Comprehensive support for management of Enterprise Applications

Jens Bruhn, Christian Niklaus, Thomas Vogel, and Guido Wirtz
Distributed and Mobile Systems Group
Otto-Friedrich-University Bamberg
Feldkirchenstrasse 21, 96052 Bamberg, GERMANY
jens.bruhn@uni-bamberg.de

Abstract

During the last decades, performance of available hardware resources constantly increased [15], which enabled the assignment of more and more complex tasks to software systems. As one consequence, the inherent complexity of these software systems also increases, influencing all phases of their lifecycle. The concept of Component Orientation *(*CO*) [17] allows the development of software systems in a modular way through functional decomposition. Administration and maintenance of software systems are addressed by the vision of* Autonomic Computing *(*AC*) [11], based on the idea to assign low level administrative tasks to the system itself. With* mKernel *an AC-infrastructure for component oriented enterprise applications is provided, based on the* Enterprise Java Bean*s (*EJB*) standard, version 3.0 [5]. In contrast to existing approaches, the main advantage of* mKernel *lies within its standard compliance, not prescribing any additional guidelines for the development of applications to enable their autonomous management. It is realized as plugin for an existing container, not requiring any adjustment of the underlying implementation. Moreover, it provides a very fine grained interface for inspection and manipulation of the managed system, taking the specifics of the supported standard into account. Within this paper we present the opportunities provided by* mKernel *to control a managed system.*

1 Introduction

During the last decades, the rapidly increasing performance of available hardware resources enabled the assignment of more and more complex tasks to software systems [15]. New concepts for addressing the resulting inherent complexity of these systems are needed badly. Otherwise, complexity will become the major burden, hindering the further development instead of missing hardware performance [11]. *Enterprise Application*s (*EA*) are a family of highly complex software systems for supporting the business of companies. Their complexity results from the different aspects coped with in combination with the manifold interrelations among them, like e.g. accounting and warehousing. Moreover, EAs are confronted with more or less regular demands for re-configuration, e.g. to support new business areas of the operating company. Changes in the environment of a system might ask for reorganization, e.g. to react to changing workload. Additionally, if services are provided to external clients, EAs must be protected against malicious interactions or attacks. Over time, types of threats will probably change which demands for adjusting defense strategies. Availability of EAs, however, is a critical success factor for the operating company. If an EA becomes unavailable – even for a short timespan – the company might miss business opportunities. The potential loss of reputation and trust can be estimated of being even worse. Against this background, even the temporary shutdown of parts of the system for a planned re-configuration would be very costly and seems to be unacceptable. Consequently, the support for seamless re-configuration is of very high value.

With the concept of *Component Orientation* (*CO*) [17] a paradigm for the development of modular software systems is provided, which addresses complexity during development and deployment. The *Enterprise Java Beans* standard (EJB), version 3.0, [5] specifies a component standard for the realization of EAs on top of the *Java* programming language. It defines, amongst others, a sound component model and includes different facilities simplifying the development of EAs.

For system management during runtime, support is still poor regarding the provision of high level functions, abstracting from fine grained configuration tasks. Here, the vision of *Autonomic Computing* (*AC*) [9, 11, 13] can help. It is based on the idea to assign low level tasks to the managed system itself to disburden human administrators. Aspects addressed might reach from short term reactions to concrete situations, like e.g. the rollback of a transaction

DOI: https://doi.org/10.1109/AICCSA.2008.4493612

to confine the effects of an error, to planning and execution of re-configurations to better fulfill the high level goals of the system. In this context the system acts autonomous regarding taking actions for adjustment. In combination, CO and AC can establish a promising foundation for addressing complexity during the *whole lifecycle* of software systems. CO leads to the development and initial configuration of modular software systems. Thus, it lays a sound foundation for realizing the vision of AC based on clearly distinguishable building blocks. In this context, the provision of solutions, compliant with existing component standards, is required, to support their integration. They should be realized generically, avoiding the need to provide specific solutions for different AC-domains.

With *mKernel* a generic infrastructure for supporting AC is realized. It is based on the broadly accepted component standard EJB 3.0. Because of being platform specific, a comprehensive interface could be realized, taking the specifics of the underlying standard into account. Therefore, a very fine grained management of EAs is possible, being a promising foundation for higher level functions of autonomous management. *mKernel* is realized as plugin for an existing EJB container, not requiring any adjustment of its implementation. Moreover, developers of EAs do not have to follow any guidelines beyond those of the EJB standard. In combination, this allows a seamless integration of CO with the vision of AC. This paper discusses the support of *mKernel* for the different phases of the deployment lifecycle of EJB based systems.

