically, such narrow models, as well as supporting tools and run-time infrastructures, have to be developed by the application developing organizations themselves.

An alternative approach is to strive for a component model for embedded real-time systems based on binary components and targeting a broader domain of applications, similarly to the domain targeted by a typical real-time operating system. Such a model could suitably be provided by platform vendors, as is the norm for component models used for desktop applications and information systems. Although any model based on binary components is likely to incur some overhead, efficient use of resources should be a primary concern in the design of such a model for embedded real-time systems. When it comes to ensuring predictable behavior, our vision is that analysis of systems should be based on specifications (i.e. models) of the included components rather than relying on access to source code. Realizing this vision requires methods for ensuring that components comply with specifications as well as for predicting the properties of a system based on properties of its constituent components [11].

Further discussion of such methods is outside the scope of this paper, however.

One possibility is to use a mainstream component model, such as Microsoft's Component Object Model (COM) [4], as the starting point for developing a component model for embedded real-time system. Previous work has demonstrated that the key concepts of COM can be used with advantage in the development of an embedded real-time system [13]. A study of COM and its extension Distributed COM (DCOM) [20] shows that these models are not inherently incompatible with real-time requirements [12]. Some reasons that COM is an attractive starting point are that the model is relatively simple and that commercial COM implementations are already available for a few real-time operating systems. Another advantage of COM is that it is already well-known and accepted in industry. It has been reported by Jean Favre *et al.* that industry is often reluctant to use the complex and domain specific solutions that academia provides [7]. One might worry that a model based on COM would be viewed as old-fashioned from the start, since COM is being increasingly replaced by Microsoft's newer .NET model [19] on the company's desktop and server platforms. However, Lutz and Laplante have found that .NET does not provide the timing predictability required for real-time systems [14].

The purpose of this paper is to lay the groundwork for a software component model for embedded real-time systems, using the basic concepts of COM as the starting point. Our vision is to make component-based software development an attractive option for embedded real-time systems by extending the basic model with standardized services of general use for this application domain, much like COM+ extends COM with services for distributed information systems. Section 2 describes the general characteristics of embedded real-time systems and outlines the requirements for a software component model targeting this class of systems. In Section 3 we clarify what we mean by component services and identify some useful services for embedded real-time systems. Related work is reviewed in section 4 with particular focus on how the problems addressed by the component services proposed in Section 3 is solved by existing technologies. Section 5 is an outline of a prototype tool we are developing to support the component services presented in this paper. Conclusions and some ideas for further work are presented in Section 6.

2. SOFTWARE COMPONENTS AND REAL-TIME SYSTEMS

2.1 Characteristics of Real-Time Systems

A real-time system is a system where the correctness not only depends on correct functionality, but also on the timing of delivered functionality. Consequently, the correct results should be delivered not too early, nor too late. Real-time systems are often embedded. Hence, resources such as computational bandwidth and memory are scarce. Typically, we find real-time systems in safety-critical applications, e.g. in the automotive and aerospace domains. Therefore, it is desired that their behavior, functional as well as temporal, is deterministic and ultimately predictable. In order to be predictable the system configurations has to be fixed, or at least fixed within an upper bound budget in terms of resource utilization. This is, however, not necessarily true in the class of real-time systems that we are targeting, where systems can be dynamically re-configured and in presence of third-party developed components whose temporal behavior, i.e. execution times, are not explicitly specified.

Temporal requirements may take many different shapes. In general, it is the physical environment that imposes the timing constraints on the system. For instance, the required sampling frequency of a control system is determined by the dynamics of the process to be controlled. A typical example of a temporal requirement is deadlines. Deadlines specify the longest acceptable response time that a particular function may exhibit. Existing methods for analyzing a real-time system in order to determine whether or not the temporal requirements are fulfilled require information about the software architecture, e.g. scheduling policy, and the temporal behavior of the different services, e.g. period times, scheduling priorities, and execution times, in the system.

Moreover, in order to be predictable, the components must exhibit a deterministic behavior. Typically, we are interested in the worst-case scenarios, e.g. worst-case execution time (WCET), to be able to predict response times. Consequently, the software architecture in a real-time system must play along the rules governed by the method used for predicting its behavior. This is especially true for the components in the system. They must provide their services in a time deterministic manner.

The thread of execution, often referred to as a *task*, defines a path of execution in the application. These can execute in parallel and are typically executed periodically or triggered by events. Depending on the scheduling strategy, the priority on a task determines the execution order in the schedule.

