Hardware configuration framework for the IBM eServer z900

by A. Bieswanger
F. Hardt
A. Kreissig
H. Osterndorf
G. Stark
H. Weber

This paper describes the concept, architecture, and implementation of the hardware configuration module within the support element of the IBM eServer z900. For the z900 project, this base system firmware component has been redesigned to obtain a software structure with a clear, simple, and scalable architecture that is suitable for future extensions to the z900. To achieve the desired flexibility, an object-oriented framework has been developed which supports the auto-sensing and configuration of hardware components as well as their status representation and management. The new configuration concept is based on a rule approach in which, for each sensed physical part in the system, a configuration rule specifies the object hierarchy to be instantiated upon it, including attributes and interconnections to other system parts. Furthermore, the concept and architecture of the framework are built upon a hardware object model (HOM) that has been designed to allow for further integration of key business logic of the z900 service subsystem and hardware support code.

1. Introduction and motivation

The z900 server is a strategic IBM product for the high-end market, specifically for e-business solutions. Because of the importance of the z900 project and the growth expectations of this market, it was decided in the early concept phase to redesign major parts of the pre-z900 service subsystem using modern software engineering methods and open standards with special emphasis on the use of object-oriented techniques in analysis, design, and implementation.

The pre-z900 service subsystem had been implemented primarily in a functional fashion, with multiple functions typically accessing global data. In an object-oriented approach, since only the appropriate methods of an object are allowed to access its data, the design principle of information hiding can be more readily achieved. In addition, one of the key advantages of an object-oriented design approach is typically an improved modularity of the software structure, which results in better code reusability and maintainability.

The zSeries z900 service subsystem contains the following three major components (**Figure 1**): the hardware management console (HMC), the support element (SE), and the cage controllers (CCs) with the service network. Besides introducing the cage-controller and service network [1], it was planned as part of the z900

0018-8646/02/$5.00 © 2002 IBM

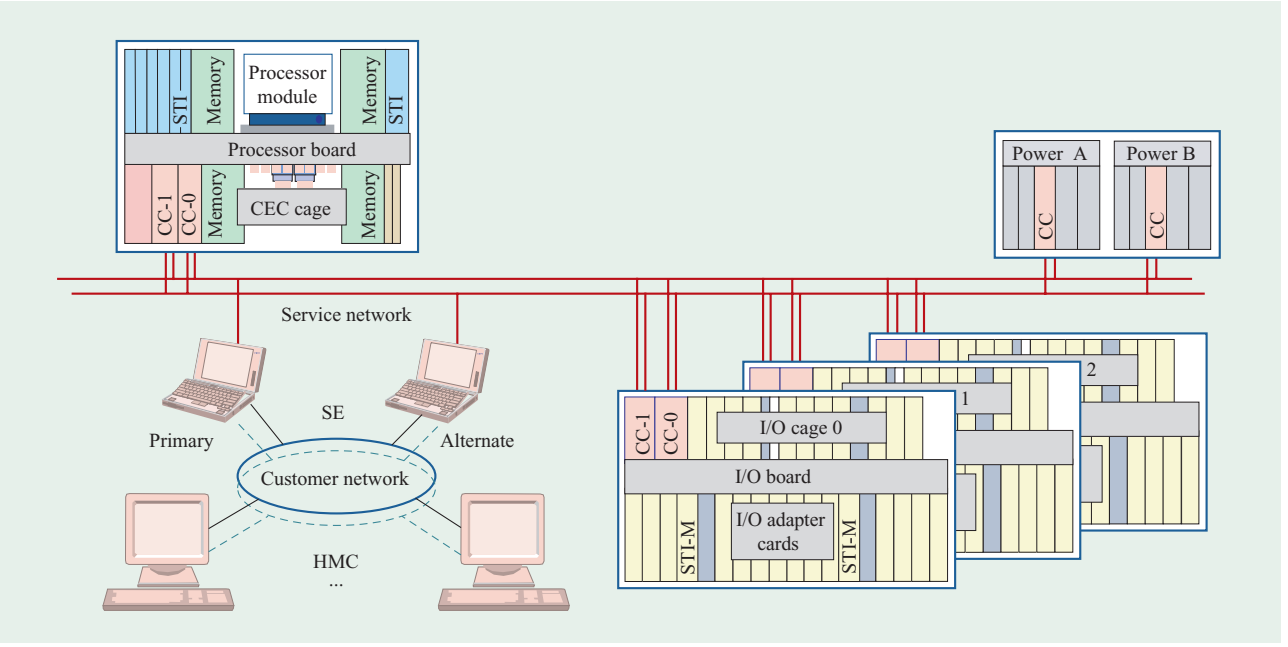

Figure 1

z900 system control structure and physical hardware packaging.

project to establish a new architecture for the hardware configuration service as one of the key base components within the support element.

The pre-z900 implementations for these services had become increasingly difficult to maintain during the years when the S/390* CMOS systems were approaching high-end functionality and larger capacity. In line with this growth, more requirements were appearing that had not been anticipated in the initial design ten years ago, and as a result, more exceptions had to be supported. Finally it was concluded that a redesign was necessary to eliminate problem areas and to build a new base for supporting the new zSeries product line beginning with the z900.

Furthermore, a new framework could be planned from the beginning to serve as a nucleus for re-engineering of larger portions of the service subsystem over time. The concept for this framework had to be established such that it could support much more than just pure configuration functionality; it had to be designed so that its objects which model the system hardware could receive more functionality in later releases and become proxy objects for the real hardware objects.

With this motivation in mind, in the next section we examine the responsibilities of the service subsystem with the SE at its center; we then explore its configuration domain in Section 3. The main part of this paper then focuses on the framework architecture, the hardware object model, and the rule concept in Sections 4, 5, and 6. We then explain the framework infrastructure with respect to object communication and event modeling in Sections 7 and 8. The paper concludes with an overview of application interfaces in Section 9 and a subsequent future outlook.

2. Overview of z900 system control structure and support element

The IBM eServer z900, like its 9672 predecessors, has an "out-of-band" service infrastructure, which is an approach generally characterized by the provision of hardware management and support functions outside the scope of the host operating system(s). In addition to the cost advantage of implementing a single service layer which supports multiple (different) host operating systems, additional rationales for this external functionality are derived from the reliability, availability, and serviceability (RAS) requirements for a large high-end server such as the z900. Such servers, with mission-critical applications and "24/7"-type workloads, typically have an "out-of-band" service infrastructure which permits service/maintenance operations to be performed concurrently with customer operations at the operating system level.

