An extensible architecture-based framework for coordination languages

Torsten Fink
Institute for Computer Science
Freie Universität Berlin
Berlin, Germany
tnfink@inf.fu-berlin.de

Karsten Otto
Institute for Computer Science
Freie Universität Berlin
Berlin, Germany
otto@inf.fu-berlin.de

ABSTRACT

The dynamic and heterogeneous nature of distributed systems make the development of distributed applications a difficult task. Various tools, such as middleware systems, component systems, and coordination languages, offer support to the application developer at different levels.

There are several coordination systems that integrate such tools into a complete environment to build applications from heterogeneous components. To achieve extensibility they usually have a layered architecture: An application is first mapped to a middle layer and then to a target system. But this approach hides the specific features of a target system from the developer, as they are not represented in the middle layer, and often induces additional run-time overhead.

In this paper we introduce the extensible coordination framework ECF that allows developers to build efficient distributed applications which exploit the specific features of the target systems. Support for target systems and application domains are encapsulated by extension modules. Modules can be built on top of other modules to support refined functionality.

Keywords

coordination language, distributed systems, component technology, developer framework, software architectures

1. INTRODUCTION

Despite the great effort invested in the creation of frameworks and tools, the implementation of distributed applications is still much harder in comparison to centralized applications. Specialized categories of tools have emerged to support the application developer at different levels.

Middleware systems provide abstractions from the technical details of network programming. They also offer basic services for communication and coordination.

Component systems allow the developer to wrap pieces of executable centralized code as components in a uniform way. Additionally they often support management (deployment, updating, removal) of components.

Coordination languages finally build distributed applications from centralized components. The developer provides glue code as a program in the coordination language that describes the interaction of the components.

A complete application development environment has to integrate tools of all these three categories. Because the coordination language represents the highest level of control we call such a development environment plus its runtime system a *coordination system*. There are several approaches to how a coordination system integrates different tools.

Stand-alone tools. Instead of using a special coordination system a developer can choose the tools he needs and uses each independently to build his applications. For example, he uses C++ to implement client and servers, Mico to connect these to CORBA, Python to implement some set-up scripts for the system, and finally SSH for remote set-up.

The problem with this approach is that these tools usually have strong dependencies. Component systems are often specific to exactly one middleware system. For example Enterprise Java Beans is built on Java RMI. Also, coordination languages often depend on a specific component or middleware system.

Because of the variety of tools in each category the developer has to integrate all of them manually. In that process he often has to resolve the interaction of incompatible tools. For this purpose he has to gather detailed knowledge of their design and of their implementation. These acquired skills are hard to transfer, therefore this approach requires a steep learning curve for application developers in general.

Specialized systems. The naive approach now would be to design a set of specialized coordination systems. Each could provide the coordination needs of a particular application domain, and also provide a mapping to a concrete middleware and component system. Such a coordination system would be customized to the application developers needs. Thus it can provide a comfortable environment and it can create efficient applications which utilize the special features of all tools. On the other hand it induces the most effort on the developer of the coordination system itself.

On the long run this approach is not viable for a number of reasons. It is nearly impossible to anticipate and design for all possible uses of a system. Once it is complete, it can only

SAC 2003 Melbourne, Florida USA

fulfill a specific set of tasks. Should different requirements arise later, the necessary changes affect the entire system, up to a point where a complete redesign is needed.

For the same reason, it is also hard to reuse parts of one coordination system in another system. This is especially true for the employed coordination paradigm. Consider that you have a coordination system which supports task-graph like CORBA applications and you want to create another coordination system with in principle the same coordination language but for SOAP. It would decrease the development effort of the latter coordination system significantly if you could reuse parts of the former system. Finally, current middleware systems evolve rapidly, and every change would deeply affect all dependent coordination systems.