2.2 Component Model Terminology

A component technology typically consists of a component model and a component model implementation. In this paper we use the following definitions of these concepts from [10]:

> A *component model* defines specific interaction and composition standards. A *component model implementation* is the dedicated set of executable software elements required to support the execution of components that conform to the model.

Examples of quality attributes used in the real-time domain are WCET and resource usage. A component framework is a set of services needed for realizing a specific component model. Examples of such are the COM-libraries, .NET Runtime and different EJB implementations. Typical services of the framework is to handle run-time creation of components and component intercommunication.

There are several definitions of what a software component is. In academia a common definition is the one by Szyperski [23]:

> A software component is a unit of composition with contractually specified interfaces an explicit

context dependencies only. A software component can be deployed independently and is subject to composition by third party.

COM specifies a component as "A piece of compiled software, which is offering a service". This is a very general definition, but in practice and specifically for this paper together with the definition above, this means that a component corresponds to a file containing binary executable code, often a DLL.

2.3 Real-Time Component Technologies

Real-Time component technologies are component technologies that enforce temporal and resource determinism. They are often less flexible than their desktop counterparts in order to save resources and increase predictability. The requirements we put on a real-time component technology, in our setting, include the following:

- Enforcing thread of execution.
- Deterministic execution times on services provided by components.
- Time deterministic communication among components.
- Synchronization of access to common physical and logical resources.

Having a deterministic execution time behavior requires that the communication between components is time deterministic as well. Consequently, components must be able to guarantee an temporal upper limit on the communication. Finally, the access to components must be synchronized in a way such that their states are kept consistent. For instance, two concurrent services may not simultaneously update state variables in a component.

3. COMPONENT SERVICES

In this paper we define Component Services as solutions to common problems that can be applied to components with no or little programming effort. Examples of such services are logging, synchronization and transaction control. The implementation of the services may reside in a component framework, be generated from component specifications or a combination of both. In the *time determinism* area, we striving for reduced pessimism in the WCET budgets of third party components. In a complex industrial setting it may not be possible to exactly determine the WCET for each method. One way of solving this problem is to use time budgets [16]. These budgets specify the upper limit of the estimated WCET. The rest of the system is assembled accordingly.

For *synchronization* we concentrate on the case where several threads of execution needs access to the same data, typically the state of a component. Here we apply different synchronization policies in order to protect shared data from being written from several sources at the same time. This means that singlethreaded component may be used in a multithreaded environment without modification. COM supports a rudimentary protection where only one thread at a time can access a component. We strive for a more flexible synchronization and with less overhead than in the case of COM . Since we are dealing with real-time applications, it should be possible to specify how long it is acceptable to wait before the call is dispatched. If many calls are blocked, the total response time might exceed the deadline even though the WCET of the component is less than the deadline.

Traditionally, most of these mechanisms have to be hand coded and off line deduced using complex theories, such as petri nets [17] which can be very time and resource consuming and in some time impossible due to the complexity of a real industrial setting. If third party binary comports are used, it may also be impossible to add for example, synchronization or execution path logging.

4. RELATED WORK

The services discussed in this paper have already been adopted by some current and emerging technologies. As a base for our discussions, we have selected a few of the most common solutions for the issues stressed in Section 2. In addition, this section briefly reviews some existing research on binary components for real-time systems.

Microsofts component model COM [4] is originally targeting the desktop software domain, which means that it has a good support for specifying and maintaining functional aspects of components while disregarding temporal behavior and resource utilization. Often this can only be overcome with a substantial amount of component specific programming. There is no built in support to automatically measure and record execution times for methods in components. This is typically done by third party applications that instrument the code in run-time. These applications are typically not well suited for executing on embedded resource constrained systems. The desktop version of COM, as well as the DCOM package available for WindowsCE, has some support for synchronizing calls to components that are not inherently thread safe. This is achieved through the use of so-called *apartments*, which can be used to ensure that only one thread can execute code in the apartment at a time. Since this technique origins from the desktop version of COM, there is no built in support for time determinism and the resource overhead is larger than desired for many embedded systems.