All of these functions, including monitoring, control, and error-handling functions for the system, are performed by a standalone external service subsystem

with the support element (SE) as its central component. The SE is physically implemented on a standard IBM ThinkPad* with OS/2* as its operating system and is packaged within the frame of the z900 server.

Figure 1 shows the main system packaging structure and the components of the service subsystem, including its communication paths, in red:

- A physical z900 system consists of multiple "cage enclosures" with one system board each: a single cage for the central electronic complex (CEC) with the main processor board, the processor module, and up to three I/O cages with I/O adapter cards. In addition, there are two power supplies which provide redundancy within the power distribution infrastructure and are treated as a single cage from a control point of view. The functional interfaces between the CEC cage and the I/O cages (not shown in Figure 1) are provided via self-timed interface (STI) link cables from STI connector cards in the CEC to STI multiplexor (STI-M) cards in the I/O cages.
- The interface between the service subsystem and the system hardware in each cage is provided in a redundant fashion by a cage controller (CC), which performs the low-level monitoring tasks for all of the physical hardware components and service interfaces within the scope of its cage.
- The support element (SE) is the home base for all initialization, configuration, control, and serviceability tasks with the scope of a single z900 system. Figure 1 shows that a primary SE and an alternate SE [2] are connected to the cage controllers via a redundant service network (which is a 100Mb Ethernet) and to the HMC via the customer network. Thus, the SEs are also able to provide firewall functions between the two networks.
- The hardware management console (HMC) provides a multi-system point of control with the customer user interface for all hardware management functions of multiple z900 systems. There may also be multiple HMCs connected to a single z900 via the customer's intranet; these HMCs then act as peers to one another (i.e., they provide redundancy for backup purposes and high availability without requiring a failover[1]).

The support element supports the following system-level requirements:

- *Configuration*: This is the task of determining the physical installed hardware at start-up and power-on time; it also includes the management of changes concurrently with customer operations. Two types of configuration data are maintained on a zSeries system: the physical hardware configuration, which is based on the sensing of the installed hardware with its vital product data (VPD); and the logical configuration, in which the customer assigns logical definitions to installed hardware via the I/O configuration data set (IOCDS) and in the system profiles.
- *Initialization*: These tasks include the initiation and monitoring of all start-up functions system-wide. The support element provides the boot server and code load support for all controllers and all system firmware. Further initialization functions control and evaluate the hardware self-test, with automatic isolation and reconfiguration of failing hardware.
- *System control operations and monitor functions*: This is the function set of all manual and automated operations which may originate from the user interface on the SE or may be issued remotely from an HMC. Also included are all system-level event handlers and monitor functions, such as system activity display.
- *Error-handling functions*: These require repositories for first error data capture, traces, and dumps, as well as automated problem analysis, error filtering, and initiating of call-home to IBM support centers via the HMC. The problem management and repair functions must support the complete problem life cycle from problem record creation until problem closure time, when parts are replaced via guided repair functions.
- *Change management and hardware upgrade functionality*: This includes control routines for concurrent code update and the management of multiple independent code streams. It also contains functions supporting hardware change management, such as concurrent installation of new hardware and concurrent upgrade on demand for processors, memory, and I/O.

Most of this functionality on the SE is realized via application programs which run in their own address space and, in some cases, have also application counterparts within the firmware running directly on z900 hardware.

3. Problem domain overview: z900 hardware configuration requirements

Figure 2 shows the high-level hardware logic structure on the main processor board and the outboard connections via STI links to I/O domains within an I/O cage. In contrast to Figure 1, Figure 2 presents a more detailed functional view of the processor module and its internal structure, together with some details on the I/O. All of the illustrated physical components, including bus structures and link interconnects, are important for the configuration modeling. In the CEC domain, these are the processing

[1] The term *failover* typically denotes the relocation of a function to a backup system in a failure situation.

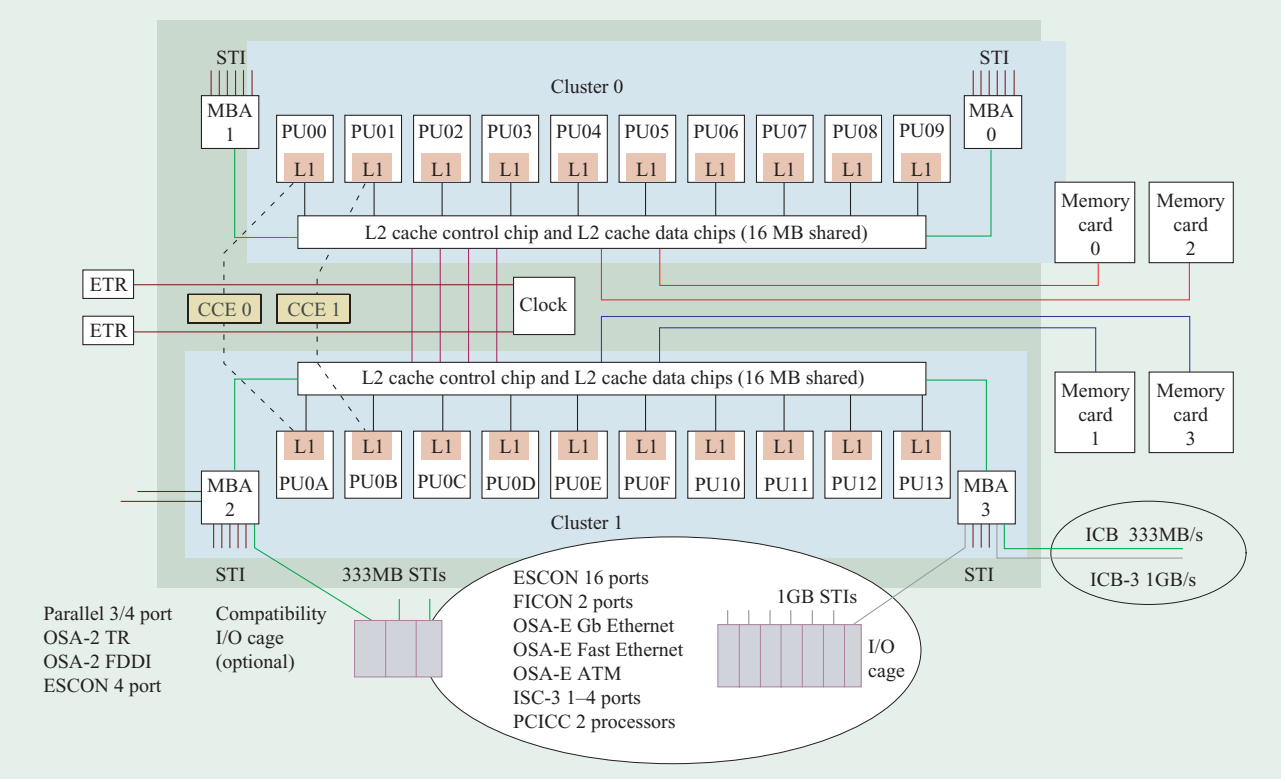

Figure 2

z900 hardware logic structure.

