Using a product line for creating component systems

Tomáš Bureš\textsuperscript{1,2}, Petr Hnětynka\textsuperscript{2}, Michal Malohlava\textsuperscript{2}
\textsuperscript{1}Institute of Computer Science, Academy of Sciences of the Czech Republic
Pod Vodárenskou věží, Prague 8, 18000, Czech Republic
\textsuperscript{2}Department of Software Engineering, Faculty of Mathematics and Physics
Charles University, Malostranské náměstí 25, Prague 1, 11800, Czech Republic
\{bures,hnetynka,malohlava\}@dsrg.mff.cuni.cz

ABSTRACT

Component systems have become a wide-spread technology and found their place in several application domains. Each component system has its specifics and particularities that reflect its focus and the application domain it is intended for. Although important, the diversity of component systems leads to a number of problems including having different tools for each system, unnecessary duplication of functionality and problems with integration when several domains are to be targeted. Based on categorization of component application domains, we propose a “meta-component system”, which provides a software product line for creating custom component systems. We focus especially on the deployment and execution environment, which is where most diversities are found. We demonstrate the usage of the “meta-component system” and propose how it is to be realized by two core concepts of SOFA 2, namely connector generator and microcomponents.

Categories and Subject Descriptors
D.2.13 [Software Engineering]: Reusable Software—Domain engineering, reuse models

General Terms
Design

Keywords
Component systems, generative programming, product line engineering, runtime environment

1. INTRODUCTION

Component Based Software Engineering (CBSE) is a very effective and quite favourite development method, which has been adopted in many domains — ranging from enterprise applications (e.g. EJB [28]) to embedded systems (e.g. Koala [22]). The cornerstones of CBSE are treating components as independent basic building blocks and explicitly capturing inter-component interactions. The concrete semantics of components and their interactions are formally defined by a component model. The component model provides a design view of components. However in order to actually design, develop, deploy and execute a component application, the model has to be supplemented with various tools and runtime libraries. According to our experience, the following three more constituents can be typically found:

1. Design and development tools, which typically include a graphical editor of component architectures or a compiler/validator/processor of components architecture description language, a generator of component implementation skeletons, a repository for storing components, various analysis tools, etc.

2. Deployment tools, which typically include support for planning of component distribution, generating connectors for inter-component communication, transformation of design components to runtime components (e.g. in the case of embedded systems when design components are merged on code level and runtime components are produced), etc.

3. Execution environment and tools, which allow for instantiating and executing components. The execution environment typically provides a container in which components can run, including the runtime libraries used by components, it manages the lifecycle of components, etc.

The joint product of these four constituents (the component model and the three categories of tools) is typically referred to as a component system. There are currently a number of different component systems. They are typically tailored for some specific domain or use. For example in the enterprise domain, the stress lies more on a richer set of features (e.g. transactions, persistence, etc.), while in the domain of embedded systems, the driving concern is the predictability and low resource demands.

Although component systems tend to have significant differences in the component model, tools and execution environment, there are unifying concepts and features (e.g. encapsulation, reuse, composition) that all component systems share — in fact the possession of these features entitles a framework to be called a component system.

Moreover, as the use of components spreads and software becomes more pervasive, it is typical to find applications that combine even several component systems (e.g. one for user interface, one for implementing business logic and one for driving an embedded device controlled by the application). Such mixing of component systems means however combining different models, tools and execution environments, which is typically not very well-supported. Also in
fact it means duplicating the support for the common component concepts in each component system.

That leads to an idea of sharing selected common parts of component systems (e.g. model, component repository, deployment tools, runtime) or at least some generic elements which are adaptable for a particular use.

In this respect, our aim is to have a family of component systems or rather a “meta-component system” from which a component system for a particular domain or a combination of domains could be instantiated — addressing the needs and obeying the restrictions of the particular domain.

Compared to existing approaches (e.g. ACME [13] or xADL [12]), which focus only on component modelling, we put equal stress on modelling as well as on tool support and runtime environment, which in our experience is the key for the general use of component systems.