Extensible layered system. To cope with these problems a layered system architecture is used. The top layer consists of an extensible coordination language, which can be extended by new elements to support different application domains. This top layer language is translated to the middle layer, usually a generic calculus or automaton based language. The bottom layer consists of multiple back-ends for all supported middleware platforms. Each back end is able to map the middle layer language to a concrete system.

An example for this approach is the Manifold coordination system [9]. In the coordination language Manifold an application consists of processes, specified as transition systems, that interact via events. An application is compiled to a middle layer called IWIM (Ideal Worker Ideal Manager). Different implementations of IWIM provide different back-ends.

Because the Manifold system uses the event paradigm it is well suited for all control-oriented applications. The disadvantage of choosing such a generic approach is that it needs some effort to learn it and it may lead to complex implementations even if the application has a simple structure.

Additionally the event paradigm is not well suited for declarative programming. An application could consist of some components and a set of rules which specifies dependencies on and between the components. At run-time the run-time system would use the rules to set up the application. Such an approach would be hard to do in Manifold.

Due to the intermediate compilation stage of the middle layer, the used middleware is opaque. Thus it is not possible to directly represent middleware specific features, such as the more sophisticated features of CORBA as extension to the top layer language. Any advantages of a specific middleware system are unavailable to the application developer.

Furthermore the middle layer often adds complexity which is not necessary for certain application domains, and in fact even reduces the overall performance. For example, if an application was built with the declarative configuration paradigm, the coordination system only needs to deploy the components and start them so that they act as a distributed application. No further runtime overhead is necessary.

Thin syntactical middle-layer. Our approach, the Extensible Coordination Framework (ECF) also uses a layered design but with a very thin middle layer that only provides a common syntax for applications. This syntax is based on the architecture description language Acme [3]. The semantics is added to the framework by extension modules which provide support for different middleware and component systems as well as different coordination paradigms.

The modular design allows us to combine modules in arbitrary ways, and thus leverages reuse. In addition, modules can be built on top of existing modules. This allows the capture of a pure coordination paradigm in an abstract module, which can later be extended to fit a concrete system.

In section 2 we give a short overview of our system introducing the core concepts. Then in section 3 we describe our framework in more detail on the basis of a small example. After a comparison with related work in section 4 we conclude finally in section 5 and outline future work.

2. A QUICK INTRODUCTION TO ECF

In our *Extensible Coordination Framework* (ECF) an application developer builds the application as a graph, using the features of *extension modules*. The graph is compiled to an executable that can be run in a specific target system.

Figure 1 gives an overview of ECF. As syntactical basis for the application graphs we chose the architecture description language Acme [3]. It distinguishes between components and connector nodes, whose type definitions are called a *vocabulary*. The vocabulary forms the core of an extension module, and is realized by Java classes which are used throughout the development cycle.

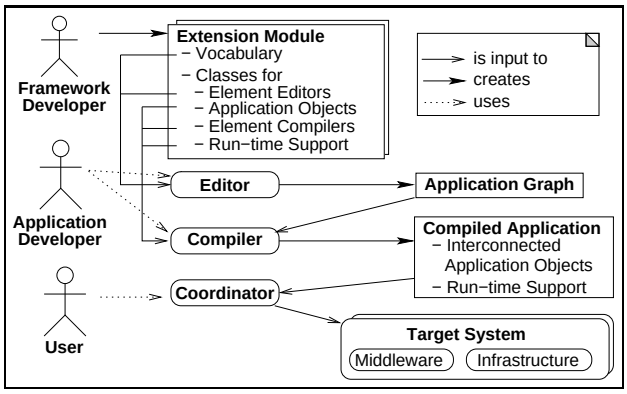

Figure 1: Overview of the ECF system

Editor classes are plugged into the main ECF editor, and can be used to configure the properties of specific elements at design time. *Application object* and *element compiler* classes add semantics to the elements of the vocabulary: An application object is part of the compiled application and performs some functionality at runtime. An element compiler maps the nodes of an application graph to application objects during compilation.

