A Prototype Tool for Software Component Services in Embedded Real-Time Systems

Frank Lüders[1], Daniel Flemström[1], Anders Wall[2], and Ivica Crnkovic[1]

[1] Dept. of Computer Science and Electronics, Mälardalen University
Box 883, SE-721 23 Västerås, Sweden
{frank.luders, daniel.flemstrom, ivica.crnkovic}@mdh.se
[2] 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. However, such models have not been widely used in the domain of embedded real-time systems. 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 domains in the sense that a basic component model is extended with a set of services for the targeted application domain. A prototype tool for supporting these services is presented and its use illustrated by a control application.

1 Introduction

The use of software component models has become increasingly popular during the last decade, especially in the development of software for desktop applications and distributed information systems. Popular component models include *JavaBeans* [5] and *ActiveX* [4] for desktop applications and *Enterprise JavaBeans* (EJB) [11] and *COM+* [15] 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 these domains, there has been no widespread use of software component models in the domain of real-time and embedded systems, presumably due to the special requirements such systems have to meet with respect to timing predictability and limited use of resources. Much research has therefore been directed towards defining new component models for real-time and embedded systems. Typically, such models are based on static configurations of source code components and target relatively narrow application domains. Examples include the *Koala* component model for consumer electronics [22], *PECOS* for industrial field devices [6], and *SaveCCM* for vehicle control systems [7].

An alternative approach is to strive for a component model based on binary components and targeting a broader domain of applications, similar to the domain targeted by a typical real-time operating system. The approach pursued in this paper is to provide a combination of restrictions and extensions of an existing component model to adapt it to our target domain. Adapting an existing component model has several advantages: It may be possible to use existing (integrated) development environments; existing components can be re-used or adapted for the real-time domain; integration with application from other domains becomes significantly simpler, and so on.

Our previous work has demonstrated that the key concepts of the *Component Object Model* (COM) [3] can be used with advantage in the development of an embedded real-time system [10]. A study of COM and its extension *Distributed COM* (DCOM) [17] shows that these models are not inherently incompatible with real-time requirements, although some restrictions on how the models are used may be necessary to ensure predictability [9]. Some reasons that COM is an attractive starting point are that the model is relatively simple, commercial COM implementations are already available for a few real-time operating systems, and COM is already well-known and accepted in industry. The goal of this paper is to lay the groundwork for a software component model for embedded real-time systems by using the basic concepts of COM as the starting point and 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.

The remainder of the paper is organized as follows. In Section 2 we clarify what we mean by software component services and identify some useful services for embedded real-time systems. Section 3 is an overview of a prototype tool we are developing to support such services, including an example control application to demonstrate the use of the tool. Related work is reviewed in Section 4 and conclusions and some ideas for further work are presented in Section 5.´

2 Component Services

In this paper we define component services as solutions to common problems that can be added to components without modifying them and with little or no adaptation of application code. This is similar to the concept of component services in EJB and COM+, where examples of services include transaction control, data persistence, and security. Our focus is on services that address common challenges in embedded real-time systems, including logging, synchronization, and timing control. Traditionally, such functions have to be hand coded and off line deduced using complex theories, which can be very time consuming and sometimes impossible in complex industrial systems. If third party components are used, it may also be impossible to implement functions by modifying the components. In the following subsections we describe some of the services we have identified in more depth and outline how they may be implemented. In general, we propose that services are implemented through the use of proxy

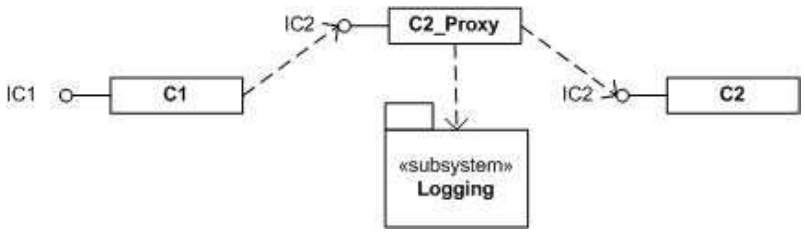

Fig. 1. Implementing a logging service through a proxy object

objects, which are automatically generated from configuration files written in an XML based format.