1.1 Goals

In this paper we aim at building a meta-component system, which shall serve as a software product line for creating customized component systems, each addressing a particular domain or a combination of them. Our intention is to adopt ideas of the product line engineering and generative software development (GSD) [11] with focusing on producing families of component systems for different target application domains rather then just for one domain. In more detail, the goals of the paper are the following:

1. To analyze existing component systems together with the domains, in which they are used. The analysis designates a scope which will be covered by the proposed meta-component system as well as it specifies common vocabulary (in words of GSD, the analysis constitutes a problem space). Based on results of the analysis we identify component systems’ characteristics, which in turn lead to variation points that our envisioned meta-component system has to accommodate.

2. To demonstrate how the meta-component system is to be used throughout the whole development life-cycle and how it eases the component development.

3. To discuss how to build the meta-component system and how to achieve its variability. In this sub-goal we focus on the last two constituents of a component system, namely deployment tools and execution environment and tools. This choice is motivated by the fact that in these two the biggest differences in current component systems can be found and as [18] states, the execution environment significantly influences component semantics. Additionally the variability of the component model part has been at least partially addressed by existing approaches [13, 12]. In terms of GSD we want to specify basic building elements of solution space as well as transformation process from the problem space to the solution space.

1.2 Structure of the text

The rest of the paper is organized as follows. Section 2 provides an analysis of the application domains and component systems used in them. Section 3 describes design of the meta-component system together with its lifecycle, while Section 4 presents realization of the proposed system in the scope of an existing component system. Section 5 presents related work and Section 6 concludes the paper.

2. COMPONENT MODEL DOMAINS

To better understand the commonalities and differences of the existing component systems, we have analyzed a diverse set of existing component systems [28, 22, 5, 23, 9, 15, 20, 21, 26, 17, 3] (we have focused only on component systems which support a complete application lifecycle, i.e. from design till execution). Also, we have taken into account our experience, which comes from the past decade of developing the SOFA component system [8].

The general analysis of the component systems has led us to identifying four main application domains (in other words, what is modeled using the components). These domains are (a) enterprise applications, (b) user interfaces, (c) configuration frameworks, and (d) embedded systems.

We elaborate on each domain separately in the rest of the section. For each domain we discuss a typical usage scenario of components and identify requirements and specifics of the domain. We relate these specifics to the component systems typically used in the domain and study what features the component systems in the domain typically posses.

2.1 Enterprise application domain

This domain represents a large number of applications, typically used in a commercial environment. An enterprise application often operates over a large amount of data stored in databases — it collects data from users, saves them into a database, does computation over stored data and presents the results. The application is typically divided into multiple layers where one layer is used for presenting information to the users (e.g. a thin client), next layer takes care of storing the data (e.g. into a database) and another encapsulates the business logic of the application.

A component system is often used in the business logic layer (e.g. EJB [28], CCM [21]), where the logic is divided into (often flat) components. Simple component systems can be also identified in a view layer (e.g. Java Server Faces [27]) where they are used to compose the user interface. The view layer however falls in the user interface domain as discussed in Section 2.2.

From the component system point of view, a component in the enterprise application is an encapsulated entity which communicates with other components through local interfaces or through remote interfaces, which allow distribution. The communication is typically implemented by a procedure call but messaging is also possible (e.g. via message sinks and sources in CCM, or via message-driven beans in EJB). Middleware (such as RMI or CORBA) is often employed.

A component is typically represented by a set of classes that are deployed (i.e. loaded and instantiated) in a container which contains an execution environment for components. In this sense a component is a well-identifiable unit even during runtime. It is further possible to deploy and undeploy a component basically independently.

The enterprise orientation of this application domain requires support of various services typically provided by the execution environment of the component system. These services typically include database access, persistence, transactions, naming, trading, web-service support, fault-tolerance (replication), etc. Each service has a standardized API defined by the component system (e.g. J2EE in EJB) and a reference to it is looked up by the component implementation using naming service. The binding of the services to particular entries in the naming registry can be typically
configured in the deployment descriptor of a component application.

In summary, the stress in the enterprise domain lies on having a rich set of services (database, transaction, persistence, etc.) that ease the development of enterprise applications. Components tend to be rather independent units existing at runtime. The composition of components (especially in the industrial component systems such as EJB or CCM) is typically quite simple, presumably owing to the fact that the component systems are usually flat. A lot of complexity in enterprise component systems is concentrated in the execution environment, which has to provide all the services and has to be able to manipulate components at runtime. To manage the complexity of the deployment and management of deployed components, graphical tools are often provided for this task.

