Model-Driven Development of Component Infrastructures for Embedded Systems.

Conference Paper · January 2005

Source: DBLP

CITATION
1

READS
31

1 author:

Markus Völter
independent/itemis
148 PUBLICATIONS **4,239** CITATIONS

SEE PROFILE

Model-Driven Development of Component Infrastructures for Embedded Systems

Version 2.0, Sept 29, 2004
Submission to MBEES Workshop 2005

Markus Völter
voelter – ingenieurbüro für softwaretechnologie
Ziegelaecker 11, 89520 Heidenheim, Germany
Tel. +49 7321 97 33 44
voelter@acm.org

ABSTRACT

Component infrastructures such as Enterprise JavaBeans, Micorosoft's COM+ and CORBA Components have become a de-facto standard for enterprise applications. Reasons for this success are the clean separation of technical and functional concerns, COTS containers (applications servers), and the resulting well-defined programming model and standardization. To benefit from these advantages in the domain of embedded systems, the same concepts can be used, but a different implementation strategy is required: monolithic application servers are not suitable because of the limited resources regarding computing power, memory, etc. on the device. An alternative can be based on using a family of code-generated containers. The container is generated from models that specify interfaces, components, system topologies and deployments. In addition to motivating the problem and looking at related work, this paper gives general guidelines for the design and implementation of such infrastructures and describes a prototype implementation that has been implemented recently. We also look at the advantages of using such an approach for the electronic control units in vehicles and the benefits the approach could have with regards to vehicle diagnostics.

The rest of the paper is structured as follows: The introduction in section 1 briefly describes embedded software development state-of-the-art and outlines some problems with this approach. Section 2 describes the solution proposed in this paper in a general fashion. The prototype implementation is descrbed in section 3, including the concrete motivation for its implementation. Section 4 describes related work, while section 5 very briefly talks about practical experience.

Keywords

Components, containers, Model-Driven Software Development, Code Generation, Embedded Systems

1. INTRODUCTION

1.1 Embedded Software Requirements

Embedded software typically faces some unique constraints not found in desktop software or enterprise systems:

- *Limited Resources:* embedded devices typically have only a limited amount of resources. This may include memory, processing power, electrical power (for battery-powered devices), and network bandwidth. In contrast to other domains, it is often not technically and ecomomically feasible to increase the available resources – instead, the software has to be optimized to cope with the situation.
- *Real-time Requirements*: Most embedded software interacts with its environment in some way. Typically, the temporal aspect of these interactions is constrained. Computations have to happen within specific time boundaries. Depending on the strictness of these constraints, we talk about soft or hard real-time requirements [25].
- *Hardware integration:* In many embedded systems, some interaction with hardware devices (actors, sensors) is necessary. While this typically imposes real-time constraints, in addition it typically also requires the developer to deal with low-level aspects of the hardware.
- *Reliability:* Many embedded devices cannot easily be repaired, rebooted or reconfigured once they have been deployed. They are often a part of some safety-critical system where failures are hardly acceptable. Also, the device simply might not be accessible to maintenance because it is used somewhere where access is not feasible (in a spacecraft, or just simply in ten million mobile phones).
- *Unit-based Cost Structures:* Embedded Systems are produced and sold in large quantities. As a consequence, the development cost per sold device is rather low, the unit and production cost becomes dominant – this results in a strong focus on cheap (and thus lower performance) processors and as little memory as possible.

Based on the requirements discussed in the previous section, we can say that embedded software needs to be more reliable than many other kinds of software, it needs to optimize it's computations for speed and resource consumption, the code size must be minimized and in many cases, real-time requirements need to be verified (maybe empirically) before the software is deployed.

1.2 State-of-the-Art

Because of these special requirements, a lot of software for embedded devices is still developed manually, from scratch for each new project. Large-scale reuse is not applied because of the requirement to optimize each piece of software for its particular environment. As a consequence, many techniques that are used to good effect in non-embedded development are not widely used in embedded systems development. Examples are object-orientation, frameworks or reflection. COTS middleware (such as minimum CORBA [21]) is only recently starting to spread in the embedded community.

There are several typical high-level application architectures for embedded systems:

- For either very simple of very constrained systems, application code is written directly for the hardware of the device. No operating system is used, some reusable libraries are typically employed, however.
- To allow for some degree of portability of these applications, sometimes a thin abstraction layer is used between the application and the hardware. This can be seen as a simple, custom-developed operating system. Porting the abstraction layer to another device allows for some limited reuse.
- More complex applications typically use more or less powerful realtime operating systems such as VxWorks [31], QNX Neutrino [24] or Osek [26]. Depending on the specific operating system, it handles tasks such as threading, scheduling, device comminication, a file system, etc.
- The most sophisticated application architectures use an OS abstraction layer on top of the operating system to be able to exchange the operating system while not having to rewrite the application code. Figure 1 shows this last alternative.