units (PUs), the cache controller and data chips (SCCs and SCDs), the main storage controllers (MSCs), the memory bus adapters (MBAs), the cryptographic coprocessor elements (CCEs), the clock chip, and the external timer reference (ETR) units. In the I/O domains, the corresponding entities are the STI multiplexors (STI-Ms) with secondary STI links wired within the I/O boards to I/O adapter cards such as ESCON*, FICON*, Open Systems Adapter (OSA*) cards, intersystem channels (ISCs), and PCI-based cryptographic coprocessor cards (PCI-CCs) [3]. For connectivity within a Parallel Sysplex* environment, integrated cluster bus (ICB) links are also provided via cables external to the system.

Apart from the general z900 hardware structure, there are also more specific hardware configuration requirements to be supported:

- *Plug detection and configuration of the physical hardware components*: The information about hardware components is gathered with the granularity of field-replaceable units (FRUs) or packaging units. The functional and operational characteristics of these hardware parts must be modeled and maintained over time. In addition, an information base for the functional entities (functional units) located on these FRUs must also be established. This includes information about addressing schemes, maintenance access paths, and the relations between functional units and packaging units.
- *Evaluation of the operational characteristics of installed hardware*: After execution of hardware self-tests, failing functional units must be electronically isolated ("fenced") and then the maximum fully operational subset must be determined in order to generate valid and consistent configuration data for system initialization.
- The zSeries system design introduced the concept of "logical-to-physical mapping" to shield an operating system from recoverable errors at the physical level. The best example of this is a function called "dynamic CP sparing": When a physical processor (functional unit) encounters an error that leaves the last committed architected state of the engine still intact, this state can be extracted and implanted into a spare, "hot-standby" processor so that instruction processing can resume without requiring any notice to the operating system on which it runs. This concept of logical-to-physical

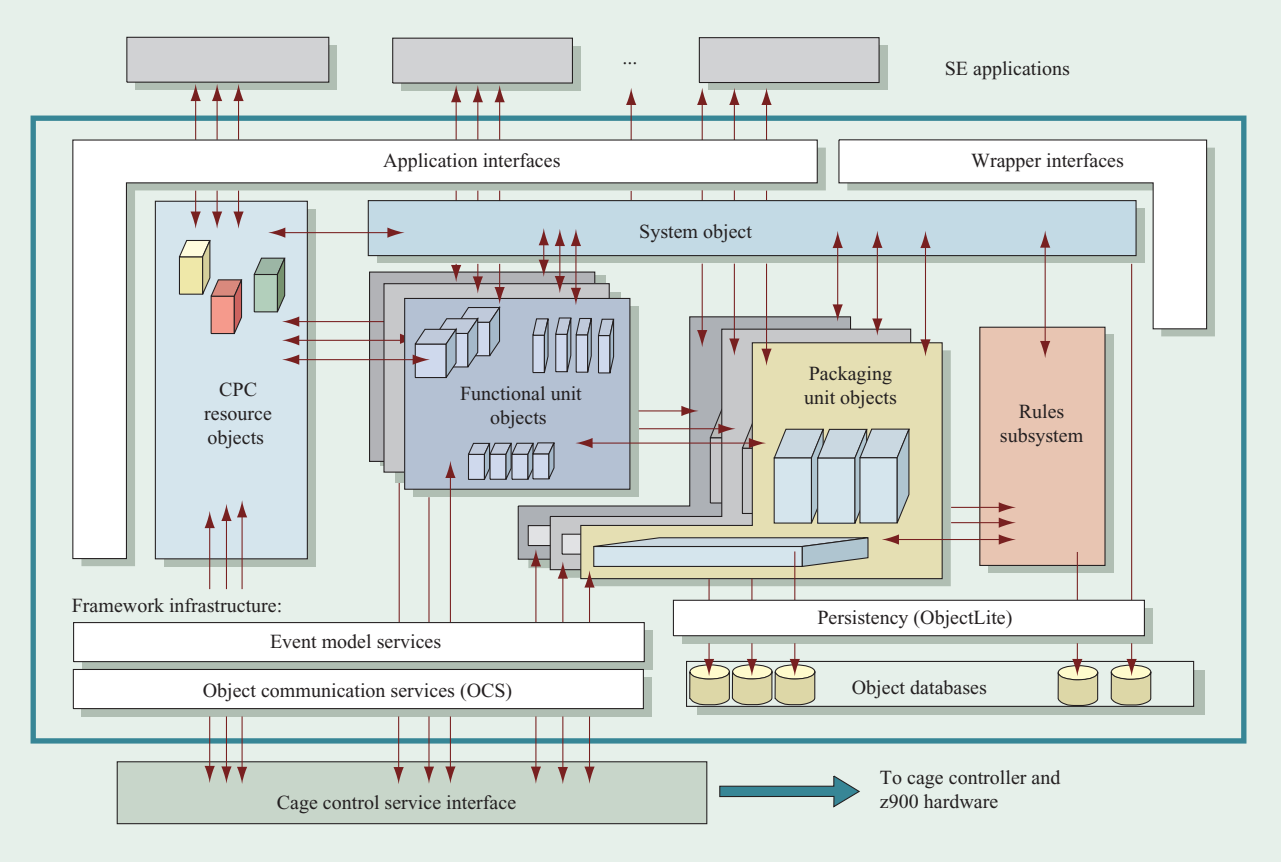

Figure 3

Framework architecture and building block structure.

mapping requires logical units independent of the physical level, which carry their state and their own logical identification independently of the underlying functional unit (this is similar to a virtual machine, except that there is always a one-to-one mapping).

4. Overview of framework structure

On the basis of a detailed analysis of the problem domain, system requirements, and future conceptual goals, a framework architecture was established. The architectural concept of the framework is based upon the principles of abstraction and information hiding, so that any component has knowledge of only the interfaces with other components and nothing more. A further design goal was the use of well-known design patterns such as *observer*, *composite*, and *facade*, in order to obtain an easily understandable architecture. According to the different abstraction levels of the hardware (i.e., the physical, functional, and logical views), there exist three different types of problem domain objects: packaging unit, functional unit, and logical unit objects. The main components of this architecture (**Figure 3**) are described in the subsequent sections.