2.2 User interface application domain

This application domain represents component systems which are used to build graphical-oriented user interfaces from predefined UI components (often called widgets). UI components are not limited to desktop applications (e.g. JavaBeans component system [26]), but they can be used to build web interfaces too (e.g. JSF [27]).

The UI components typically provide a standardized API that allows generic handling of them (methods as show, hide, paint, etc.). The relations among components are mainly of parent–child (corresponding to the graphical nesting of components) and publisher–subscriber (making possible to react and reflect on changes done in another component).

In this sense the component models are hierarchical and messaging (publish-subscribe) is used as the primary communication style. Interesting feature is that the messaging usually disregards parent component boundaries (i.e. it connects components that do not reside in the same parent component).

The composition of UI components is typically performed on running components, so as the result of the composition may be immediately visible (e.g. using the Bean Builder [25]). That requires components to be runtime entities, which are composable and replaceable at runtime.

Important features of UI components are introspection and persistence. The former allows querying the component to obtain a list of its attributes and messaging interfaces, which enables the UI design tools to compose and configure components in a generic way without having previous knowledge of them. The latter feature allows serializing a composition of configured components to a file and using the file later to instantiate the user interface at runtime.

In summary, UI component systems are relatively lightweight and do not require any elaborate services (as opposed to enterprise systems). The stress lies on easy and generic composition of running components. UI component systems are typically local (i.e. not distributed) with no or very simple deployment.

2.3 Configuration frameworks

Configuration frameworks are component systems typically used for configuring, customizing and extending an application. They span a wide range from simple plug-in frameworks or flat component systems (such as OSGi [23]) up to elaborate hierarchical component systems (such as Fractal [5] or OpenCOM [10]).

The common denominator of these systems is that components are well-defined runtime entities, which means that they can be instantiated and composed at runtime; basic introspection mechanism is also often present.

Typically, configuration frameworks do not rely on a rich execution environment. Rather, the services usually provided by the execution environment are modeled as application components, which often leads to the need of accessing the same “service” component from several other components. There are different ways of supporting this — in OSGi a component may expose part of its functionality as a service and other components may resolve a reference to the service dynamically at runtime, while Fractal has the concept of the shared component, which allows using the same instance of a component at several places in a nested architecture.

Component systems for the configuration are typically local, since for simple configurations the distribution is usually not required. If the distribution becomes an issue, it is realized by components wrapping middleware and acting as connectors. However, in configuration frameworks aiming at grid computing [3], the distribution is the core concept together with special multicast and gathercast interfaces.

As the configuration frameworks address customization and extensibility of an application, they also sometimes contain means allowing extensions of the component system itself. For example in Fractal this is supported by having the possibility of customizing component controllers, which act as management interfaces of a component. This way, it is easily possible to switch on/off several checks (e.g. blocking calls to a component that has not been started yet), alter existing functionality and implement new one (e.g. verifying that calls to a component follow a particular behavior protocol [1]).

In summary, the component systems for configurations tend to have a simple and quite flexible execution environment. The environment does not offer rich services itself, however these services may be introduced in the form of components as there is typically a way of viewing a component instance as a shared service. The stress lies on having a strong concept of a component that can be manipulated (instantiated, composed, bound, replaced, etc.) at runtime. The example of Fractal also shows that it is advantageous if component systems in this domain are hierarchical and extensible.

2.4 Embedded systems domain

This domain covers software for embedded systems — such as those found in cars, mobile phones, home appliances, industry automation, etc. This area is rather wide and the particular requirements of the embedded systems may quite differ, however the common requirements typically include:

- **efficient resource usage**: The devices running embedded systems are typically small with restrictions on power usage and overall cost (save for automation where this is typically not a problem). This means that the embedded system must perform correctly even with a slow CPU and little memory. As the result, the coupling between hardware and software is quite high.

- **high demands on dependability**: Embedded systems are often used for controlling safety critical tasks (e.g.
brakes in a car), where the cost of failure is very high (it can even cost human lives).