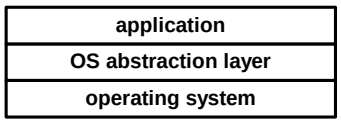

Illustration 1: High-Level Application Architecture for Embedded Systems

Independent of this structure, parts of the application are generated from models such as state charts or signal flow diagrams and need not be implemented manually. See the related work section for a more detailed discussion.

1.3 Component Infrastructures

Component infrastructures [30] provide a (potentially distributed) execution environment for software components. The execution environment is typically called a component container, or container, for short. Components cannot be executed standalone, they require the container to provide essential services. These services handle the technical concerns of an application. Technical concerns are typically cross-cutting aspects that are not directly related to the application functionality implemented with the components. What exactly consititues techical concerns depends on the application domain. In an enterprise environment, the technical concerns are things such as transaction management, resource access decision, fail-over, replication and persistence. The benefit of a component-based approach is that the component (i.e. application) developer does not need to implement the technical concerns over and over again. The developer only *specifies* the container services required by a component, and the container makes sure these services are available to the deployed components. Containers are implemented against some kind of standard (such as EJB [28]) by professional container vendors. Applications just use the containers as they are. Application developers thus don't need to be experts with respect to the (typically non-trivial) technical concerns. The following paragraph lists essential building blocks for component infrastructures. For a more detailed explanation see [30].

A *component* encapsulates a well-defined piece of the overall application functionality. An application is assembled from collaborating components accessing each other through a well-defined *component interface*. Because the functionality of components and the way to access it is well-defined and self-contained, preexisting components can be reused in several applications. The interface is technically separate from the *component implementation* which can be exchanged without affecting clients. The strict separation of interface and implementation allows the container to insert *component proxies* into the call chain between the clients and the implementation. On behalf of the container, these proxies handle technical concerns. The *lifecycle callback interface* of a component is used by the container to control the lifecycle of a component instance. This includes instantiating components, configuring instances, activating and passivating them over time, checking their state (running, standby, overload, error, …) or restarting one in case of severe problems. Because all components are required to have the same lifecycle interface, the container can handle different types of components uniformly. *Annotations* are used by the component developer to declaratively specify technical concerns (i.e. which of a container's services are needed by a component and in which way). A *component context* is an interface passed to the component implementation that allows it to control some aspects of the container (e.g. report an error and request shutdown). A component is not allowed to manage its own resources. It has to request access to *managed resources* from the container, allowing the container to efficiently manage resources for the whole application (i.e. several components). These resources also include access to other component interfaces. All the resource a component instances wants to use at runtime must be declared in the annotations to allow the container to determine if a component can correctly run in given context, and prepare accordingly. When operations are invoked on instances, the invocation might carry an additional *invocation context* that contains, in addition to operation name and parameters, data structures which the container can use to handle the technical concerns (such as a transaction id). Last but not least, a component is not just dropped

into a container, it has to be explicitly installed in it, allowing the container to decide (based on the annotations and required resources) if it can host the component in a given environment.

1.4 Benefits and Liabilities of Component Infrastructures

Using component infrastructures provides several benefits:

- *Portability:* Components are developed against the interfaces of the container, the container can adapt this to different environments (such as operating systems, databases or transaction monitors in the enterprise world).
- *Potential for Container-based Opimization:* Within the boundaries specified by the specifications of the container and the lifecycle interface, the container is free to optimize different aspects of the application.
- *Standardized, Simplified Programming Model:* Because the environment in which components execute is well-defined, and because the developer does not need to deal with low-level implementation details of the technical concerns, the programming model for application devleopers is simplfied and consistent over the family of applications implemented for the same container.
- *Clearly defined developer roles:* Because application developers can focus on their specific application requirements, and because infrastructure experts deal with the implementation of the container, both aspects can be implemented by people who are experts on their respective field, improving the quality of the the software.

Of course there is no such thing as a silver bullet. Typical component infrastructures also suffer from some liabilities. Note that none of these liabilities are inherent to the approach taken by component infrastructures, however, they can be observed in all of today's mainstream implementations:

- *Performance Overhead:* Because requests are intercepted by the container, and because its services are implemented generically to be reusable, performance of component-based applications is impacted.
- *Loss of control:* Some people feel that handing over control over technical aspects to the container limits their control over what is actually happening. While this is true, in most scenarios this is not a liability, however, because the container can handle most of these aspects better and more reliably than code handcrafted by the average developer.
- *Large and heavy:* Most of today's implementations are large and heavy software monsters. Installing, configuring or (re-) starting them can take a while.
- *Complexity:* Of course, by providing a reusable solution to a recurring problem, component infrastructures imply a lot of accidental complexity. This might be a problem for safety-critical applications.