This approach of defining the semantics is very generic. An application object from one extension module can perform a binding to a remote object whereas an application object from another extension module can compute a numerically intensive algorithm. This implies that there cannot be a single mechanism to manage and activate all application objects of a compiled application. Instead, instances of specific *Runtime support* classes are used, which also provide interfaces to access the target system.

Extension modules can be built upon other extension modules. They then can subclass elements of the vocabularies and other editor, compiler, or run-time classes. This ap-

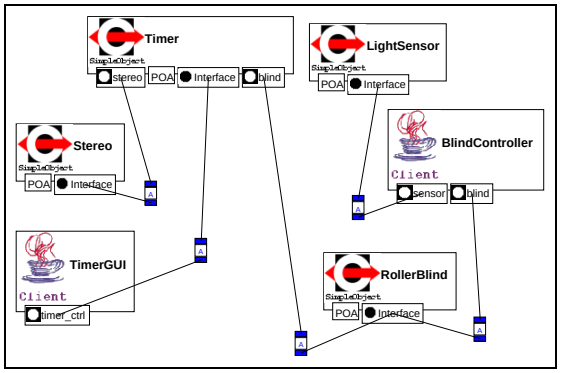

Figure 2: The configuration of the home system

proach enables the definition of abstract extension modules for generic coordination paradigms. For example, an extension module can provide a generic vocabulary to build task graphs without support for specific tasks. Then other modules provide tasks for specific target systems.

At the beginnig of the development cycle an application developer chooses the extension modules she needs for her application. She then builds the application graph from the vocabularies of the modules. Finally she compiles the application to a set of application objects and a set of objects for run-time support.

The application is executed by a coordinator. It utilizes the run-time support objects to execute the application objects and to interact with the target systems. The following section describes these development phases in more detail accompanied with an example application.

3. USING THE FRAMEWORK

To demonstrate application development with ECF, we consider a simple home automation example. The customer already had some home appliances installed: electronic roller blinds, a light sensor, a timer, and a stereo. The devices are attached to a network, and each can be controlled remotely via a CORBA interface. The vendor also provided an ECF extension module, based on the CORBA module. It supports the configuration paradigm for distributed CORBA objects, i.e. it contains a vocabulary with elements to specify the set-up and locating of CORBA objects, to define dependencies such as client/server or colocation, and to specify their resolving. Figure 2 shows the resulting application graph presented in the ECF editor.

The home automation module consists of component types for each device, which are simple extensions of the generic `SimpleObject` type from the CORBA module. This type provides access to CORBA interfaces, and configuration properties such as a CORBA naming service name. The vendor did not need to provide any additional CORBA-specific code to use the naming service.

The home automation module also contains some utilities: a controller type that translates the analogous measurements of the light sensor to a control signal for the roller blinds, and a graphical user interface component type to manipulate the timer. The vendor built them by extending the generic `Client` type from the CORBA module with custom code to control the hardware devices. Finally, the module does not define its own connector types, but reuses the generic connector types from the CORBA module.

The vendor used this module to configure a home system for the customer, by placing appropriate elements for each device and utility, using their corresponding property editors (not shown) to configure them, and connecting them to resolve their dependencies.

For example, consider the Stereo and Timer types shown in figure 2. The Stereo type provides a facet port labelled `Interface`, which originates from the generic CORBA type, but is restricted by the Stereo type to represent the Stereos CORBA interface, with operations to turn the stereo on and off, play a song, etc. The Timer type has a receptacle port labelled `stereo`, which represents its need to access another component type which provides a certain facet. It is restricted the Stereos CORBA interface in this case. The vendor connected the two ports with an association connector. At run-time this dependency is resolved by first creating both objects and then by calling the method `setStereo()` with a reference to the CORBA-object that controls the stereo. This method name is given in a property of the receptacle port.