2.1 Logging

A logging service allows the sequence of interactions between components to be traced. Our suggested solution for achieving this is to use a proxy object as illustrated in the UML class diagram in Fig. 1. In the diagram, the object C2 implements an interface IC2 for which we wish to apply a logging service. A proxy object that also implements IC2 is placed between C2 and a client that uses the operations exposed through IC2. The operations implemented by the proxy forward all invocations to the corresponding operations in C2 in addition to writing information about parameter values, return codes, and invocation and return times to some logging medium. To add logging of all operation invocations through an interface, we simply add an entry in the configuration file:

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

No programming is required in the client C1 or the component C2. To add logging only for a particular operation, the entry is modified as follows:

```
  <interface name="IC2">
    <operation name="DoSomething">
      <service type ="Logging"/>
    </operation>
  </interface>
```

2.2 Execution Time Measurement

This service allows operation invocations to be monitored and information about execution times accumulated. Different measurements, such as worst-case, best-case, and average execution time may be collected. A possible use of the information is to dynamically adapt an on-line scheduling strategy. The suggested solution is to use a forwarding proxy that measures the time elapsed from each operation call till it returns and collects the desired timing information. As with the logging service, the time measurement service is specified in the configuration file:

```
<interface name="IC2">
  <service type="Timing">
    <measurement type="Mean" />
    <measurement type="Worst"/>
  </service>
</interface>
```

Again, no programming is required.

2.3 Synchronization

A synchronization service allows components that are not inherently thread-safe to be used in multi-threaded applications. The suggested solution is to use forwarding proxies that use the basic mechanisms of the underlying operating system to implement the desired synchronization policies. A synchronization policy may be applied to a single operation or to a group of operations, e.g. all operations of an interface or a component. Several different policies may be useful and will be described further in this section. Most synchronization policies rely on blocking and it may be useful to combine such policies with timeouts to limit blocking time. If the blocking time for an operation call reaches the timeout limit, the proxy return an error without forwarding the call. A more advanced timeout policy is one where the proxy tries to determine if a call can be satisfied without violating the timeout limit a priori and, if not, returns an error immediately.

The simplest synchronization policy is *mutual exclusion*, which blocks all operation calls except one. After the non-blocked call completes, the waiting calls are dispatched one by one according to the priority policy. This policy may be applied merely by adding an entry in the configuration file but, if timeouts are used, the client should be able to handle the additional error codes that may arise. Another class of synchronization policies is different *reader/writer* policies. These differs from the previously described policy in that any number of calls to read operations may execute concurrently, while each call to write operations has 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 calls are scheduled according to their priorities.

Using this policy requires that it be specified for each operation whether it is a read or write type of operation. This can be done in the component specification (e.g. a COM IDL file) or in the configuration file. If this is left unspecified for an operation, the proxy must assume it may write data. No programming is required, except possibly to handle error codes resulting from timeouts. For all synchronization policies, we may select if the priority of the dispatching thread should be the same as the calling thread, or explicitly specified in the configuration file. A specification of a reader/writer policy may look as follows:

```
<interface name="IC2">
  <service type="Synchronization" policy="RWPolicyX"/>
  <operation name="DoSomething" type="Write"/>
  <operation name="WriteData"   type="Write"/>
  <operation name="ReadData"    type="Read" />
</interface>
```

2.4 Execution Timeout

This service can be used to ensure that a call to a component´s operation always terminate within a specified deadline, possibly signaling a failure if the operation could not be completed within that time. The solution is to use a proxy that that use a separate thread to forward each operation call and then wait until either that thread terminates or the deadline expires. In the latter case the proxy signals the failure by returns an error code. Also, it is possible to specify different options for what should be done with the thread of the forwarded call if the deadline expires. The simplest option is to forcefully terminate the thread, but this may not always be safe since it may leave the component in an undefined and possibly inconsistent state. Another option is to let the operation call run to completion and disregard its output. Obviously, using this service requires that the client is able to handle timeouts. Again, the service is specified in the configuration file:

```
<interface name="IC2">
  <service type="Timeout" deadline="10ms" fail="Terminate"/>
</interface>
```

2.5 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* [8]. Among the services we have considered are cyclic execution, which are much used in process control loops [1], and support for redundancy mechanisms such as N-version components, which are useful in fault-tolerant systems [2]. The prototype tool presented in the next section includes an implementation of a cyclic execution service.