COM+ [18] is Microsoft's extension of their own COM model with services for distributed information systems. These services provide functionality such as transaction handling and persistent data management, which is common for applications in this domain and which is often time consuming and error prone to implement for each component. Application builders declare which services are required for each component and the COM+ run-time system provides the services by intercepting calls between components. COM+ is a major source of inspiration for our work in two different ways. Firstly, we use the same criteria for selecting which services our component model should standardize, namely that they should provide non-trivial functionality that is commonly required in the targeted application domain. Since our component model targets a different domain from COM+, the services we have selected are different from those of COM+

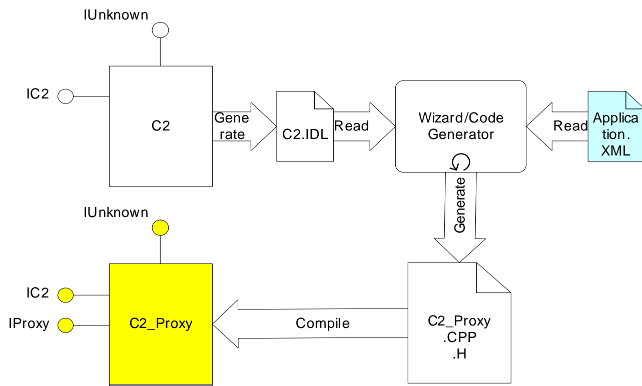

Figure 1: Generating a Component Service.

as well. Secondly, we are inspired by the technique of providing services by interception. Section 5 demonstrates how this can be achieved using automatically generated proxy objects.

Wolfgang *et al.* [21] describes a method, using C# [1] attributes to generate a proxy that handles component replication for fault tolerance. Our work is primarily targeting COM and C++, which does not support attributes as used in that paper. However, there is a possibility to use custom IDL attributes instead. As opposed to C# attributes, these attributes can only be read in compile time, which means that we need a compilation step just before deploying the component.

Zerzelidis and Wellings [25] describe their requirements for adding real time capabilities to the .NET framework. They have approached the problem from the Java Real-Time Specification [3]. This report aims to adjust the CLR (Common Language Run-time) for real-time support.

Sundmark *et al.* [22] describes monitoring of components in order to gain more realistic WCET estimations. In their model the WCET is guessed at development time and the component is then continuously monitored at runtime and measurements of execution times are accumulated. This is specifically useful when moving components between different hardware and software platforms where executions times will obviously vary. This technique is very similar to our approach.

5. PROTOTYPE OUTLINE

In this section we outline our intentions for a prototype that adds support for well known state-of-the-art solutions of common real-time problems to the widely accepted COM technology. We suggest that COM is extended with wizard generated light weight component services together with a resource efficient service framework. Attributes that only applies to a specific component, and does not depend on the application is specified in the IDL file of that component. Information that is application dependent is stored in the application configuration file. These files are used by the wizard when realizing the component services. Figure 1 shows this process in a schematic way.

Staying as close as possible to the original COM and COM+ concepts is considered to be an important design goal. Another design goal is that the programmer or integrator should be able to choose what ever services is needed for each specific component without having to change the implementation or doing any programming. There are however cases, for example when we add timeouts on actions or converting synchronous interfaces into asynchronous ones, where there is a need for adapting the code of the client component to fully benefit from the service.

Specific to COM is that a component is realized by a set of co-classes that in turn implements a number of interfaces. All interfaces have a method `QueryInterface` that allows us to change from one interface to another on the same co-class, given that we know the interface ID. In this prototype we apply services on interfaces of a component. In practice this means we have know the co-class of the component and generate a proxy code for all interfaces of that co-class. This is required by the definition of `QueryInterface`. A proxy is a COM co-class that acts another co-class. Incoming calls are processed and then delegated to the real implementation. Proxies are heavily used in COM, DCOM and COM+ e.g for representing components on another node or for adding transactional support. This concept makes component services more or less transparent to the component client and server. This is an attractive alternative as it gives the opportunity to add and remove functionality at deployment-time. Although a proxy adds some probe effects, that effect should be deterministic and constant.

We suggest that a deployment host (typically a desktop PC with a development environment) is used where wizards and compilers can generate and build the necessary proxies and interconnections. This allows a high level of flexibility when assembling and deploying a system, while runtime flexibility is kept on a reasonable level. Information not directly required in run-time is removed from the components when they are deployed on the target system.

5.1 Logging

The logging component service allows us to exactly determine the execution path and the timing of an application without recompiling or in any other way programmatically changing the components involved. The chosen solution is combine proxies with a simple time stamped logger function. To add logging for an interface, we simply add an entry in the application configuration file. Figure 2 shows an example of such a file, written in XML. In the figure, C2 implements the requested interface IC2. For this interface we wish to apply a logging service. A proxy that handles the interfaces of the co-class C2 is generated and deployed together with the original components.