First, the hardware object model (HOM) is the centerpiece of the framework, with class instantiations representing the system components and abstractions consisting of packaging units, functional units, and central processor complex[2] (CPC) resources (logical units), together with a system object as an anchor point. Second, a rule subsystem is established on top of a rule database which specifies the blueprint of the object model and the system via rules. The foundations for the framework infrastructure are object communication services (OCS), which provide "controlled" object access (defined by an interface architecture) and separate interfaces from class implementations. In addition, event model services have been designed on top of the object communication services in order to support event propagation and

[2] *Central processor complex* is a term defined by the z/Architecture* to denote a "system" (see [4]).

registration mechanisms, not only inside the framework but also as service to applications accessing it. The object model is provided with hardware events and status information by the cage control service interface, which provides a path to the physical hardware via cage controllers.[3] Finally, application interfaces and wrapper services have been established to provide access to the framework for application programs and also to hide the new functionality behind compatibility interfaces for applications that were not intended to be modified to access the framework directly. The framework persistency via object database files has been designed with the help of an object-streaming utility (ObjectLite) which supports object mapping into virtual memory.

5. The hardware object model

The hardware object model is the object-oriented representation of all parts of the system configuration: packaging units, functional units, and logical units. It creates individual components and maintains relationships among them. It also provides addressability to them and keeps track of their internal states.

The packaging unit objects reflect the physical hardware based on plug detection. A packaging unit object is instantiated for every sensed or defaulted (a unit not sensed) hardware field-replaceable unit (FRU). The packaging hierarchy of the z900 is reflected in the object model and follows the composite design pattern. A cage may consist of nodes or cards, and a card may contain daughter cards.

The functional unit objects reflect functional entities, which are active elements residing on packaging units. Examples of functional units are processing units (PUs), memory, clock, and channels. A functional unit keeps track of clock states and function states [e.g., running and isolated (fenced)] and propagates state changes to other related functional units.

The CPC resources are a logical representation of the system resources as defined by the z/Architecture [4]: central processor complex (CPC), CPs, CHPIDs, and LPAR images. The CPC object instantiates all of the other CPC resource objects on the basis of the reported functional units and also maintains the mapping between the functional units and the CPC resources.

Object model initialization

The anchor point for the complete object model is the system class, which follows the "singleton" design pattern [5]. It begins the configuration process for the cages and performs some overall system configuration tasks. This cage configuration process is event-driven; as soon as a cage controller enters the service network, the system object receives a "cage detected" event from the cage control service interface which listens to the service network. It uses the configuration rule component of the framework (described in the next section) to instantiate the required cage object and then tells it to configure itself by determining its contained units and re-applying rules.

Since events like this may occur at any time, even after the z900 system is up and running, there is a need for a certain synchronization point at which the system object can assume that all operational cages have registered so that it can begin the global configuration validation. To this end, the system object must establish global I/O configuration information across all cages; this is gathered by a hardware topology sensing for the STI link connections. Any further detected changes beyond this synchronization point trigger a new configuration and validation sequence based on the last validated and saved results, and add incremental changes to the previous baseline.

Within the initialization process, the individual steps are performed recursively along the containment hierarchy of the packaging units. Each packaging unit container object is responsible for the instantiation and attribute settings of its contained objects, for validation across multiple objects, and for establishing relationships among multiples of its contained objects. This modularity allows for parallelism during the configuration step and also eases the support for hardware hot-plugging.

The configuration of the functional unit objects is more complex than that of the packaging units, since they must be connected according to their dependencies. This results in a "functional path" (a directed graph with the functional units as nodes), which is used to propagate state information. In general, a functional path depends on the static physical layout of cards and boards and the sensable interconnect topology, which together describe how functional units from multiple FRUs are connected to one another.

At the cage hierarchy level, the CEC cage instantiates a CPC object as the logical representation of a single z900 system. The CPC object then configures its dependent resource objects on the basis of the functional unit mapping information acquired during system activation. This processing is performed at the logical unit level and is completely based on the z/Architecture, which is directly modeled by the logical unit class hierarchy; therefore, additional rules are not required here.

Run-time functions

During steady state, the object model keeps track of status and state change information for the functional units within the associated objects. This is the responsibility of functional unit objects and so-called clock domain objects, which represent union sets of functional units driven

[3] The cage-controller and service network subsystem are described in [1].

by the same physical clock. Such state changes are propagated along the functional path; thus, if a functional unit is disabled, all units "behind" it within its functional path have to be disabled as well.

All of the state change processing is event-driven; it is based on a registration mechanism which notifies the clock domain objects when certain clock events are reported from the cage controller service interface.

Figure 4 shows the main class relationships [5] for packaging and functional units.

Design rationales and alternatives

Given the structure of the z900 service subsystem as shown in Figure 1, one of the key design strategies was to avoid precluding a later function redeployment of object model components to different locations. For example, the packaging unit objects and even some of the functional unit objects could reside on the cage controllers, and the CPC resource objects could be established on the hardware management console. This, together with the multi-process application nature of the support element, was the principal reason for establishing specific object communication services, which support a transparent cross-process method invocation suitable also for extension across platforms. However, in the current implementation it was decided to establish the complete object model on the support element and to connect to the hardware access services on the cage controllers via a cage control service interface layer.

The modeling of packaging units had to take into account hardware design changes at the physical packaging level as well as model dependencies, since processor cards and boards vary from one model to another and will probably change their structure from one generation to the next. Given such hardware dependencies and the need to link the classes to hardware identifications in the vital product data information, the packaging units might be implemented on the basis of class inheritance, i.e., by using a subclass for each processor card version. However, such an approach has some disadvantages:

- During hardware bring-up, there is a requirement to support "partially good" cards and to create only functional units for the defect-free components on them. Doing this by means of subclassing would result in maintaining additional classes and complexity, since adding case statements to a class would either produce a considerable amount of "dead" code after the hardware bring-up phase or require code changes for code already tested.
- In addition, also typically during the hardware bring-up phase, a substantial number of schedule-critical workarounds are sometimes required. A subclassing approach would tend to be more inflexible here, since it would require more frequent recompiles and software driver builds during the critical bring-up time.

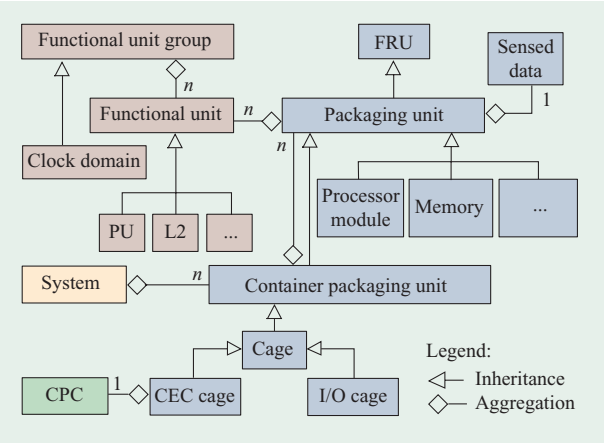

Figure 4

Class relationships for packaging and functional units.