This paper proposes that the benefits presented above would also be desirable in the embedded software world, while ideally not showing the same liabilites. Sections 2 and 3 describes an approach how this could work.

2. PROPOSED SOLUTION

In this paper we propose a component infrastructure that uses model-driven code-generation [12] instead of a generic container. In this context it is critical to understand that we do *not* propose to code-generate the components, i.e. the core application logic. Several tools exists (see related work and [10], [13]) that can generate source code from state charts or signal flow diagrams, and wrapping such functionality in a component is simple. Instead we propose to generate the complete infrastructure that is needed to execute the components on an embedded device, aka the container.

2.1 Required Features

The following features are required for a component container for embedded systems:

- *Portable:* Components that are written for a specific container must be able to run on every (real-time) operating system for which a container implementation is available. This requires an abstraction of operating system features[1].
- *Modular:* Enterprise containers typically ship as a big monlithic application that is capable of handling all features of the respective specification, such as transactions, security and persistence. In the embedded world it is not acceptable to carry "excess baggage" in case some features are not needed in a particular application scenario. Consequently, the container infrastructure must be modular itself, only including those features in a particular container instance that are really needed and supported by the target device.
- *Simple:* Again in contrast to the well-known component containers such as EJB, CCM or COM+, a container infrastructure for embedded systems must be lightweight, providing a really simple programming model. Because the target systems (devices) are much more diverse than in enterprise systems, we should focus on the reusable core.
- *Deterministic:* For many embedded applications, determinsm is a critical property. Determinism means that we know in advance (i.e. before runtime) how long something (an operation, a statement) takes to execute. Using dynamic featues such as polymorphism, reflection, etc. makes this kind of determinism much harder to achieve.

[1] Portability here does does not neccessarily mean programming-language independence. This is so because the application logic is implemented in a specific programming language, and also the templates (see later) are implemented in a specific language. Of course, the concepts introduced below are independent of any particular programming language, but implementations are not.

2.2 Basic Design Decisions

Before we actually look into the implementation of the prototoype, let's look at a couple of additional design decisions that have influenced the system concept, and the prototype described in 3.

First of all, we assume that a system configuration (i.e. the set of components running in a container in the context of an application) is determined statically, before it executes. This is typical for many, but not all embedded applications. So, when the container is generated, the generator knows which components need to run in the container and it knows their resource requirements. As a consequence, the container can validate large parts of the system before it actually starts up. It can detect if a component wants to talk to another component that isn't there, or if a component requires services from the container that cannot be provided because of limitations of the device.

The resulting absence of dynamic decisions has one very big benefit: We are able to statically analyse the code for resource problems or scheduling problems, and with regards to performance and timing using standard code analysis tools [5]. This would not be possible if decisions are taken at run- or load-time. This is the determinism property described in 2.1. It is not important for all kinds of embedded systems, but it is important for many.

Second, we assume that the components themselves are written manually. This means that the container generator does not care about the implementation of the components. While there might be some "stub generation" from more abstract interface specifications (such as IDL or a UML model), the component implementation is provided by the application programmer. Of course, the developer is free to include code in the components that has been generated by state chart or signal flow tools (such as [10], [13]).

2.3 The solution in a nutshell

The following illustration shows the approach described in this paper in a nutshell.

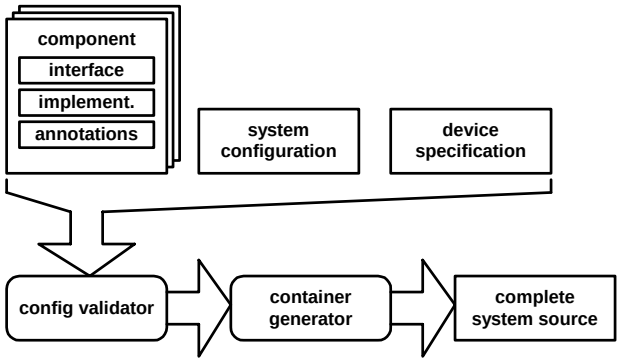

Illustration 2: The proposed soltution in a nutshell

Let's look at the different components in detail. *Components*, as described several times now, contain the application logic, the functional aspect of an application system. The building blocks that constitute a component are the component interface, the implementation and the annotations which state the requirements of the component regarding the container and other components. The *system configuration* specifies which instances are needed of which components, how their resource requirements will be satisfied ("wiring" the components) as well as the configuration of container services and how they apply to components and their instances. The *device specification* describes the available features of the target device (and operating system) on which the resulting application should be deployed. This ultimately determines which container features are available in the target system, as well as how these features are implemented.