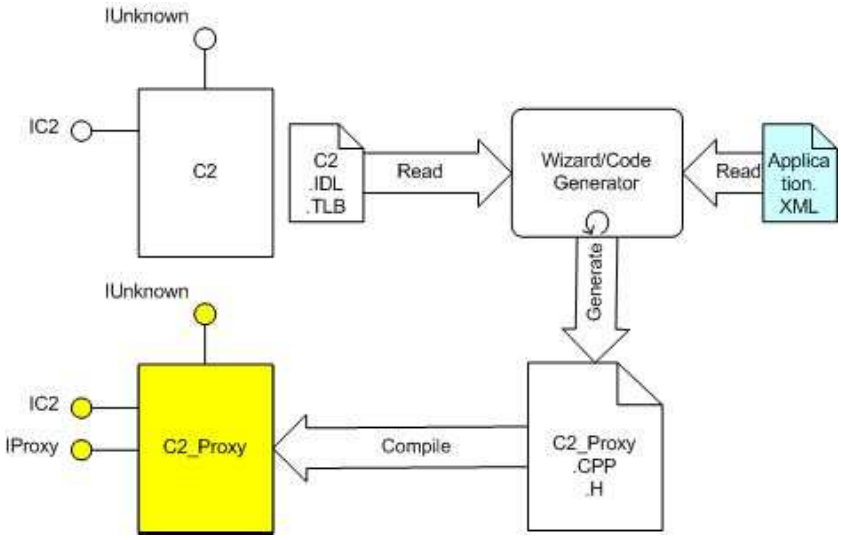

Fig. 2. Generating a proxy object for a component service

3 Prototype Tool

This section outlines a prototype tool we are developing that adds services to COM components on *Microsoft Windows CE*. The tool generates source code for proxy objects implementing services by intercepting method calls to the COM objects. The tool takes as inputs component specifications along with a specification of the desired services for each component. Component specifications may be in the form of Interface Definition Language (IDL) files or their binary equivalent Type Library (TLB) files. Desired services are either specified in a separate file using an XML-based format or in the tool´s graphical user interface, described further below. Note that access to component source code is not required. Based on these inputs, the tool generates a complete set of files that can be used with *Microsoft eMbedded Visual C++* (sic) to build a COM component implementing the proxy objects (i.e., the proxies are themselves COM objects). This process is depicted in Fig. 2.

3.1 Design Consideration

The use of proxy objects for interception is heavily inspired by COM+. However, rather than to generate proxies at run-time, we suggest that these are generated and compiled on a host computer (typically a PC) and downloaded to the embedded system along with the application components. There, the proxy COM classes must be registered in the COM registry in such a way that proxy objects are placed between interacting application components. This process may occur

when the software is initially downloaded to the system or as part of dynamic reconfiguration of a system that supports this. In the latter case, one can imagine updating or adding proxies without updating or adding any application components. The current version of the tool only generates proxy code and does not address the registration and run-time instantiation of components. This means that the client code must instantiate each proxy along with the affected COM object and set up the necessary connection between them. A desirable improvement would be to automate this task, either by generating code that performs setup for each proxy object or by extending the COM run-time environment with a general solution.

We consider staying as close as possible to the original COM and COM+ concepts an important design goal for the tool. Another goal is that the programmer or integrator should be able to choose desired services for each component without having to change the implementation or doing any programming. There are however cases, e.g. when adding invocation timeouts, 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 COM classes that, in turn, each implements a number of interfaces. All interfaces have a method called *QueryInterface* that allows changing from one interface to another on the same COM class. Since each proxy is implemented by a COM class, which must satisfy the definition of QueryInterface, we must generate one proxy for each COM class to which we wish to add any services.