The same happens for all other components, resulting in the finished home automation system which can be deployed at the customers home. The port `POA` enables the usage of specific CORBA features. We describe the CORBA extension module in more detail in [2].

Now, the customer would like to add an evening entertainment program for her automated home. She wants this program to close the roller blinds, get the current music top-ten charts from the World Wide Web, and play them on her stereo. The customer hires a developer to create the desired entertainment program.

3.1 Design

The developer starts designing the new application by selecting appropriate ECF extension modules, see figure 3-I. The existing system is based on a configuration paradigm, the declarative specification of inter-component dependencies, which is clearly not suitable for the entertainment program. Instead a task graph paradigm seems more appropriate, so the developer selects the corresponding extension module `Workflow` to coordinate the application. As the home appliances use CORBA for communication, the developer also needs the extension module `CORBA`. Finally, to access the music top-ten charts, the developer adds an extension module `WebServices`. Together, these extension modules provide the vocabulary for the new application.

I) Creation **II) Editing of an application**

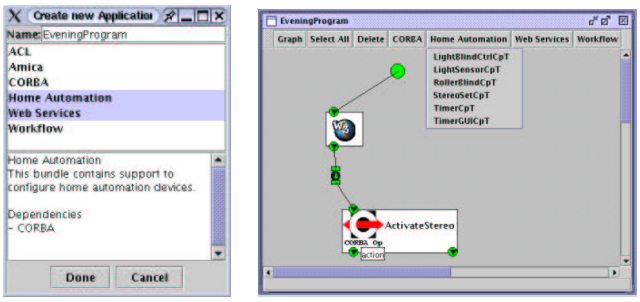

Figure 3: Creation and editing of an application

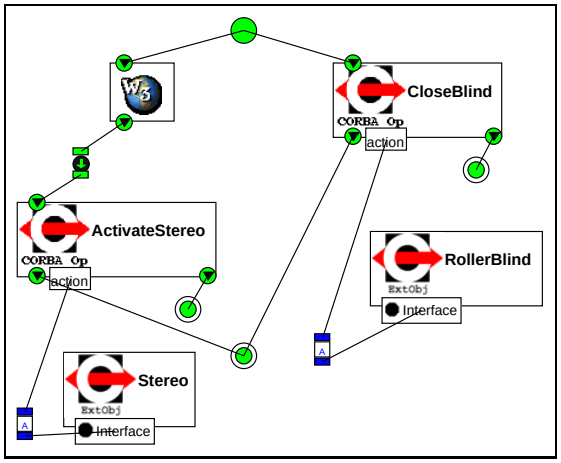

Figure 4: A program to set the home system

The developer uses the graphical editor shown in Figure 3-II to design the entertainment program. She chooses a vocabulary element from the menus to create an instance node in the application graph, sets its properties, and attaches it to other element nodes. Figure 4 shows the final result.

The application needs to control the installed roller blinds and stereo elements from the home automation system, so the developer adds two external object reference (`ExtObj`) elements from the CORBA module. She then configures them to resolve to the existing elements, by setting a configuration property for the CORBA naming service name of the referenced object. The developer finds the correct names by reviewing the existing home system design. To actually access the interfaces of these devices, CORBA operations must be invoked on them, so the developer adds two `CORBA Op` elements from the CORBA module, which represent such invocations, configures them for the correct operation to call, and names them `CloseBlind` and `ActivateStereo` respectively. Then she associates them with the corresponding `RollerBlind` and `Stereo ExtObj` elements. Furthermore, the developer adds a Web services element, and configures it with an URL to retrieve the top-ten music charts.

Now it is time to build the application graph. The developer first places a Start element from the task graph module, and connects it to the Web services and `CloseBlind` elements. This way the application will concurrently close the blinds and get the charts. The developer then places the main End element, and connects the `CloseBlind` and `ActivateStereo` elements to it. This means that after both invocations return, the application will terminate. Note that in our case the methods are designed to block, and will delay termination until their task is finished. The invocations might also raise exceptions, which should terminate the application too, so the developer connects them to two additional End elements. Finally, the developer adds a special connector that passes the retrieved charts from the Web services element to the `ActivateStereo` operation element.