All these artifacts are supplied to the *configuration validator* (or buildability checker) that checks if the container will be able to work correctly (as far as this is possible at this early stage). If it determines it is, the artifacts will be supplied to the *container generator* which generates the source code for the container according to the system configuration taking into account the specification in the annotations and the device configuration. Otherwise the container is not code generated and thus cannot be deployed on the device (which is good, because it would not work correctly.)

If the container was generated correctly, this code is then compiled with a normal programming language compiler for the respective target. Optionally, it can be analysed statically to verify its correctness (as far as such static code analysis is feasible [5] – this is no different than static analysis of hand-written code).

2.4 Technical Concerns: Container Features

The selection of what constitutes the technical concerns in a particular family of applications depends on the specific requirements of the domain. Unlike in enterprise systems, where all applications typically consist of some database/transaction related logic, embedded systems as more diverse and it is thus not feasible to decide once and forall on what constitutes the technical concerns. This is the reason why we do not propose one specific embedded container in this paper, but rather an approach, or an architecture, to construct such containers for a specific software system family.

However, there are some candidate aspects that lend themselves to being implemented as container features.

- *Scheduling:* Controlling of thread and task priorities, creation and maintenance of thread pools, deadlock detection
- *Interrupt Handling:* A high-level notification interface in case interrupts occur can be provided by the container. It can invoke previously configured operation on components instances.
- *Simple Event Propagation:* The propagation of events from one component to another, synchronously or asynchronously can be supported by the container.
- *Timer:* Time-based events can be triggered by the container
- *Remote Communication:* In case communicating components reside in different containers on different devices (boards, controllers, computers, …) in a distributed system, the container can take care of remoting [33] (using CORBA [16], CAN [6], plain sockets, ASN.1 [3], etc.)

- *Generic Driver Interface:* Accessing low-level drivers can be simplified. The container can provide more abstract, higher level access to lower lever drivers, or convert data structures depending on the device. In addition, it can control concurrent access to shared hardware devices from several components.
- *Lifecycle Control:* In case determinism is not absolutely important for a system, the container can control the lifecycle of component instances. This includes restarting instances in case errors are detected, lazily inszantiating instances only when they are actually needed, passivating instances when they have not been used for a specified period of time, etc.
- *Resource Control:* The container can control resources, or manage pooling of critical resources. Also, it can enforce quota allocated to component instances (e.g. memory or disk quota) and coordinate concurrent access to shared resources.
- *Safety Watchdogs:* The container can provide watchdogs for critical system properties. It can safely shut down the system if an error is detected.
- *Advanced Error Detection:* In embedded systems with more or less strict timing constraints, many errors are an indirect consequence of a (seemingly unrelated) timing problem or illegal invocation sequences on component operations. If interfaces are annotated with state charts that include timing constraints, a container can be generated that controls these timing and state constraints and report the real root-cause errors. In general, the concept of "programming by contract" can be used effectively because the container can contain code to actually check pre- and postconditions, as well as invariants, specified for interfaces of the components.
- *Management Façade:* The container can provide a coherent, homogeneous external interface for the management of embedded applications. For example, it can provide an SNMP MIB or issue SNMP traps [27] for the container itself and the components inside. Management of devices will be simplified, and application programmers need not bother with specifics of management protocols such as SNMP.

2.5 Interface specifications

Interfaces play the central role in component infrastructures. Interfaces define contracts among components, and between the container and the components. In traditional systems, interfaces are typically defined as a set of operations including typed arguments, as well as a return type. For serious system composition, more detail must be given on interfaces, including

- services required from the container to allow the component to run
- other component's interfaces required by a component
- timing constraints regarding interface operations
- pre- and postconditions for operations, or a state machine that defines legal invocation sequences
- data published by a component, or data consumed (required) by a component

Interface definitions as outlined above are logical definitions of what a components provides, or requires. It does not say anything about how these interfaces are implemented. The realization of the interfaces can be supported by the generated container. For example,

- operations can be called directly if the caller and the callee are colocated in the same process, or can include proxies and some kind of remoting infrastructure for remote calls.
- published or required data items can be stored to/retrieved from a shared memory area or it can be put on/taken from a CAN bus.
- timing constraints or pre/postconditions can be checked by the container and errors can be reported

2.6 Applicability of the solution

Considering the different architectures for embedded systems as explained in 1.2 the question is: in which architecture can the proposed approach be used sensibly? Let's look at each of these architectures in turn.