- **real-time:** The correct function of embedded systems often depends not just on the correct result of the computation but also on delivering the result in time. That classifies many of the embedded systems as soft real-time or even hard real-time.

Component systems in this domain (e.g. Koala [22], Pecos [20], Pin [15], ProCom [7], Robocop [17], AUTOSAR [2]) reflect the hardware limitations by assuming a thin execution environment, which provides only a basic abstraction layer between hardware, a real-time operating system (RTOS), and components. The execution environment typically offers only limited services such as a persistent storage [17].

In many cases components are transformed during deployment to RTOS concepts (tasks, processes) and they are statically linked with the execution environment. Because of the absence of runtime components, components are mostly composed and configured statically at deployment, which often includes creating a real-time schedule for component execution based on the data/control-flow in the component architecture [7]. Configuration at runtime is missing or is very restricted (e.g. only to setting attributes via a previously defined interface) [22]. The static configuration, however, allows for dramatic optimizations such as discarding unused components in an application [22].

The requirements on dependability put a lot of emphasis on analysis (such as of resource consumption, worst-case execution time, reliability). This concern is often reflected in the component systems by having a simple semantics for components, which are sometimes seen as being purely reactive (i.e. having no own thread of activity) [14, 7]. The communication is also kept simple by supporting the procedure call or the asynchronous message delivery. In some cases exogenous connectors are employed to explicitly capture the data and control flow [7]. To allow for analysis, component models often support annotations for expressing resource usage, RT requirements, reliability attributes, etc.

In summary, component systems for embedded devices have often no explicit execution environment running in a target device, but the environment is represented by a bundle of libraries statically linked to components during deployment. Important is the role of tools, which comprise a transformation of components to RTOS concepts and linking with an execution environment, simulator and so on. The stress further lies on analyzability and support for real-time.

### 3. META-COMPONENT SYSTEM

As shown in the previous section, there is a broad range of component systems targeting different domains. Although these systems differ in many aspects, they share a lot of commonalities regardless of their particular domain. The most significant commonalities include:

- All component systems have the concept of an encapsulated component that communicates with other components only via designated interfaces.

- The communication between components is typically realized by a procedure call or a kind of messaging. The communication is local or it is realized by a kind of middleware.

- Components may be composed horizontally by connecting their interfaces or vertically by the parent component–subcomponent relationship.

- Components themselves require a component container, which is an entity providing them various services (such as naming, transactions, persistence, etc.). The container is also responsible for managing the component lifecycle and managing connections among components.

- The container itself is controlled by a set of tools that allow configuring it, creating it, and destroying it. Also, there are often tools available for controlling components running in the container.

- The components are typically deployed according to a deployment plan that describes the allocation of components to the containers and concretizes the resources used by components. Tools for creating the deployment plan and for performing the actual deployment are typically utilized.

All the concepts and activities related to components are reified not only in the component model and the execution environment, but also in tools, which cover the whole development process and typically include an IDE for design and development, a repository for storing and reusing components, various deployment tools and tools for managing components at runtime. In fact, it is the tool support where common component system concepts are evident even more than in the component model.

In our approach we propose taking advantage of the commonalities among component systems and build a meta-component system (i.e. a kind of product line) for creating custom component systems. In this section we elaborate this idea in detail and show a typical use case of the meta-component system.

As an example, we consider development of an application for control and management of a manufacturing line. The application consists of two parts — an embedded system part controlling the move of the line and other actuators, and an enterprise system part providing an interface for retrieving the status of the line and history of the production and planning for new production. Development of such an application using the proposed meta-component system consists of the following steps:

1. **Gathering and analyzing the requirements of the application.** This step is the same as in the classical CBSE. In the case of the example it means identifying the need of transaction support, distribution, replication and monitoring in the enterprise part and support for real-time and emphasis on a low memory and CPU footprint in the embedded part.