3.2 Supported Services

Fig. 3 shows the graphical user interface of the tool. After a TLB or IDL file has been loaded all COM classes defined in the file are listed. Checking the box to the left of a COM class causes a proxy for that class to be generated when the button at the bottom of the tool is pressed. Under each COM class, the interfaces implemented by the class is listed and, under each interface, the operations implemented by the interface. In addition, the available services are listed with their names set in brackets. Checking the box to the left of a service causes code to be generated that provides the service for the element under which the service is listed. In the current version of the tool, a service for cyclic execution may only be specified for the IPassiveController interface (see example below), while all other services may only be specified for individual operations. Checking the box to the left of an interface or operation is simply a quick way of checking all boxes further down in the hierarchy.

If the cyclic execution service is checked, the proxy will implement an interface called IActiveController instead of IPassiveController (see example below). Checking the logging service results in a proxy that logs each invocation of the affected operation. The timing service causes the proxy to measure the execution time of the process and write it to the log at each invocation (if timing is checked but not logging, execution times will be measured but not saved). The synchronization service means that each invocation of the operation will be synchronized with all other invocations of all other operations on the proxy

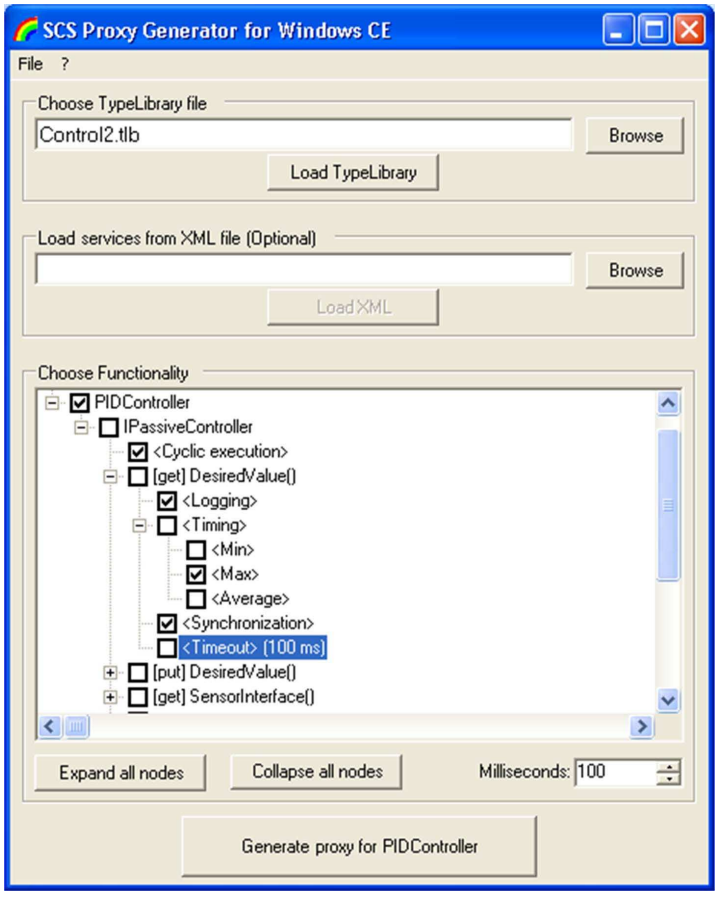

Fig. 3. The graphical user interface of the prototype tool

object for which the synchronization service is checked. The only synchronization policy currently supported is mutual exclusion. The timeout service has a numeric parameter. When this service is selected (by clicking the name rather than the box) as in Fig. 3, an input field marked Milliseconds is visible near the bottom of the tool. Checking the service results in a proxy where invocations of the operation always terminate within the specified number of milliseconds. In the case that the object behind the proxy does not complete the execution of the operation within this time, the proxy forcefully terminates the execution and returns en error code.