- *No operating system:* In these very small systems, the proposed architecture is very suitable. First of all, software on these devices typically is very static, not featuring dynamic aspects. Efficiency and small code size is important, while we still need some flexibility regarding different hardware platforms/devices (because there is no OS). Also, because there is no OS, there is a lot of use for reusable, cross-cutting technical concerns handling of the container. The container thus serves as an efficient implementation of the abstraction layer described in the second architectural alternative – providing flexibility while still being efficient.
- *With (realtime) operating system:* realtime operating systems (as any operating system) typically provide APIs on a very low level. Also, there is no handling of domain- (or software system familiy-) specific technical concerns. Containers can provide this higher-level abstractions. The container can also serve as a means of integrating different tools, systems, middlewares, etc. For example, the container can provide remoting based on CORBA or a different middleware.

3. THE PROTOTYPE

This section looks at a prototype implementation of the proposed architecture. The prototype is still work in progress, but considerable functionality has already been achieved.

3.1 The example domain: automotive ECUs

The prototype of a generative component infrastructure for embedded system is currently being developed in the context of automotive ECUs, the electronic control units (i.e., computers and controllers) that control various features of a modern car, such as engine, gearbox, air conditioning, the brake system or the dashboard. A modern middle-class vehicle has about thirty ECUs

installed, constituting a distributed system typically based on a CAN network [6] or proprietary topologies. There are several reasons why the software structure of ECUs needs to be standardized and enhanced, for example using a component infrastructure, in addition to the reasons given in 1.4:

- The ECUs of different vendors need to interoperate in the context of a vehicle. A coherent software infrastructure is thus necessary.
- The separation of application logic and technical infrastructure as explained in 2.4 is especially important, since the same application logic (e.g. brake control) should be reusable in the context of several vehicles, potentially featuring different technical infrastructures. The container can adapt for this.
- Configurability is another important aspect. You want to be able to run the same piece of functionality on different ECUs depending on the vehicle model – you want to utilize the available ECUs as good as possible. A graphical configuration tool that helps is distributing the components to containers and devices.
- The container can also implement a global vehicle state manager (ignition on/off, engine on/off). For example, you are not allowed to reflash (i.e. reprogram) an ECU while the vehicle is driving. As the container can intercept all interactions among components, it is easy for it to track global state and either notify components of state changes or prohibit certain interactions that are not currently allowed.
- Last but not least, diagnosability is a serious issue. After the vehicle has been delivered to the customer, it must be possible to diagnose problems in garages. Typically, an external diagnosis tool is attached to the vehicle. The tool reads the ECUs' internal error buffers and reasons on these errors with the goal of finding the root cause of the problem. Making these tools more efficient and accurate is one of the most urgent tasks for today's after-sales operations. As a precondition, the errors reported by the vehicle must be correct, expressive and accessible through a standardized interface. Also, the description of the ECU topology of the car (which is currently kept outside of the cars in the tool) must be consistent with the actual network deployed in the car. Providing the information based on a reflection on the component infrastructure, can help to avoid inconsistencies.
  Also, error conditions can be specified abstractly as part of the component definition (such as "raise XYZ error when speed < 100 and fuelLevel > 10"). Code can be generated that efficiently implements the detection of this error on a specific platform.

3.2 Prototype Implementation and Technologies

The prototype is implemented in C/C++. The following illustration shows how the prototype is implemented in general.

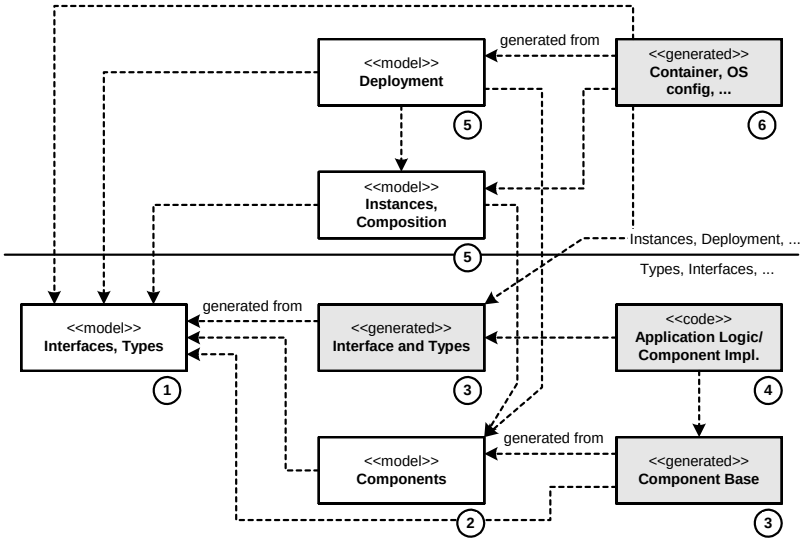

Illustration 3: Prototype implementation structure

In the first step, interfaces and (potentially) complex data types are modeled (this can be done using UML 2.0, or using other DSLs). In a second step, we define the components including the interfaces they provide, and the interfaces they require. From these two models, component base code (header files) can be generated; also, complex type implementations and interfaces are created. In the next step (step 4), the implementation of those components can be created manually by the developers. This completes the first phase, *component development*.