2. **Configuring and generating the component system.** This step is new compared to the classical CBSE. The system architect specifies the requirements of the desired component system in a configuration tool of the meta-component system. The configuration tool uses common vocabulary specified by the analysis presented in Section 2 and permits selection and configuration of variation points. Based on a selected configuration the corresponding component system is generated/configured — that means the component model,
3. **Designing and developing the application.** This step is similar to the classical CBSE. The component developer uses the provided IDE and the models to design the application by components and to implement the components. In the case of our example, it means designing and developing components of the enterprise part and the embedded part. The component system contains support for interconnecting these two parts. The enterprise part is developed in Java, using JTA for transactions, RMI for distribution, etc. The embedded part is developed in C. Existing domain specific tools are utilized (e.g. simulator of an embedded device) — they are actually selected by the configuration tool.

4. **Application deployment.** This step is again similar to the classical CBSE, only the tools provided by the created component system are used. In many cases it means reusing existing tools, which were selected and pre-configured by the configuration tool. In the case of the example, the deployment comprises allocation of enterprise components to containers and generating connectors for their distribution. On the embedded side, the components are merged and a binary image is synthesized. The binary image contains the RTOS, basic API for the components and the actual components turned into tasks and processes of the RTOS. The image is uploaded to the target device using tools provided by the created component system. The communication between the enterprise part and the embedded part of the application happens through CAN-bus abstracted by special generated connectors.

5. **Monitoring and managing the application at runtime.** This step is again similar to the classical CBSE. Only the management and monitoring is performed using the tools provided by the generated component system. In the case of the example it means using a JMX console to access the enterprise components, which are automatically exposed by the pre-configured container via JMX. The tooling also contains support for starting, stopping and updating components.

In summary, the proposed meta-component system inserts the step 2 (configuring and generating the component system) into the classical CBSE process. The other steps remain the same or similar, only they utilize the component system (i.e. models, tools and execution environment) configured and generated at the step 2 (see Figure 1). An important feature is also the ability of reconfiguring the component system at any stage of the development lifecycle, when new requirements are discovered or some existing requirements are evaluated as not needed.

The benefits of using the meta-component system lie in reducing the duplication of functionality and providing a common approach to different component features and application domains. Another strong benefit is the automated evaluation of application’s requirements by the configuration tool, which guides the choice of used technologies by ruling out those which do not satisfy the requirements. The ability of customizing the component system also reduces the footprint of the unused features, which would be present in the component system otherwise. Finally, the fact that the generated component system provides support only for the required features makes design and development easier and more straightforward as the developer has available only the features required for building the application.

The production of customized component systems is also
very interesting for cross-domain applications (as shown in the example). In this respect, the meta-component system instantiates an interrelated family of component systems (e.g. having one model for enterprise systems and one model for embedded systems). The tooling provided in this case supports the whole family, thus making possible to design, develop and deploy components in the same way (preferably through a single IDE).

4. REALIZATION OF DEPLOYMENT AND EXECUTION ENVIRONMENT

The previous section analyzes the commonalities of component systems, proposes the meta-component system and demonstrates its utilization. Although the many commonalities, the meta-component system has to accommodate also the differences that stem from different domain requirements. In this section we focus on the differences and show how they can be addressed in deployment tools and the execution environment. When inspecting them closer, it is possible to identify the following areas of variations.

- **Component semantics** — it is defined by the component model and comprises decisions such as whether components are flat or hierarchical, what are the possible connections and if they can be distributed, whether shared components are permitted, what is the component lifecycle, etc.

- **Target platform** — it influences the choice of underlying technologies used by the execution environment and it defines activities to be performed in the deployment — e.g. generation of connectors, transforming components of an embedded application to RTOS tasks and processes, etc.

- **Services required by components** — they comprise the services required for a proper functioning of the components, such as a transaction support, a persistence, a database access, etc.

Thus, with respect to the target domain and specified requirements, the execution environment has to support different functionality and services. The deployment is also influenced by the domain and requirements as it is totally dependent on the execution environment (e.g. deployment of enterprise applications consists of dynamic uploading of code to a container, while in embedded applications the container’s functionality must be merged with the application).

In our proposed approach, we achieve the necessary level of variability of the deployment process and execution environment by composing them from components. We demonstrate this on the SOFA 2 component system and in the rest of this section we show how SOFA’s deployment and runtime environment can be extended to become fully configurable.