3.3 Example Application

To illustrate the use of the tool we have implemented a component that encapsulates a digital Proportional-Integral-Differential (PID) controller [1]. For the

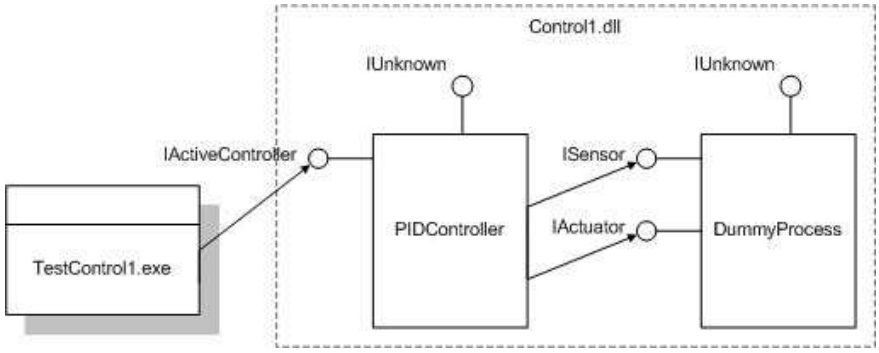

Fig. 4. An application using a controller component without services

purpose of comparison, we first implemented a component that does not rely on any services provided by the tool. Fig. 4 shows the configuration of a an application that uses this component. PIDController is a COM class that implements an interface IActiveController and relies on the two interfaces ISensor and IActuator to read and write data from/to the controlled process. For the purpose of this example, these interfaces are implemented by the simple COM class DummyProcess that does nothing except returning a constant value to the controller. The interfaces are defined as follows:

```
interface ISensor : IUnknown  {
  [propget] HRESULT ActualValue([out, retval] double *pVal);
};
interface IActuator : IUnknown  {
   [propget] HRESULT DesiredValue([out, retval] double *pVal);
   [propput] HRESULT DesiredValue([in] double newVal);
};
interface IController : IActuator {
   [propget] HRESULT SensorInterface([out, retval] ISensor **pVal);
   [propput] HRESULT SensorInterface([in] ISensor *newVal);
   [propget] HRESULT ActuatorInterface([out, retval] IActuator **pVal);
   [propput] HRESULT ActuatorInterface([in] IActuator *newVal);
   [propget] HRESULT CycleTime([out, retval] double *pVal);
   [propput] HRESULT CycleTime([in] double newVal);
   [propget] HRESULT Parameter(short Index, [out, retval] double *pVal);
   [propput] HRESULT Parameter(short Index, [in] double newVal);
};
interface IActiveController : IController  {
   [propget] HRESULT Priority([out, retval] short *pVal);
   [propput] HRESULT Priority([in] short newVal);
   HRESULT Start();
   HRESULT Stop();
};
```

IController is a generic interface for a single-variable controller with configurable cycle time and an arbitrary number of control parameters. PIDController uses three parameters for the proportional, integral, and differential gain. IActiveController extends this interface to allow control of the controller´s execution in a separate thread. The reason for splitting the interface definitions like this is that we wish to reuse IController for a controller that uses our cyclic execution service rather than maintaining its own thread. Note that IController inherits the DesiredValue property from IActuator. This definition is chosen to allow the interface to be used for cascaded control loops where the output of one controller forms the input to another.

The test application TestControl1.exe creates one instance of PIDController and one instance of DummyController. It then connects the two objects by setting the SensorInterfaca and ActuatorInterface properties of the PIDController object. After this it sets the cycle time and the control parameters before invoking the Start operation. This causes the PIDController object to create a new thread that executes a control loop. A simple timing mechanism is used to control the execution of the loop in accordance with the cycle time property. At each iteration the loop reads a value from the sensor interface, which it uses in conjunction with the desired value, the control parameters, and an internal state based on previous inputs to compute and write a new value to the actuator interface. To minimize jitter (input-output delay as well as sampling variability), this part of the loop uses internal copies of all variables, eliminating the need for any synchronization.