At run-time, when a client tries to create a `C2` co-class, it gets a proxy that writes a time-stamped log message to a log-file,before and after each call delegation. Figure 3 is a schematic of the involved objects.

5.2 Execution Time Measurement

This service monitors a component with respect of execution time. Different cases like for example worst, best and average case can be monitored. The information can be used by

```
<application>
 ...
<component name="myProject.C2">
  <interface name="IC2">
    <service type ="Logging"/>
  </interface>
</component>
 ...
</application>
```

Figure 2: Specification of a Logging Service.

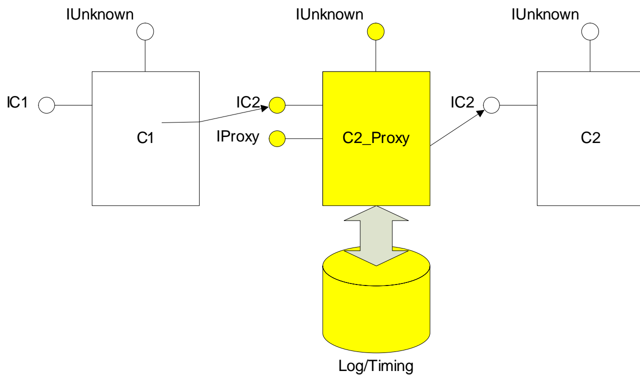

Figure 3: A Logging Service Proxy.

an on-line scheduler to adapt its scheduling strategy. The chosen solution here is a proxy with a simple time collection function. A timer is started before delegating the call, and stopped immanently after the call has returned.

Adding time measurement for an interface is done similar to the previous example. We simply add an entry for the affected co-class in the application configuration file as illustrated in Figure 4. No programming is required in the client nor in the server component.

5.3 Time Deterministic Synchronization

Using a synchronization policy together with a priority policy, this service synchronizes calls from several threads to a non-thread safe component in a time deterministic manner.

```
<application>
 ...
<component name="myProject.C2"
  <interface name="IC2">
    <service type="Timing">
      <measurement type="mean" />
      <measurement type="worst"/>
    </service>
  </interface>
</component>
 ...
</application>
```

Figure 4: Specification of a Timing Service.

Several different policies may be useful and will be described further in this section.

Our solution here is to use proxies that encapsulate synchronization mechanisms. Such a proxy can be applied to an interface, a co-class, or a complete component, depending on how the component is built up internally. For example; interfaces of a co-class that does not write to the same data may be served by different proxies to avoid unnecessary blocking.

Time monitoring is used for calculating if the calls will meet their deadlines. The worst case response time for a function call is calculated as the WCET sum of all calls waiting in the proxy and compared to the deadline. If the deadline can be met, the call is put into the queue of the synchronization mechanism. If, on the other hand, the deadline cannot be met, an error code is immediately returned. A synchronization policy can be specified for an interface, co-class, or component, in the application configuration file.

A *mutual exclusion* policy blocks all threads in the queue except one. The calls are then dispatched one by one according to the priority policy. When a call has completed, the calling thread unblocks and returns to the client. Thus guaranteeing that exactly one thread executes code within the component and that functions are always allowed to complete before switching thread. Using the synchronization service with this policy requires no programming in the server component, but the client side should be able to handle error codes that indicate that deadlines cannot be met.

Another class of synchronization policies is different *reader/ writer* policies. These differs from the previously described policy in that any number of read operations may execute in parallel, while write operations have exclusive execution. Thus, the operations subjected to a reader/writer policy must be classified as either writer or reader operations, depending on whether they may modify state or not. Concurrent "read"-threads are scheduled by the OS with respect of their priorities. The locking time for those threads is reduced as well. Using this policy means that the IDL file for the component must be augmented with custom properties for each method, telling whether it is a read or write type of operation. If a method does not have such an attribute, the proxy must assume it may write data. No programming is required in the server component, but the client side should be able to handle error codes as with the mutual exclusion policy.

For all synchronization policies, we may select if the priority of the dispatching thread should be the same as the calling thread, or as specified in the application XML file.

5.4 Vertical Services

In addition to the type of services discussed above, which we believe are generally useful for embedded real-time systems, one can imagine many services aimed at more specific application domains, often called *vertical services* [10]. Among the services we have considered are cyclic execution, which are much used in process control loops [2], and support for redundancy mechanisms such as N-version components, which are useful in fault-tolerant systems [21].