SOFA 2 [8] (shortly SOFA in the following text) is a hierarchical component system. In addition to component model, SOFA comes with a set of deployment tools and an elaborate runtime environment, which is responsible for component instantiation, life-cycle management, distribution of a using software connectors, dynamic update, etc. SOFA contains two concepts which are very important for the variability and configurability of the deployment process and execution environment. Although they do not directly implement all the envisioned variability yet, they serve as the proof of the concept and represent a viable way of achieving the variability. These two concepts are the *connector generator* and *microcomponents*.

The connector generator [6] is responsible for creating connectors, which are entities realizing the communication among components. The connector generator takes on the input a declarative description of communication requirements — component interfaces, a communication style and required non-functional properties (e.g. security, monitoring); and based on these it creates the requested connector.

Internally, the connector generator contains two main components — an *architecture resolver* and a *code generator*. The architecture resolver uses constraint solving techniques (based on Prolog) to derive a connector architecture that satisfies specified requirements. The connector architecture is built out of hierarchical connector components (so called *connector elements*). In addition to composing and connecting the connector elements, the architecture resolver also parametrizes each element in the architecture (e.g. adapting an interface, setting attributes). The code generator follows the resolved architecture and builds the connector implementation.

From the point of view of the meta-component system, the connector generator represents a skeleton of the configurable deployment tool. It is itself modular — the knowledge about the particular deployment and runtime environment is introduced in the form of available connector elements and actions, which are plug-ins that take care of source code adaptations, compilation, generating middleware stub and skeletons, packaging, etc. Thus by extending the connector generator and configuring its parts, it is possible to achieve simple packaging of components (e.g. in configuration frameworks) as well as elaborate transformation of components to RTOS concepts (e.g. in embedded systems). The Prolog-based constraint solver in the connector generator additionally provides a ground for planning the transformation and optimizations.

Microcomponents [19] are used to provide runtime variability with respect to component semantics and services. The idea behind microcomponents is to divide an application component into the *control part* and the *content*. While the content is the business code written by the application developer, the control part is provided by the component runtime. In fact the control part wraps the *content* and exhibits component interfaces. In addition to business interfaces, the *control part* also implements control interfaces (e.g. lifecycle interface, lookup interface) that allow the execution environment to manage the component.

The innovative approach of SOFA is that the control part of a component is modeled using the *microcomponents*, which are flat local components with a simple lifecycle. Each microcomponent is responsible for a specific control functionality (e.g. looking up an interface, or blocking calls to a business interface when the component is stopped).

Since a particular component feature (e.g. component lifecycle) is realized by a number of microcomponents (e.g. one implementing the start/stop method, others attached to business interfaces to block incoming calls when the state is "stopped"), the microcomponents are grouped to *agents*. Each aspect defines a consistent extension of the control part introducing a particular feature. Thus, the runtime component semantics and services are configured by a selection of
aspects to be applied to the application components (or to a subset of them).

Microcomponents share a lot of commonalities with connector elements, which makes natural and advantageous to unify them in one common concept that may be used both during deployment and runtime. Thus, coming out from the two concepts of the connector generator and microcomponents, we propose the configurable deployment process and execution environment to be based on two main constituents — a synthetizer and a microcomponent system (see Figure 2).

The synthetizer is a part of deployment tools responsible for transforming business components to runtime artifacts. It is built on the SOFA connector generator and it allows adapting, building and assembling microcomponents (see Figure 2), which represent various parts of the execution environment, connector functionality, and business code of application components. The output of the synthetizer is either a microcomponent architecture (together with microcomponents) that is to be instantiated, or it is a binary image to be uploaded to an embedded device and executed there. In the former case, the microcomponent architecture produced by the synthetizer contains only the business code of application components merged with control parts and connectors. These microcomponents are to be deployed in the execution environment, which consists of a minimalist core of the microcomponent system and preconfigured microcomponents providing remaining services (e.g. naming, transactions, remote interface to component deployment, etc.). In the latter case the produced binary images contain not only microcomponents for the business code, control parts and connectors, but also the microcomponents implementing the services of the execution environment.

With regard to the current status of SOFA, we have to enhance the connector generator to use the unified concept of connector elements and microcomponents and to handle whole application components, not just connectors. Further, we have to advance in componentizing the deployment container with the execution environment, i.e. the container in which components are executed. Currently, SOFA provides only a single type of container, suitable for rather large-scale applications. To follow the meta-component system concepts, the container needs to be reimplemented also as a set of microcomponents that can be configured together to provide a required functionality. Finally, with all infrastructure prepared, it is necessary to provide a configuration tool, which allows the configuration of the synthetizer and the execution environment.