Next, the control loop updates its internal variables for subsequent iterations. Since the desired value and the control parameters may be changed by the application while the controller is running, this part of the loop uses a mutual exclusion mechanism for synchronization. In addition to performing its control task the loop timestamps and writes the sensor and actuator data to a log. The control loop is illustrated by the following pseudo code:

```
while (Run) {
   WaitForTimer();
   ReadSensorInput();
   ComputeAndWriteActuatorOutput();
   WriteDataToLog();
   WaitForMutex();
   UpdateInternalState();
   ReleaseMutex();
}
```

Note that, due to the simple timing mechanism, the control loop will halt unless all iterations complete within the cycle time.

Next, we implemented a component intended to perform the same function, but relying on services provided by generated proxies. A test application using this component and generated proxies is shown if Fig. 5. In this application, PIDController is a COM class that implements the IPassiveController interface. Note that, although this COM class has the same human readable name as in

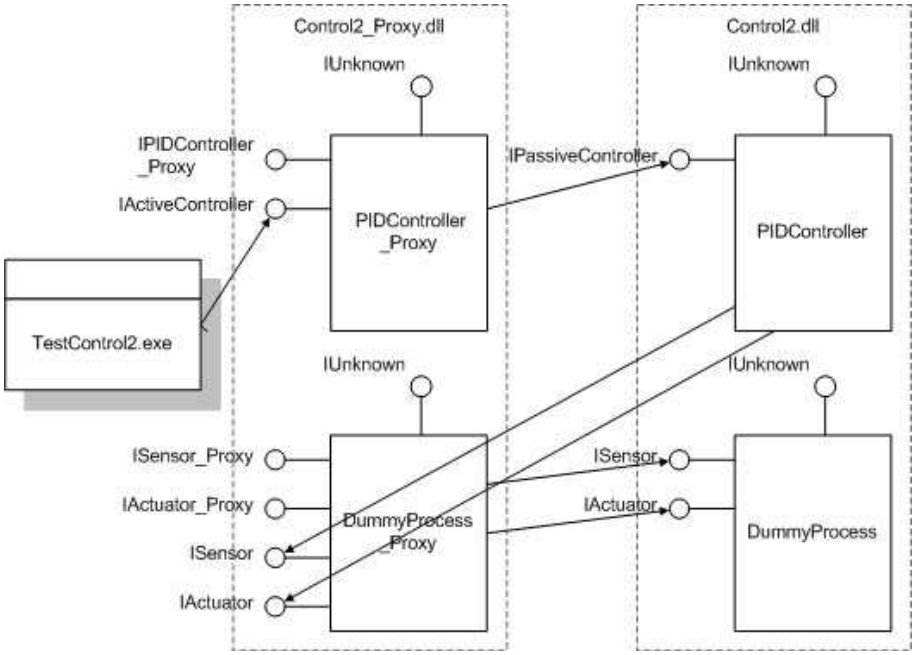

Fig. 5. An application using a controller component with services

the application described above, it has a distinct identity to the COM run-time environment. To avoid confusion we use the notation Control2.PIDController when appropriate. IPassiveController extends IController as follows:

```
interface IPassiveController : IController {
   HRESULT UpdateOutput();
   HRESULT UpdateState();
};
```

These operations are used by the PIDController_Proxy object to implement a control loop that performs the same control task as in the previous example.

PIDController_Proxy was generated with the use of the tool by checking the cyclic execution service under the Control2.PIDController´s IPassiveController interface and the synchronization service under the UpdateState operation as well as the operations for accessing the desired value and the control parameters. The DummyProcess_Proxy provides the interface pointers for the controller´s SensorInterface and ActuatorInterface properties. Behind this proxy is a DummyProxecess object with the same functionality as in the previous example. DummyProcess_Proxy was generated by the tool with the logging service checked. As a result, all data read and written via the sensor and actuator interfaces are logged. The interfaces ISensor_Proxy, IActuator_Proxy and IPID-

Controller_Proxy are only used to set up the connections between proxies and other objects. They are defined as follows:

```
interface ISensor_Proxy : IUnknown {
   HRESULT Attach([in] ISensor *pTarget);
};
interface IActuator_Proxy : IUnknown {
   HRESULT Attach([in] IActuator *pTarget);
};
interface IPIDController_Proxy : IUnknown {
   HRESULT Attach([in] IPassiveController *pTarget);
};
```