A better alternative to this is a rule-based approach, since many of the classes do not differ in their key functionality but primarily in different attribute values (e.g., one processor card has two processors and another one may have four, although both cards support the same methods). With a rule-based approach, the values of the attributes derived from the sensed hardware information are defined by rules; and during the initialization of the object model, the system, the cages, and the card objects use rules to determine the type and number of packaging units or functional unit objects that are to be created. The object creation is not accomplished by a direct call to the new operator, but instead by dynamic loading. This enables a cage object to create card objects that were unknown at the time the cage class was implemented.

The chosen approach has advantages with respect to minimization of dependencies, and it also results in a larger flexibility, which is certainly justified over multiple machine generations. In particular, the rule concept was chosen so that the class implementations do not depend on the implementation of the rules by using a facade-based concept [5] for the rule access.

6. Concept of configuration rules

It is advisable to concentrate configuration knowledge in one place instead of distributing such data over multiple areas. With a rule-based approach, it seems possible to establish a concept that limits most of the required changes which occur over time to local changes of configuration rules and avoids changing the code that uses the rules.

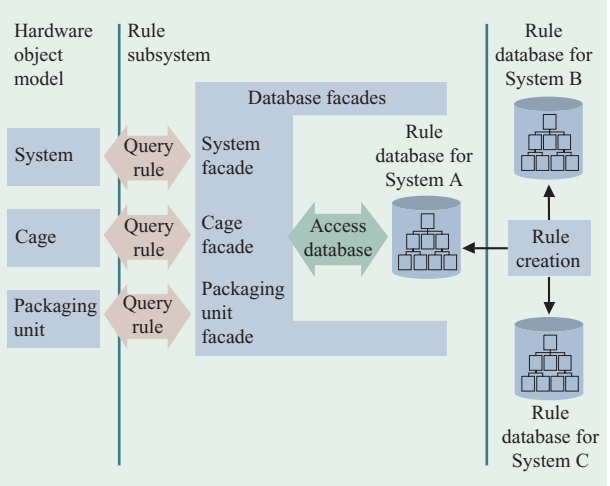

Figure 5

Hardware object model and rule database.

The new configuration functions were implemented on the basis of a rule database concept that describes for each potential FRU the object hierarchy that must be built upon it and how these objects interconnect with others.

The main idea of the configuration rule concept is to model configuration rules as C++ classes. For example, a configuration rule is used to create certain configuration objects such as packaging units or functional units, to describe a relationship between two or more of these configuration objects, or to set the values for attributes of such objects. Each C++ class then represents one particular type of rule, and instances of a class represent specific rules of the same type.

Configuration rule objects are used to create the configuration objects of the object model, which represent cages, cards, and chips, packaging units, and functional units. They are also used to set up the functional units within sensed packaging units, and they define the names of these units. They describe which cards are allowed to be plugged and specify attribute values of cards (e.g., the memory size on the memory cards). In addition, they describe how cards are internally structured and also express certain relationships between different cards (e.g., one card may require the presence of another). The rule subsystem can be considered to be a factory for configuration objects intended to make the hardware object model design more stable against design changes to the z900 hardware component structure.

The rule instances are made persistent and stored in an object-oriented rule database that is machine-dependent. All details of the database setup are hidden behind so-called database facades, and the hardware object model code performs its processing by making calls to the appropriate rule facade. This approach allows for a database implementation which is independent and decoupled from the rest of the framework so that the database design itself can also be easily replaced. **Figure 5** shows the relationships among the object model, the database facades, and the rule database.

The rule design distinguishes among the specification of a rule (i.e., the design of a class as a representation of an abstract rule), the instantiation of a rule object (i.e., the instantiation of a concrete rule object in a database), and the execution of a rule (which may result in the creation of a configuration object or the setting of an object attribute). There is also an explicit separation between rule creation and run-time rule access: The creation of rules and databases is part of the compile and build process and is not performed in a customer environment.

The process of updating and deleting rules in the database is mostly transparent for the object model classes. In most cases, it is sufficient to create a new instance of an appropriate, already existing rule class when a new rule is required. In this case, only the rule database must be updated; no code change for the facades and for the callers of the facades is necessary. If, for example, a new processor module is to be supported which differs from a previous one in the number and types of functional units and the values of some attributes, this can be achieved by changing the rule database only.

The rule facades encapsulate database details (such as opening and closing the database, transaction handling, retrieving data from the database) and specific database implementations.

Each facade class supports several rule types for queries to the database. A rule type is a set of related methods (e.g., for validation of the cage layout) and is offered by a rule facade but may internally lead to several calls to different rule objects from different classes. The database and the rule classes are encapsulated. The facades are the only rule interface for the framework code.

Unlike the rules, the facade objects are not persistent. Instead, they are created dynamically for certain objects (e.g., a specific facade object is created for each cage object) and retrieve all rules from the database that apply to this object by introducing rule dependency objects. This avoids any additional searches for rules that may be applicable, and the calling object does not have to know how to retrieve the applicable rules from the database. The facade provides an interface tailored for this object and ensures that only valid rules are called.

7. Object communication services (OCS)

During the analysis and design phase for the framework in 1997, several publicly available object broker services

were evaluated regarding their capability to serve as a conceptual base for the object communication of the framework. Among these were several implementations of CORBA** [6], an application framework for real-time applications, and also the Component Object Model (COM**) [7] from Microsoft. We also looked at Sun Microsystems Java** with its remote method invocation capability, but concluded that it was not feasible at that time because of our underlying performance requirements. Building on the native OS/2 system object model (SOM) was also rejected because of considerations of portability to the cage controller environment. Directly reusing CORBA implementations or COM was not possible on the OS/2 platform because of license costs for CORBA object request broker packages and the availability of COM on only the Microsoft Windows** platform. Also, relying on an external vendor or supplier for a code package which typically would largely have exceeded our requirements and which would have created unnecessary project dependencies was rejected.

Influenced by some of the publicly available concepts and ideas from CORBA and from COM, these evaluations finally resulted in the decision to build the functionally required minimal set of object broker services ourselves.

Interfaces and components

The first key concept we chose to adapt and implement was the interface concept as proposed by COM and other broker services. Objects are given logically separate interfaces that can be also aggregated into multiple compositions separated from the object interfaces. The main rationale for this was to have a means to separate client and server class definitions and to avoid the typically heavy code interdependencies which can sometimes be found in object-oriented software projects.