5. RELATED WORK

In the area of CBSE, as far as we know, there is no approach covering the complete application lifecycle that intends to serve as a meta-component system for instantiating custom component systems. But in a wider scope, projects with many similarities can be found.

The first group of projects concerns reflective middlewares. The main idea behind them is to provide a highly configurable middleware layer which would reflect (statically or dynamically) requirements of an application built on the top of this layer. To build such a layer, projects typically utilize some ideas of component systems that ensure the demanded adaptability and dynamic reconfiguration.

A well-known member of the reflective middleware family is OpenORB [4]. It assembles a middleware layer with the help of OpenCOM [10] component system and it focuses on adaptation and runtime reflection. OpenORB provides multiple meta-layers for describing interfaces, architectures and resources. These layers are fully configurable statically as well as dynamically (at runtime).

Another example of the reflective middleware is the Dream [16], which is built using the Fractal component system [5]. Dream allows building middleware that offers messaging, scheduling, task creation, etc. For adaptation and runtime reconfiguration it uses the principle of controllers provided by the Fractal component model.

The mentioned examples partially realize the idea presented in this paper, however they focus on a single domain only and allow configuration just in the scope of their domain. Also, they aim at adaptation and dynamism of resulting systems and therefore they do not consider requirements of embedded systems for static configuration or efficient use of system resources.

Another group of projects related to our approach are configurable enterprise platforms. Probably the most popular representative of this group, widely used in industry, is Spring [24]. It is primarily aimed at enterprise applications but it can be used in other application domains as well (e.g. desktop applications — the Spring Richclient project).

Spring employs the concept of a configurable runtime which is based on a lightweight container used as an execution environment for application components. The container configuration is based on a declarative XML descriptor and annotations in code. The advantage of Spring is that it is straightforward and allows easy configuration of different runtime environments. On the other hand, Spring is closely coupled with J2EE platform and, hence, its configuration possibilities are restricted to J2EE.

The last group of projects being related to our approach is constituted by extensible architecture description languages (such as ACME [13] and zADL [12]). ACME allows for describing a system as a composition of components and connectors, where each part of the system can have associated properties (e.g. configuration values for components or a communication style for connectors). To support families of architecture ACME introduces element types describing prototypes which can be instantiated and separately con-

---

Figure 2: Deployment and execution environment

---

<table>
<thead>
<tr>
<th>Component System</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOFA</td>
<td>Connector generator and microcomponent system</td>
</tr>
<tr>
<td>ACME</td>
<td>Architecture component model</td>
</tr>
<tr>
<td>OpenORB</td>
<td>Middleware layer for adaptation and runtime reflection</td>
</tr>
<tr>
<td>Dream</td>
<td>Middleware for messaging, scheduling, task creation</td>
</tr>
<tr>
<td>Spring</td>
<td>Configurable enterprise platform</td>
</tr>
</tbody>
</table>

---

**Figure 2**: Deployment and execution environment.
figured in different architectures. xADL is in many ways similar to ACME description language but it also includes multiple extensions for defining architecture product lines or to support linking architecture elements with Java artifacts.

In summary, the architecture description languages describe component-based applications in a generic way, which is necessary for modeling applications from different domains. However, they do not deal with deployment or execution environment, which makes them relevant only to modeling part of our approach. In fact reusing their concepts for the component model in our work seems as a viable idea.

6. CONCLUSION

In this paper we have presented an approach of instantiating component systems for different domains using a common meta-component system. We have shown the usage of the meta-component system, which basically follows the classical CBSE process, only inserts a component system configuration step at the beginning (after collecting requirements). Based on identifying typical application domains of component systems and discussion of their characteristics, we have proposed how to achieve the configurability of the deployment process and the execution environment. Our solution is based on two distinct concepts of SOFA 2 component system, namely on the connector generator and microcomponents.

7. ACKNOWLEDGMENTS

This work was partially supported by the Czech Academy of Sciences project 1ET400300504.

8. REFERENCES