3.2 Compilation

Having finished the design, the developer starts the compilation process. In the first stage, the ECF needs to find an appropriate element compiler for every element used in the design, which can be difficult: An element type from one extension module may simply be a restricted sub-type of a more generic type from another module. It will not define its own compiler, but rely on its parent elements compiler instead. Also, an extension module may be purely abstract, and may not provide compilers for its elements at all. Accordingly, the ECF needs to find for each element type the most derived type in its inheritance hierarchy that actually provides a compiler. For that purpose, it queries all participating extension modules in the inverse order of their dependencies.

In the second stage, the ECF uses the found element compilers to create application objects for their elements. Note that multiple application objects may be created to represent a single application element. For example, the End element in a task graph is represented by two objects: a synchronization node to collect all incoming triggers, and the actual end node. Such objects are already connected by the compiler. In this stage any recursive elements are resolved too: They are replaced by their internal graph, and their external connection points are mapped to the corresponding internal ones. In the third stage, the ECF connects ("wires") the created application objects according to the connection information from the design graph.

In the fourth and final stage, the ECF queries all participating extension modules for any necessary runtime support objects. It then combines these with the wired application objects into the finished application. Finally, the ECF serializes the application and stores it for later deployment and execution.

3.3 Execution

The customer deploys the evening entertainment application into her home automation system, and tells the ECF coordinator, which is part of her system, to run this new program. The coordinator deserializes the application, and tells the contained runtime support objects to start it.

The runtime support for CORBA resolves the `RollerBlind` and `Stereo` external objects to reference the actual objects in the home automation system. The extension module of the Web service object did not contain any runtime support, so no actions need to be taken. The task graph runtime support is more interesting: It finds the Start object, and triggers it.

The Start object triggers the `CloseBlind` object, which invokes an appropriate operation on the `RollerBlind` CORBA interface. Once the blinds are closed the `CloseBlind` object triggers the main End object. Concurrently, the Start object triggers the Web service object. It retrieves the current top-ten charts from its configured URL, and passes the result to the `ActivateStereo` object, which in turn invokes an appropriate operation on the Stereo CORBA interface. Once the stereo finished playing the charts the `ActivateStereo` object triggers the main End object too. If any of the CORBA invocations raises an exception, they trigger their extra End objects instead.

After an End object has been triggered from all its incoming connections, it informs the runtime that this task is finished. The runtime has no more tasks to run, so it tells the coordinator that it is finished. The coordinator notices that all its run-times have finished their jobs, so it terminates the application.

4. RELATED WORK

The main objective in the design of the imperative programming language Lua was extensibility [4]. Lua consists of a small core which is extendable with external libraries and fall-back functions. An example of the usage of Lua for distributed application is LuaSpace [1] which allows developers to build applications by configuring CORBA objects.

Although Lua is highly extensible it is still restricted to the imperative paradigm. But it provides a good basis to implement more abstract and sophisticated coordination systems. In the system ZCL, for example, it is possible to describe configuration and reconfiguration of CORBA objects as a software architecture [10]. The behavior of the components are specified in Z. A ZCL application is compiled to Lua and executed.

One of the essentials of ECF is to use an architecture description language (ADL) as a purely syntactical middle layer for a coordination system. This was inspired by previous systems. UniCon is based on typed components and connectors which can be hierarchically composed [11]. Applications are specified as software architectures and compiled for the target system. This approach was tested on several target systems, but for each system the compiler and the run-time environment was built in an ad-hoc manner. It was also impossible to mix sets of element types for different target systems or to reuse sets to build a new set.