The chosen interface approach avoids the propagation of implementation-specific knowledge by separating the class definition from the interface definition and using an interface definition language (IDL) which connects client and server and, at the same time, separates their code to avoid creating dependencies between them. The original class interface is typically subdivided into multiple subset interfaces which have the form of abstract interface classes and which define public methods and their signatures as pure virtual functions, but do not define attributes and nonpublic methods. The implementation for these methods is provided by an implementation class, which inherits from the abstract base class. A client is given access only to the definition of the abstract base class; thus, it has knowledge of the methods only, and is not given any information about implementation details such as attributes or physical structure of objects.

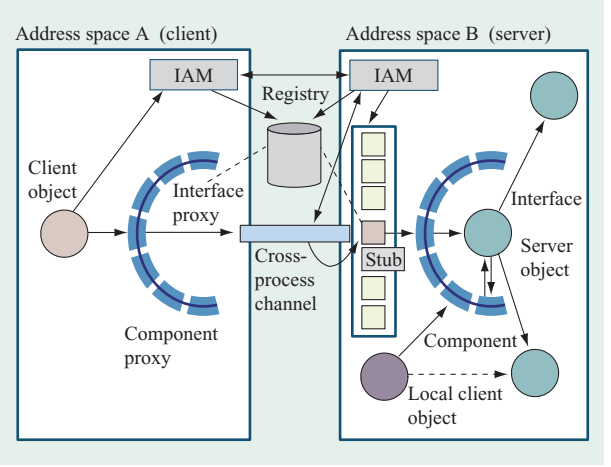

Figure 6

OCS interface mechanism.

Furthermore, individual interfaces can be reused in different class contexts or composed to interface aggregations; this is what is usually described as an object component concept. By using this approach, client complexity can be reduced significantly, since a client dealing with one particular interface to an object has no need to know about the object's other interfaces. Thus, the client can handle totally different types of objects with only the knowledge of how to ask the objects for the specific subset of interfaces that are required and how to use them.

Remote interface invocation

In distributed applications, objects typically exist in multiple address spaces within a single system and sometimes even on multiple remote systems. The selected remote interface access for OCS assists objects to access objects in other address spaces in a manner that is almost totally transparent to the calling object. This means that for a client object, the method invoked on the interface of another object, which happens to exist in a different address space, appears to be identical to a local method invoked within the same address space.

The only visible implication of cross-process and cross-system method invocation is within the initial creation of interfaces and their registration. An object must have some knowledge about where it might find its communication partner: within its own address space via a library call, in another process on the same processor, or even remotely in another process on another machine.

Figure 6 shows the use of interfaces with OCS. Objects are always accessed using a set of interfaces which are aggregated to components. This is independent of whether

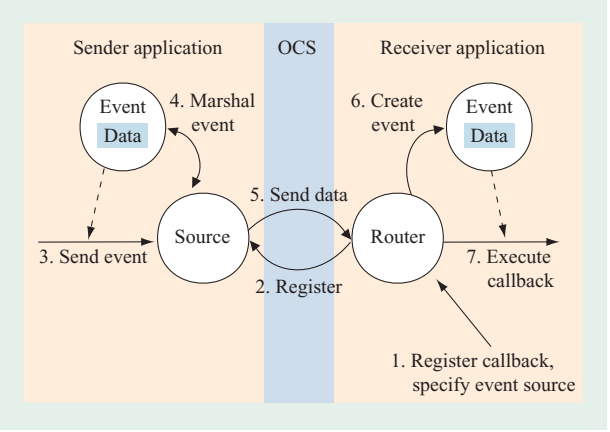

Figure 7

Event model control flow.

the access is in-process or across process boundaries. If a process boundary is to be crossed, client proxies and server stubs are transparently introduced. The proxy server in the client's address space behaves to its client exactly like the real server, while a client stub propagates the client's request to its server in exactly the same way as a local client. This also includes a synchronous server behavior from the client's point of view. The interfaces between client and in-process object are exactly the same as between client and proxy and between stub and target object on the server side. OCS does not expose the way in which messages are to be transported from one module to another; this is performed via an underlying cross-process channel implementation which is hidden from its users.

All OCS services are provided via a separate run-time library called interface access methods (IAMs), which is used on both sides of the address space boundary for synchronization and transport services (Figure 6) and which generates the interface proxies and stubs with the help of appropriate object factory mechanisms.

In summary, with OCS a powerful object broker has been designed which has proved to be a flexible base for the overall framework. It has served as the glue between the framework components, as shown in Figure 3. All of the key object interfaces which required external access have been built with OCS; to some extent this has also been done for "internal" interfaces for reasons of architectural separation (e.g., between CPC resources and the other object categories).

The next two sections provide more details on the use of OCS: the event-propagation mechanism for the framework, internally as well as externally, and the application interfaces, which were both built with OCS.

8. Event model services

An important task of the support element is to monitor the system's hardware and software status and initiate appropriate actions when a failure is detected. This requires a mechanism for registration and distribution of the status changes, which are reported via interrupts from system hardware and firmware to the framework entities and other support element applications. The framework component providing this service is denoted as the event model in the further context of this section. The design of this component is based on the publisher/subscriber design pattern, which is also known as the observer pattern [8]. It defines a one-to-many dependency among objects and provides a mechanism which notifies and updates all dependents automatically when a registered object state changes. The roles of events, sources, and routers, which are the key elements of the event model, are described in the following and illustrated in **Figure 7**.

User-defined event class

Status information is represented as event objects. During the processing of a state change of an object, these event objects are automatically sent to every object which previously showed its interest in this particular change by issuing a registration request. The event object contains all necessary information. A key problem in any type of interprocess communication via a generic transport mechanism is the marshaling of data to be transported on the sender side and the reconstituting of the object on the receiver side. The event model thus requires the designer of an event to put this knowledge into the implementation of a corresponding event class that converts the data inside the event class into a generic byte stream (marshaling) and converts it back into the event object (de-marshaling).

Event source and router classes

With a generic transport mechanism in place, there is still a twofold problem: how to distribute the event from the source to any registered receiver, and what to do when an event has been received.

For this purpose, the event source class was designed so that each event is distributed to every registered receiver. The event source object then uses the event-marshaling method to extract the data before sending it to the registered receivers via a generic transport mechanism.

The event router class allows each user to associate his own function/class method with an event to be executed when the corresponding event has been received.