In order to evaluate to two test applications we built and executed them on the Windows CE 4.0 Emulator. Since the timing accuracy on the emulator is 10 milliseconds, it was not possible to measure any timing differences between the two applications. In both cases the controller worked satisfactory for cycle times of 20 milliseconds or more (the measured input-output delay as well as sampling variability was zero—from which we can only conclude that the actual times are closer to zero than 10 milliseconds). For shorter cycle times, both controllers ultimately halted since the limited timer accuracy caused the control loop to fail to complete its execution before the start of the next cycle. Also, we were not able to see any systematic difference in memory usage for the two applications. Clearly, further evaluation of the effects of the services on timing and memory usage is desirable.

To estimate the difference in programming effort and code size for the two applications we compared the amounts of source code and sizes of compiled files. These size metrics for the various components are presented in Table 1. The middle column shows the number of non-empty lines of source code. For the first three components, the number only include the source code of the C++ classes implementing the COM objects, i.e. the automatically generated code included in all COM components is not included. Taking these numbers as (admittedly primitive) measurements of programming effort, we see that using the tool to generate service proxies has resulted in a saving of 127 lines or 42 per cent. On the other hand, we see that the effort required for the client program is substantially greater in the case where the proxies are used. This is due to the need for the program to set up the connections between the proxies and the other objects. We conclude that the usefulness of our approach would greatly benefit from automation of this task.

As for the code size, there is only a small difference between the three COM components, leading to an overhead of roughly 100 per cent from using the proxies. This is largely due to the fact that the implemented COM objects are relatively small, leading to the obligatory house-keeping code of all COM components taking up a large percentage of the code size. For larger COM objects, the relative code sizes approaches the relative sizes of the source code. The small size of the COM objects is also the main reason that the component implementing the proxy objects is the largest of all the components. In addition, the generated

code is designed to be robust in the sense that all the operations of the proxy objects verify that the interface pointers have been set before forwarding operation calls. An obvious trade-off would be to sacrifice this robustness for less overhead in execution time as well as space. From the file size of the two test programs we find that the code overhead for setting up the connections between the proxies and the other objects is a little more than 10 per cent. This overhead, unlike the overhead on programming effort, cannot be eliminated by automating the setup task.

Table 1. Size metrics for components

Component	Lines of source code	File size in KB
Controller1.dll	300	56.5
Controller2.dll	173	53.5
Controller2_Proxy.dll	351	60.5
TestControl1.exe	81	12.5
TestControl2.exe	157	14.0

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 these. In addition, this section briefly reviews some existing research on binary components for real-time systems.

Microsoft´s component model COM [3] originally targets the desktop software domain. Thus, it has 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 Windows CE, 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+ [15] 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. Builders of COM+ application declare which services are required for each component and the 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 application domain. Since our component model targets a different domain than COM+, the services we have selected are different from those of COM+ as weöö. Secondly, we are inspired by the technique of providing services by interception. This mechanism is also used in other technologies and is sometimes called *interceptors* rather than proxies, e.g. in the *Common Object Request Broker Architecture* (CORBA) [14] and the *MEAD* framework for fault-tolerant real-time CORBA applications [13].

The approach presented in this paper is similar to the concept of aspects and weaving. In [21], A real-time component model called *RTCOM* is presented which have support for weaving of functionality into components as aspects while maintaining real-time policies, e.g. execution times. However, RTCOM is a proprietary source code component model. Moreover, functionality is weaved in at the level of source code in RTCOM whereas in our approach, services are introduced at the system composition level.

Another aspect-oriented approach is presented in [18], which describes a method using C# 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. An obstacle to the use of C# for the type of systems we are interested in is the lack of real-time predictability in the underlying *.NET Framework* [16]. The possibility of adding real-time capabilities to the .NET framework are described in [23].

A model for monitoring of components in order to gain more realistic WCET estimations is described in [20]. In this 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 technique is very similar to our execution time measurement service.