In a second phase, *system development*, we define component instances and the connections among those instances. Then we specify the deployment of these instances on hardware elements of the system (those specifications are not shown in illustration 3). Based on these models, we can generate the containers for the hardware elements, as well as the OS config files.

In the prototype, we generate the code using the openArchitectureWare generator framework [4]. This particular generator tool is based on an explicitly programmed metamodel which is implemented in Java. Thus, the generator needs to be supplied with the metaclasses that describe the metamodel for the various models used in the approach. Also, we need to define templates that specify the mapping from the metamodel to the generated source code

The output of the generator is the source code (skeletons) for the components, the complete container implementations, as well as a *make* file used to compile and build the container and the components. The subsequent C++ compiler/linker must be fed with these generated sources, the manually created component implementation files as well as additional runtime libraries. We also generate suitable config files for the operating system, OSEK in our example.

The configuration file itself can be set up using a graphical configuration tool based on the Eclipse framework. It imports the interface definitions from the models and allows the configuration of instances, their threading behaviour, association of instances to containers, automatic remoting, event propagation, etc.

3.3 Prototype features

This section focuses on some noteworthy features implemented in the prototype.

3.3.1 Proxies

Application functionality is realized by having components collaborate – a component instance invokes operations on other components instances. It is important that such invocations implies only the smallest overhead possible. For example, if the container does not need to intercept invocations (because the configured technical concerns don't require intervention by the container), an operation invocation does not have *any* overhead at all. An ordinary method invocation is used. If, for example, a method should be invoked asynchronously (which needs to be specified at generation time), then the container generator generates a proxy for the instance. The proxy, when an operation is invoked, creates a thread (or obtaines one from a pool) and then subsequently invokes the operation on the instance in this thread. Whenever a client (component) wants to get a reference to the instance, the container makes sure that the proxy is returned instead of the real instance. Consequently, operations on the instance are invoked asynchronously without any involvement of the client or the component implementation.

The same conceptual approach is taken when an instance invokes operations on a remote component instance. The client component's container contains a proxy that translates the call to whatever remoting technology is configured – CORBA, sockets, or something else. In the server container, there is another proxy that receives the remote message and invokes the target operation on the target instance. Both proxies are automatically generated based on information in the configuration file.

3.3.2 Signals

Signals are simple noifications (typically integers) that are exchanged among component instances. The propagation of signals is handled by the container. The configuration file specifies which signals should be propagated to which component instance. If a component raises a signal, the container propagates the signal to all receivers. If the configuration file specifies that the propagation should happen asynchronously, the container creates a thread and handles propagation in this thread.

Note that if there are no signals to be propagated, the generated container does not contain any propagation logic, i.e. no runtime overhead and no size overhead.

3.3.3 Diagnostic Features

Last but not least let's have a look at the diagnosis-specific extensions for the tool.

First of all, a generic diagnostic interface is defined in the model. All components are required to implement this interface for generic access by an external diagnosis tool and to allow components to query other components for their state (à la "if I have a problem, let's see if my supplier also has a problem which might cause my own problem"). The code generator is later supplied with the DTC/FaultCode specification (a specification that defines which errors might occur in an ECU and how the ECU can detect them) so that the implementation for the diagnostic interface can be generated to a large extent. Note that this very same specification, together with the config file (which specifies the topology and the dependencies) can then be supplied to external diagnosis tools, which uses this information as the basis for its diagnoses. The generator also receives the state/timing information for the component interfaces. The generator generates code into the container that diagnoses errors in the timing/state sequence of components efficiently and reports them. Finally, the implementation of the application logic (e.g. controlling the anti skid system) can be generated from other tools such as Matlab [13] or Statemate [10], if necessary.

4. RELATED WORK

This section looks at related work done by other people. I have separated this section into a couple of subsections, because my work touches on several other areas.

4.1 Infrastructures for embedded systems

Several efforts are currently undertaken regarding the provision of infrastructure for embedded software development. Let's look at some of them.

OSGi, the Open Services Gateway Initiative [23] aims at providing infrastructure for dynamic service infrastructures on devices, so-called gateways. Gateways are considered to be "facades" around complex distributed, embedded systems (such as vehicles, wired homes, industrial estates) onto which services can be installed remotely. OSGi implementations help in installing these services, tracking dependencies among them, starting and stopping services, etc. In addition to this basic functionality, OSGi provides a set of services, e.g. an simple HTTP server, messaging or a generic driver interface for hardware. In contrast to the approach proposed here, there is no notion of a container as such, because the OSGi infrastructure does not handle crosscutting technical concerns for the installed services. It only serves as a framework that handles some, well-defined tasks. Also, OSGi targets dynamic environments where services can be dynamically installed and removed at runtime. This is in direct contrast to the approach presented in this paper, where as much as possible is generated statically. As such, this approach is targetted at the core embedded system whereas OSGi systems are targetted for dynamic gateways.