Also, the event model allows for an m-to-n relationship between event sources and routers. Since the registered call-back methods are executed in the context of the event source by default, an event queue class has been designed to allow receiving applications to queue up incoming

events and dispatch the method call-back within the receiver context. However, the use of an event queue has been made optional for flexibility purposes.

9. Application interface and wrapper service for legacy code

Application interface

The main task of the framework is to provide abstracted information and functionality to other application programs running within the support element environment, as described in the Introduction. Therefore, an application interface has been established which provides access to the HOM objects.

The primary service required by applications is a query function for the zSeries hardware configuration and topology. It enables them to determine which hardware components (packaging units and functional units) are available, how these parts are connected, and how these resources are configured to provide logical computing resources (CPC resources). This type of information can be gathered by traversing the object relationships, since these were created to model the system structure.

In order to achieve this, the application interface of the framework has not been implemented as an additional software layer on top of the objects, but instead has been designed to allow direct external access to individual objects (as indicated in Figure 3). However, with applications running in separate address spaces, remote method invocation is required for this purpose and could be achieved by externalizing functionality via the OCS mechanisms. Since OCS also provides a powerful component model, the visibility and accessibility of methods could be provided in a controlled manner, which is important for hiding implementation details and internal functions.

Besides static configuration information, some applications also require dynamic or transient status information. Examples of this are the power status of a hardware component, the functional status of a processing unit (running/stopped), or the defect information of a certain part. This category of information is queried system-wide during each SE start-up, and all further status changes are reported to the SE as interrupts and distributed to the required objects using the event model services.

From the application side, there are two ways to access this information. The first one is analogous to the way configuration information is provided. Since the status information is stored as part of an object, it could be made available through method invocations and externalized via OCS. This mechanism is sufficient for many simple applications that must check the hardware status before triggering an action. For more complex

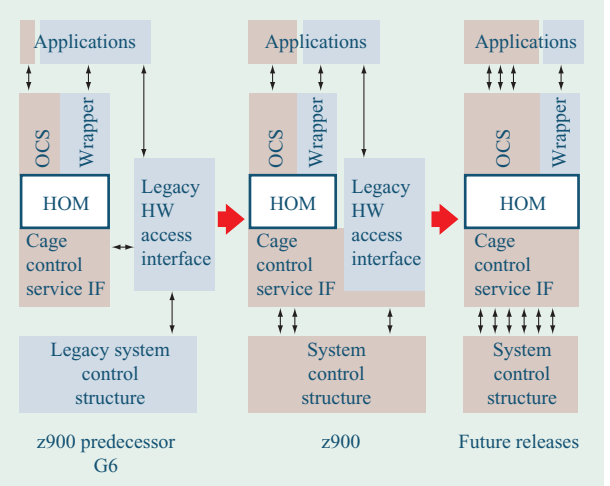

Figure 8

The evolutionary approach.

applications with monitor-like tasks, however, this is not appropriate. In order to avoid polling, this class of applications requires status distribution following the publisher/subscriber pattern. The event model services, also used to distribute information inside HOM, offer this capability to the applications: By registering for status change events at the corresponding objects, the event distribution flows seamlessly from HOM objects to application objects, since the event model services are based on (transparent cross-process) OCS technology.

The third important aspect covered by the application interface is providing functionality of the z900 hardware and firmware to application programs for method invocation (e.g., powering off a slot for hot-plugging or stopping a processor). Since HOM objects have to represent their real-world counterparts, they are responsible for providing such functionality as methods. The object itself must have detailed internal knowledge about its service as a representative so that it can provide the necessary implementation to trigger the appropriate functionality. This provides a means of offering abstract interfaces to applications and concealing hardware-related implementation details within the framework.

Wrapper service for legacy code

For reasons of risk mitigation and workload constraints, the hardware object model was not delivered in a single code release. **Figure 8** indicates that a basic object model, limited in size and functionality, was released with the predecessor system to the z900 and deployed by a small set of applications in order to establish the infrastructure elements and to gain experience with the technology.

Even with this preparation, it was not possible to adapt the complete application set during the development cycle of the zSeries 900. Thus, the focus for the z900 release was to provide a fully functional object model with a complete infrastructure including an application interface and a limited set of adapted applications. A complete transition with a wholly restructured or redesigned set of applications utilizing the enhanced capabilities of the object-oriented approach was set as a mid-term goal, and a phase-in strategy was also established to release adapted applications along that path.

To make progress with this evolutionary approach, it was essential to enable the execution of applications already using the new interfaces while still supporting the majority of unchanged applications. A large set of applications that had been created, expanded, and maintained over the last ten years required support with all system-specific information and function calls as they had been provided in the predecessor systems. The services and information now covered by the HOM were provided by a variety of different interfaces in previous systems. Each one was focused primarily on a single specific aspect, such as providing static configuration information or offering query support for specific dynamic status informations. Generally, these functions were implemented independently of one another; thus, several data repositories with contents that sometimes overlapped were created and maintained. A key benefit and strength of the HOM design point was the centralizing of all this information to a single source; keeping the legacy services in parallel to serve unchanged legacy applications would have been counterproductive. These legacy interfaces were provided on top of the object model by using wrapping mechanisms to map syntax and semantics of data and functionality between the old "function-oriented" interfaces and the object-oriented HOM approach.

Most of the creation of these mapping layers was rather straightforward. One reason for this was that an important step during the analysis phase of the HOM project was to collect, understand, categorize, and abstract the existing functions and interfaces.

The effort put into wrapping was a significant percentage of the overall project work, but compared to the effort of adapting all applications in a single step, it was only marginal. However, it is important to note that the need for wrappers will be eliminated over time, since all new functions can be designed into HOM objects and applications which use the objects directly. Along this path, new or adapted applications will then take full advantage of the capabilities of the object model; they will be able to obtain access to information and functions beyond the limited scope of the wrappers and accomplish their tasks more easily.

10. Conclusions and outlook

As a primary result of the hardware object model concept, a framework has been built for the z900 that is clearly much more than just a configuration framework. From a functional perspective, a framework has been developed that provides an excellent base for further integration of the key "business logic" of the zSeries support element. By building on the z900 version of the framework, the configuration and status representation functionality of the SE can easily be extended to include further functional aspects, such as hardware and firmware initialization support logic, as well as the back-end error handling and service/repair control logic.

From a software engineering perspective, the introduction of the hardware object model has become a success story with the general availability of the z900. The introduced methodologies have clearly improved the design flexibility and have also resulted in a better overall code quality that is already evident from the reduced numbers of field problems. The evolutionary approach of introducing the new design with a staged delivery of the functions beginning with the preceding server has already paid off by diminishing the risk of delaying the scheduled availability of the z900 system.