Another effort to support binary software components for embedded real-time systems is the *ROBOCUP* project [12], which builds on the aforementioned Koala model and primarily targets the consumer electronics domain. This work is similar to ours in that the component model defined as part of this project is largely based on the basic concepts of COM. Furthermore, the sequel of the project, called *Space4U* [19], also seems to use a mechanism similar to proxy objects, e.g. to support fault-tolerance.

5 Conclusion and Future Work

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 a tool that more or less invisibly add real-time services to a standard component model. The example application demonstrates that the use of generated proxies to implement services may substantially reduce the complexity of software components. Another conclusion to be drawn from the example is that our approach would benefit from also automating the configuration of applications with proxies.

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.

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. So far, our prototype implementation has been tested with the Windows CE emulator, where we have found no noticeable run-time overheads. In our future work, we plan to evaluate the solution experimentally on a system running Windows CE. Measurements will be made to determine the effect on timing predictability as well as time and memory overhead.

We furthermore aim to empirically evaluate our approach with respect to its effect on development effort and such quality attributes as reliability and reusability. Our hypothesis concerning reliability is that it may improve as a result of reduced complexity of application components, provided off course that the generated proxies are reliable. We also believe reusability may be affected positively, as e.g. the use of synchronization services could make it easier to reuse components across applications that share some functionality but rely on different synchronization policies. The primary evaluating technique will be to conduct replicated student projects where software is developed both with and without the prototype tool. A possible complementary technique is industrial case studies, which implies a lower level of control and replication but may allow more realistic development efforts to be investigated.

References

1. K. J. Åström and B. Wittenmark. *Computer Controlled Systems — Theory and Design*. Prentice Hall, 2nd edition, 1990.
2. A. Avizienis. The methodology of N-version programming. In M. R. Lyu, editor, *Fault Tolerance*. Wiley, 1995.
3. D. Box. *Essential COM*. Addison-Wesley, 1997.
4. D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

5. R. Englander. *Developing Java Beans*. O'Reilly, 1997.
6. T. Genßler, C. Stich, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, 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.
7. 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.
8. G. T. Heineman and W. T. Council. *Component-Based Software Engineering — Putting the Pieces Together*. Addison-Wesley, 2001.
9. F. Lüders. Adopting a software component model in real-time systems development. In *Proceedings of the 28th Annual IEEE/NASA Software Engineering Workshop*, 2004.
10. 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 Development for Embedded Systems*. Springer, 2005.
11. R. Monson-Haefel, B. Burke, and S. Labourey. *Enterprise JavaBeans*. O'Reilly, 4th edition, 2004.
12. J. Muskens, M. R. V. Chaudron, and J. J. Lukkien. A component framework for consumer electronics middleware. In C. Atkinson, C. Bunse, H.-G. Gross, and C. Peper, editors, *Component-Based Software Development for Embedded Systems*. Springer, 2005.
13. P. Narasimhan, T. A. Dumitras, A. M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava. MEAD: Support for real-time fault-tolerant CORBA. *Concurrency and Computation: Practice and Experience*, 17(12):1527–1545, February 2005.
14. Object Management Group. Common object request broker architecture: Core specification, March 2004. OMG formal/04-03-12.
15. D. S. Platt. *Understanding COM+*. Microsoft Press, 1999.
16. D. S. Platt. *Introducing Microsoft .NET*. Microsoft Press, 3rd edition, 2003.
17. F. E. Redmond III. *DCOM — Microsoft Distributed Component Object Model*. Hungry Minds, 1997.
18. W. Schult 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.
19. Space4U Project. Space4U public hompage, January 2006. http://www.hitech-projects.com/euprojects/space4u/, Accessed on 28 April 2006.
20. 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.
21. A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Aspects and components in real-time system development: Towards reconfigurable and reusable software. *Journal of Embedded Computing*, 1(1), February 2004.
22. 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.
23. A. Zerzelidis and A. J. Wellings. Requirements for a real-time .NET framework. *ACM SIGPLAN Notices*, 40(2):41–50, 2005.