There are several implementations of **CORBA** [16] for embedded devices. The TAO ORB [7] can be used in embedded settings, it is available for realtime embedded operating systems such as QNX [24] and it is currently ported to really small, embedded OSeK environments [26]. CORBA, however, does not provide a component infrastructure. CORBA, especially the embedded versions based on the minimum-CORBA specification [21], provides a means allow remote operation invocations, not very much more. As such it can serve as a basis for some of the features provided by the container proposed in this paper. The CORBA component model [16], which does provide a container/component infrastructure on top of CORBA is a very sophisticated component infrastructure that is much too complicated for the embedded world. Also, no implementations are currently available.

4.2 Lightweight component containers

Most current implementations of component containers (specifically for EJB) are rather large, monolithic tools and neither intended nor suitable for embedded systems. However, a couple of projects aim at creating smaller, more modular component containers. For example the JBoss EJB implementation [11] has the concept of the "generalized aspect container". A container for components can be configured with an arbitrary set of interceptors that can each handle a specific aspect. This is a rather flexible approach, and the functionality of the container can be adapted to the specific needs of the system. However, JBoss uses reflection for all this and thus does not optimize for performance. While this approach is conceptually not far from what I propose in the paper, it is not suitable to use in embedded systems because of its dynamism. Also, it is currently bound to the EJB component model [28] and thus, the Java programming language.

In general, aspect oriented programming [2] is a way to selectively introduce crosscutting (typically technical) concerns into an application. It is thus a good way to build lightweight, modular component containers. A container feature is basically an aspect. AspectJ [9] is a Java AOP extension that can be used for this purpose, specifically as it is based on static code weaving [12]. The Java Aspect Components framework [1] is an attempt at building a generic framework for providing a selection of technical concerns for enterprise applications (failover, persistence, GUI, etc.). It is based on Java and uses mainly reflection and other dynamic techniques. Again, this tool is not explicitly targetted for small embedded environments.

In several vertical domains, standards are currently being defined for component infrastructures. A popular example is the AUTOSAR [35] standard that is currently being defined for the automotive domain.

4.3 Modularized Infrastructure

The idea of providing reusable services to applications is not revlolutionary at all. Operating systems do exactly this. Realtime operating systems for use in embedded systems provide a set of services to the applications that run on them. Some realtime and embedded operating systems such as QNX [24], OseK [26] or even Windows CE [15] are even customizable in the sense that the image that is deployed to the embedded device only contains the features required by the particular application. As such it can be seen as some kind of "component container" with the applications being the components. However, there are several important differences: First of all, the developer is not able to extend the infrastructure (i.e. the operating system) with additional technical concerns. In contrast, the approach presented in this paper can be adapted with new container features at any time. Second, operating systems do not do things such as creating proxies for threading or remote access to other programs. Operating system features, especially embedded, realtime OS features, are typically much more low-level.

4.4 Code generation

The approch presented in this paper is based primarily on soure code generation (for an overview of code generation technologies, see [12]). Source code generation is already heavily used in embedded software development in tools such as Statemate [10] or Matlab/Simulink [13]. However, these tools don't use the principle of separation of concerns to factor out and generate the code for handling the technical concerns, instead they typically create the "application logic" from signal flow diagrams or state charts. These tools can be easily integrated with the approach presented in this paper by "wrapping" the functional code generated by them in components that can be deployed on the container generated by the approach here.

4.5 Model Driven Software Development

Model-Driven Software Development (MDSD) is concerned with generating complete applications from models. Those models can be anything that is useful to specify application functionality on an abstraction level higher than implementation code – optionally a domain-specific notation can be used. OMG's MDA [20] is a standard to use UML [22] for model-driven development. The approach presented in this paper uses model-driven techniques extensively. The models are specified in UML and other notations such as XML. While Model-Driven Software Development aims at (but does not require) the generation of the complete application including the behavior, we only generate infrastructure code here. Implementation of the core application logic is out of scope. For more information on MDSD see [32] and [33]

5. PRACTICAL EXPERIENCE

I have been part of several projects implementing component infrastructures for various domains (among others, automotive and mobile phones). Although I cannot provide details about these projects in this paper, the approach has proven *very* successful. Specifically, the tools that are required for the generative aspect of the approach are practically usable and easy to use. MDSD makes the concepts of components and communication middleware as explained in [30] and [33] applicable to the embedded domain.