From a software technology perspective, the framework design has been a major undertaking which has used and extended state-of-the-art methodologies. Altogether, it has been a balanced mix of development unique to itself where necessary, and the reuse of concepts and components where possible. Since most "open standards" were in their early stages and in flux at this time, their reuse has not been an easy task during the concept phase of the project. In the meantime, Java has clearly emerged as the predominant object technology, and it is considered to be the base for future developments in this area. Also, the use of technology such as the Common Information Model (CIM**),[4] and interfacing with an industry-standard CIM object manager is under investigation for future projects.

Acknowledgments

Without the support of the Boeblingen and Endicott development teams during the analysis, design, and implementation of the framework, this project would not have been successful. Special thanks and recognition belong to Michael Johanssen for his contributions to the OCS design, and to Jeff Conklin, Jill Lavin, and Nancy Pellicciotti for the design of the CPC resource part of the framework and their help to debug the overall object model and make it work. We also thank Prof. Dr. Joachim Goll, Andreas Maier, and Stephen Nichols for their valuable comments.

[4] The Common Information Model (CIM) is a standard for management of computer systems developed by the Distributed Management Task Force (DMTF) (see [8]).

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Object Management Group, Sun Microsystems, Inc., Microsoft Corporation, or Distributed Management Task Force.

References

1. F. Baitinger, H. Elfering, G. Kreissig, D. Metz, J. Saalmueller, and F. Scholz, "System Control Structure of the IBM eServer z900," *IBM J. Res. & Dev.* **46,** No. 4/5, 523–535 (2002, this issue).
2. B. D. Valentine, H. Weber, and J. D. Eggleston, "The Alternate Support Element, a High-Availability Service Console for the IBM eServer z900," *IBM J. Res. & Dev.* **46,** No. 4/5, 559–566 (2002, this issue).
3. D. J. Stigliani, Jr., T. E. Bubb, D. F. Casper, J. H. Chin, S. G. Glassen, J. M. Hoke, V. A. Minassian, J. H. Quick, and C. H. Whitehead, "IBM eServer z900 I/O Subsystem," *IBM J. Res. & Dev.* **46,** No. 4/5, 421–445 (2002, this issue).
4. IBM Corporation, *z/Architecture Principles of Operation*, Order No. SA22-7832-00, December 2000; see *http://www-1.ibm.com/servers/eserver/zseries/zos/bkserv/r1pdf/zsys.html*.
5. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Co., Reading, MA, 1994.
6. *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, Object Management Group, July 1995, updated July 1996; see *http://www.omg.org/technology/documents/vault.htm#CORBA_IIOP*.
7. *The Component Object Model Specification*, Version 0.9, Microsoft Corporation and Digital Equipment Corporation; see *http://www.microsoft.com/com/resources/comdocs.asp*.
8. Distributed Management Task Force, *Common Information Model (CIM) Specification*, Version 2.2, June 1999; see *http://www.dmtf.org/standards/standard_cim.php*.

Received August 27, 2001; accepted for publication March 19, 2002

Andreas Bieswanger *IBM Server Group, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (anbie@de.ibm.com).* Mr. Bieswanger is an Advisory Engineer working on zSeries service subsystem design. He studied computer science at the Georg-Simon-Ohm-Fachhochschule Nuernberg and graduated in 1994, receiving his Diplom-Informatiker (F.H.) degree. He joined the IBM Development Laboratory in Boeblingen that same year and began work on the design and implementation in different areas of the support element. His work currently focuses on design and evaluation of advanced server concepts.

Franz Hardt *IBM Server Group, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (hardt@de.ibm.com).* Mr. Hardt is an Advisory Engineer currently working on the IBM eServer hardware management console. He graduated in 1988 from the University of Mainz, Germany with an M.S. degree in mathematics, joining IBM at the Boeblingen Development Laboratory that same year. He worked on several projects in the z/OS area and also on an object-oriented framework for embedded controllers in manufacturing systems. Since 1997, he has focused on design and implementation in the hardware configuration of the support element. Mr. Hardt has been on international assignment in the IBM Endicott, New York, laboratory since 2001.

Astrid Kreissig *IBM Server Group, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (kloss@us.ibm.com).* Mrs. Kreissig is an Advisory Engineer currently working on IBM eServer service subsystem architecture. She graduated in 1987 from the University of Aachen, Germany, with an M.S. degree in computer science and joined IBM at the Boeblingen Development Laboratory in 1988. She worked on a variety of projects ranging from z/OS administration software to an object-oriented framework for embedded controllers in manufacturing stations. From 1993 to 1997, Mrs. Kreissig coached several object-oriented projects, working closely with the IBM OOTC. Between 1997 and 2000, she contributed to the zSeries development as a software designer and a tools team leader. Since 2000, Mrs. Kreissig has been on international assignment in the IBM Austin, Texas, laboratory.

Harm Osterndorf *IBM Server Group, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (osterndo@de.ibm.com).* Mr. Osterndorf is a Staff Engineer currently working on zSeries service subsystem design. He studied computer science at the Berufsakademie Stuttgart and graduated in 1990, receiving a Diplom-Ingenieur (B.A.) degree. He joined IBM at the Boeblingen Development Laboratory that same year and worked on design and implementation of AIX software in the area of banking applications and hardware until he joined the S/390 service subsystem in 1994. Currently, Mr. Osterndorf focuses on the transition of the service element code into an object-oriented design in the area of base infrastructure and components.

Gerhard Stark *IBM Application Development and Services Division, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (gstark@de.ibm.com).* Mr. Stark is a Software Development Engineer currently working on development of the PvC WebSphere portal server. In 1977 he joined IBM at the plant

in Sindelfingen, Germany. From 1983 to 1985, Mr. Stark was on international assignment in East Fishkill, New York. He joined the Boeblingen Development Laboratory in 1986; from 1997 to 2000, he contributed to S/390 development.

Helmut Weber *IBM Server Group, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (weberh@de.ibm.com).* Dr. Weber is a Senior Technical Staff Member currently working on IBM eServer system design concepts. He graduated in 1979 from the University of Marburg, Germany, with an M.S. degree in mathematics and physics; he received a Ph.D. degree in mathematics from the University of Marburg in 1982. He joined IBM in 1984 at the Boeblingen Development Laboratory, where he worked on operating system concepts in the Advanced Technology Group. Between 1988 and 1999, Dr. Weber worked on most support processor development projects in the IBM Boeblingen and Endicott laboratories: S/390 systems 9370 and 9371, S/390 CMOS G1 to G6, and the zSeries z900 server. Since 1999, Dr. Weber has been on international assignment, working in the IBM Poughkeepsie laboratory.