6. SUMMARY AND CONCLUSION

The aim of this work has been to lay the groundwork for component services for embedded real-time systems using COM as a base technology. A major benefit of this approach is that industrial programmers can leverage their knowledge of existing technologies. Also, extending COM with real-time services probably requires less effort than inventing a new component technology from the ground. The initial experiences with the prototype shows that it is possible to create tools and wizards that more or less invisibly add real-time services to a standard component model.

We realize that the proposed solutions imposes some time and memory overhead, and we believe that this is an acceptable price for many embedded real-time systems if using the model reduces the software development effort. It is, however, necessary that this overhead can be kept within known limits. In our future work, we plan to evaluate a prototype implementations of the proposed component model experimentally. Measurements will be made to determine the component model's effect on timing predictability as well as time and memory overhead.

We have been able to identify some component services which we believe are useful for embedded real-time systems. As part of our future work, we plan to evaluate the usefulness of the services as well as to extend the set of services. We hope to do this with the help of input from organizations developing products in such domains as industrial automation, telecommunication, and vehicle control systems.

7. REFERENCES

[1] T. Archer. *Inside C#*. Microsoft Press, 2001.

[2] K. J. Åström and B. Wittenmark. *Computer Controlled Systems — Theory and Design*. Prentice Hall, 2nd edition, 1990.

[3] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison Wesley Longman, 2000.

[4] D. Box. *Essential COM*. Addison-Wesley, 1997.

[5] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[6] R. Englander. *Developing Java Beans*. O'Reilly, 1997.

[7] J. Estublier, J.-M. Favre, and R. Sanlaville. Tool adoption issues in a very large software company. In *Proceedings of the 3rd International Workshop on Adoption Centric Software Engineering*, 2003.

[8] T. Genler, C. Stich, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arvalo, B. Schönhage, and P. Müller. Components for embedded software — the PECOS approach. In *Proceedings of the 2002 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2002.

[9] H. Hansson, M. Åkerholm, I. Crnkovic, and M. Törngren. SaveCCM — a component model for safety-critical real-time systems. In *Proceedings of the 30th EROMICRO Conference*, 2004.

[10] G. T. Heineman and W. T. Council. *Component-Based Software Engineering — Putting the Pieces Together*. Addison-Wesley, 2001.

[11] S. A. Hissam, G. A. Moreno, J. Stafford, and K. C. Wallnau. Enabling predictable assembly. *The Journal of Systems and Software*, 65(3):185–198, 2003.

[12] F. Lüders. Adopting a software component model in real-time systems development. In *Proceedings of the 28th Annual IEEE/NASA Software Engineering Workshop*, 2004.

[13] F. Lüders, I. Crnkovic, and P. Runeson. Adopting a component-based software architecture for an industrial control system — a case study. In C. Atkinson, C. Bunse, H.-G. Gross, and C. Peper, editors, *Component-Based Software Engineering for Embedded Systems*. Springer Verlag, 2005.

[14] M. H. Lutz and P. A. Laplante. C# and the .NET framework: Ready for real-time? *IEEE Software*, 20(1):74–80, 2003.

[15] R. Monson-Haefel, B. Burke, and S. Labourey. *Enterprise JavaBeans*. O'Reilly, 4th edition, 2004.

[16] C. Norström, K. Sandström, J. Mäki-Turja, and N.-E. Bånkestad. Findings from introducing state-of-the-art real-time techniques in vehicle industry. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, 2000.

[17] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.

[18] D. S. Platt. *Understanding COM+*. Microsoft Press, 1999.

[19] D. S. Platt. *Introducing Microsoft .NET*. Microsoft Press, 3rd edition, 2003.

[20] F. E. Redmond III. *DCOM — Microsoft Distributed Component Object Model*. Hungry Minds, 1997.

[21] W. Shult and A. Polze. Aspect-oriented programming with C# and .NET. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2002.

[22] D. Sundmark, A. Möller, and M. Nolin. Monitored software components - a novel software engineering approach. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference, Workshop on Software Architectures and Component Technologies*, 2004.

[23] C. Szyperski. *Component-Software — Beyond Object Oriented Programming*. Addison-Wesley, 2nd edition, 2002.

[24] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.

[25] A. Zerzelidis and A. J. Wellings. Requirements for a real-time .NET framework. *ACM SIGPLAN Notices*, 40(2):41–50, 2005.