Please contact the author in case you want to know details.

6. ACKNOWLEDGEMENTS

Several people provided valuable feedback on earlier versions of this paper: Frank Buschmann, Michael Englbrecht, Michael Kircher, Alexander Schmid and Uwe Zdun. Many thanks to all of them.

7. ABOUT THE AUTHOR

Markus Völter works as an independent consultant on software engineering and technology, focusing primarily on software architecture, middleware and model-driven software development. He has experiences in many different domains including health care, banking, astronomy, mobile system and automotive; over the last year, Markus has worked in the embedded domain, using a model-driven approach to implement component/container infrastructures in the automotive and mobile phone domains.

In addition, Markus is an active author and conference speaker. He is known as an authority on middleware and model-driven software development and regularly speaks on this topic at conferences such as OOP, JAOO, ECOOP or OOPSLA. Markus is the co-author of *Server Component Patterns* and *Remoting Patterns* pattern books as well as of the forthcoming dPunkt title on *Model-Driven Software Development*. Details can be found at www.voelter.de

8. REFERENCES

[1] AOPSYS, *Java Aspect Components*, http://jac.aopsys.com/

[2] AOSD steering committee, *Aspect-Oriented Software Development*, http://aosd.net

[3] ASN.1 consortium, *ASN.1 home page*, http://www.asn1.org/

[4] Sourceforge.net, *openArchitectureWare GeneratorFrameWork*, www.sourceforge.net/projects/architecturware

[5] Chung, T.M.; Dietz, *Static scheduling of hard real-time code with instruction-level timing accuracy*, http://www.computer.org/proceedings/rtcsa/7626/76260203abs.htm

[6] CiA, *Controller Area Network (CAN), an overview*, http://www.can-cia.de/can/

[7] Doug Schmidt, *The ACE ORB*, http://www.cs.wustl.edu/~schmidt/TAO.html

[8] Doug Schmidt, *The Adaptive Computing Environment*, http://www.cs.wustl.edu/~schmidt/ACE.html

[9] Eclipse.org, *The AspectJ project*, http://www.eclipse.org/aspectj/

[10] Ilogix, Inc., *Statemate MAGNUM*, http://www.ilogix.com/products/magnum/index.cfm

[11] JBoss.org, *The JBoss Application Server*, http://www.jboss.org/

[12] Markus Voelter, *A collection of Patterns for Program Generation*, http://www.voelter.de/data/pub/ProgramGeneration.pdf

[13] Mathworks, *Matlab / Simulink*, http://www.mathworks.com/products/tech_computing/

[14] Microsoft, *COM+ Specification*, http://www.microsoft.com/com/tech/COMPlus.asp

[15] Microsoft, *Windows CE*, http://www.microsoft.com/windows/embedded/ce.net/default.asp

[16] OMG, *CORBA*, http://www.corba.org/

[17] OMG, *CORBA and the CCM*, http://www.corba.org/

[18] OMG, *CORBA, XML and XMI® Resource Page*, http://www.omg.org/technology/xml/

[19] OMG, *Meta-Object Facility (MOF), version 1.4*, http://www.omg.org/technology/documents/formal/mof.htm

[20] OMG, *Model-Driven Architecture*, http://www.omg.org/mda

[21] OMG, *Minimum CORBA Specification*, http://doc.ece.uci.edu/CORBA/formal/02-08-01.pdf

[22] OMG, *UML Resource Page*, http://www.omg.org/uml/

[23] OSGi Consortium, *The Open Services Gateway Initiative*, http://www.osgi.org/

[24] QNX, *QNX Neutrino RTOS*, http://www.qnx.com/products/ps_neutrino/

[25] Realtime Development Corp, *Realtime defined*, http://www.realtimeonline.com/RealTimeDefined.htm

[26] Several, *OSEK/VDX*, http://www.osek-vdx.org/index.htm

[27] Several, *Simple Network Managment Protocol*, http://www2.rad.com/networks/1995/snmp/snmp.htm

[28] Sun Microsystems, *EJB Specification*, http://java.sun.com/products/ejb/

[29] Unisys, *Unisys website*, http://www.unisys.com

[30] Voelter, Schmid, Wolff, *Server Component Patterns - Component Infrastructures illustrated with EJB*, Wiley, 2002

[31] Windriver Software, *VxWorks*, http://www.windriver.com/products/vxworks5

[32] Markus Völter: MDSD Tutorial, http://www.voelter.de/services/mdsd-tutorial.html

[33] Voelter, Kircher, Zdun: *Remoting Patterns*, Wiley 2004

[34] Stahl, Voelter, Bettin: *Modellgetriebene Softwareentwicklung*, dPunkt 2005

[35] AUTOSAR consortium, AUTOSAR website, http://www.autosar.org