Other approaches based on ADLs added semantic to their middle layer. The ADL Darwin is based on the π-calculus and allows the developer to specify dependencies between components [6]. Regis is extending Darwin with the possibility to add functionality in the form of C++ code [7]. In [8] Darwin is used to specify CORBA-based applications. The formal foundation of Darwin allows the developer to analyze the behavior of an application but it also restricts its extensibility.

The system Olan uses an ADL to specify applications which are build from arbitrary components for arbitrary middleware systems [5]. In contrast to ECF Olan uses a thick middle layer. The interfaces of a component are described in the Olan Interface Language (OIL). Components are specified in a uniform manner to enable the runtime system to automatically distribute and deploy them. Olan has a fixed set of connector types that must be implemented for every supported middleware system.

The usage of Olan allows the application developer to build distributed applications from heterogeneous components on several middleware systems. New component types or middleware systems can be integrated with acceptable effort by adding some plug-ins to the middle layer. The interaction of components is restricted by the set of connector types and the overhead of the run-time system is significant.

ECF provides a more generic approach because it does not force any semantic constraints on the supported target systems and application domains. Every extension module is free to provide every kind of functionality. On the other hand, because of the lack of a general semantic basis for components in ECF the compatibility of components is not guaranteed. It is possible in ECF to create an application from incompatible extension modules in which not all components can interact. For example a CORBA object cannot automatically access a SOAP server just because they are used in the same application. This problem can be solved by another extension that provides a mediator element.

5. CONCLUSIONS AND FUTURE WORK

In this paper we introduced the Extensible Coordination Framework ECF. It offers a uniform view on application design, by allowing developers to create distributed applications as software architectures in a graphical editor. However this abstraction is not reflected by a complex runtime support avoiding unnecessary runtime overhead. Instead we use a specifically tailored runtime support that even offers access to special functionality of the employed middleware systems. This is achieved by the extensible framework, which encapsulates support for different target systems and application domains in extension modules. Such modules can reuse other modules to provide more sophisticated functionality.

In the future we want to investigate the problems of incompatible extension modules. There are two approaches: Either a mediator is needed to bridge the incompatibility gap between interacting elements, or these elements must have a common foundation. Currently it is not clear which approach is superior. We also want to gather more experience in the use of ECF for larger applications. Finally, we want to compare the efficiency of applications built with ECF with the usage of other systems.

6. REFERENCES

[1] T. Batista and N. Rodriguez. Dynamic reconfiguration of component-based applications. In *5th International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 32–39. IEEE, June 2000.

[2] T. Fink, J. Schröter, and K. Otto. An architecture based configuration system for CORBA based applications. submitted to TACAS 2003.

[3] D. Garlan, R.T. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, 1997.

[4] R. Ierusalimschy, L.H. de Figueiredo, and W. Celes. Lua-an extensible extension language. *Software: Practice & Experience*, 26(6):635–652, 1996.

[5] V. Issarny, L. Bellissard, M. Reveill, and A. Zarras. Component-based programming of distributed applications. In *Recent Advances in Distributed Systems*, pages 327–353, 2000.

[6] J. Magee, N. Dulay, and J. Eisenbach, S. Kramer. Specifying distributed software architecture. In *ESEC*, 1995.

[7] J. Magee, N. Dulay, and J. Kramer. Regis: A constructive development environment for distributed programs. *Distributed Systems Engineering Journal*, 1(5):304–312, September 1994.

[8] J. Magee, A. Tseng, and J. Kramer. Composing distributed objects in CORBA. In *ISADS*, pages 257–263. IEEE, 1997.

[9] G. Papadopoulos. Distributed and parallel systems engineering in manifold. *Parallel Computing*, 24(7):1137–1160, 1998.

[10] V.C.C. de Paula and T.V. Batista. Mapping an ADL to a component-based application development environment. In *FASE 2002*, pages 128–142, 2002.

[11] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE-TSE*, 21(4), 1995.