The remainder of this paper is structured as follows: Section 2 discusses related work. Section 3 gives background on CO, AC and *mKernel*. Afterwards, section 4 presents the view on an EA as provided by *mKernel*. Subsequently, in section 5 the phases of the deployment lifecycle are considered regarding opportunities for inspection and re-configuration through *mKernel*. Finally, section 6 gives a conclusion and an outlook on ongoing and future work.

2 Related work

mKernel was developed as generic infrastructure, enabling the management of component based systems following the EJB standard, version 3.0. It explicitly excludes the development phase of applications and does not require the adherence to specific guidelines beyond the EJB standard from developers. Moreover, it does not demand for the specification and integration of AC aspects into the application during development, like e.g. in [18]. In contrast, it supports a clear separation of the original application logic from management aspects and allows their seamless integration and removal during runtime. Compared to existing approaches for AC-infrastructures [8, 10], *mKernel* does not address the management of different types of resources. Therefore, it is possible to provide a very fine-grained and rich set of sensors and effectors for a specific domain, namely EJB. In contrast to existing solutions for the Java Enterprise Edition like [2, 3], *mKernel* does not only focus on the deployment level of application systems. It also addresses aspects of types of components, as well as the instance level regarding interactions during runtime. *mKernel* was designed as plugin inside an existing EJB container, solely based on the EJB standard. It does neither include an implementation of an EJB container, like [12], nor require any adjustments of the implementation of an existing container.

3 Background

As foundation for the further discussion, this section provides a short overview over the basic building blocks of a component based system. In this context, relations to the EJB standard are pointed out. Afterwards, the central concept of AC, namely the so called *Control Loop*, is presented. Finally, it is discussed how *mKernel* is positioned for supporting AC in a component based environment.

3.1 Component Orientation and EJB

In the context of CO, software systems are constructed out of modules, called *Component*s, which provide their encapsulated functionality through well defined *Interface*s to their environment. For fulfilling their tasks, they can make use of interfaces provided by other components. Here, the declared requirement of an implementation of an interface is called *Receptacle*. Regarding the provided and required functionality, a component based system can consequently be established through connection of interfaces and receptacles of the constituting components. The EJB standard follows this concept by providing a component model based on so called *Enterprise Beans* – or *Beans* for short. There are two different types considered, namely *Session Bean*s and *Message Driven Bean*s. For the former type, interfaces can be specified for accessing the encapsulated functionality. The latter type is intended to be used for asynchronous interaction through message passing. In the following, the terms *session bean* and *message driven bean* are used if the concrete type is considered while the term *bean* is used if the corresponding statement holds for both types. Receptacles may be declared for beans through *EJB References* which can be connected to interfaces of session beans. Beans may be configured through so called *Simple Environment Entries* which can be interpreted as a kind of *properties*. These are set during deployment without the need to access the source code of the affected bean. During deployment, each bean is bound to a certain mapped name which can be used to look

up its instances. As unit of deployment, so called *EJB Modules* are considered in the standard. They consist, amongst others, of a collection of beans and a *Deployment Descriptor* (*DD*), used for configuration purposes. For EJB, the deployment stage of the bean lifecycle is the latest time when configuration is possible. Moreover, supervision of runtime behavior of bean instances is not addressed. Beyond the basic component model, the EJB standard covers the definition of different facilities, e.g. for naming or persistence management. These aspects and a discussion of how the behavior of beans can be controlled, are left out for brevity.

3.2 Autonomic Computing

The view on a managed system in the context of AC is presented in figure 1.

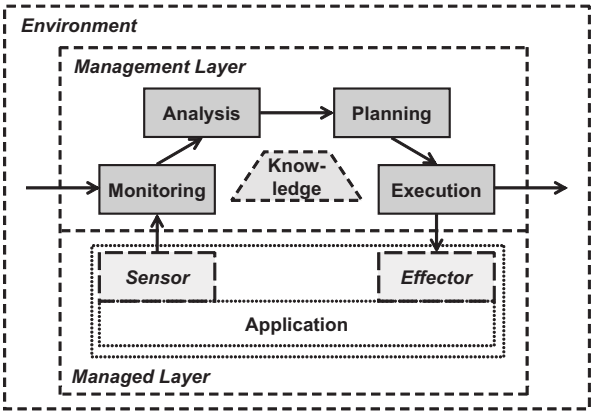

Figure 1. View on a managed system

It basically consists of three main areas. The first area is called *Managed Layer*. It represents the managed application itself, including those parts of the system providing the original *Application*. The *Management Layer* is responsible for administrating the *Managed Layer*. In the context of CO, this mainly covers aspects of analyzing the structure and behavior of the interconnected components, as well as the initiation of appropriate actions for system adjustment, e.g. manipulating the connection structure. Finally, the *Environment* includes entities directly or indirectly influencing the software system but not being part of it. The aspects of autonomous management of application systems are discussed in literature in the context of the *self*-property of autonomic systems [14, 16], consisting of *self-configuration*, *self-optimization*, *self-protection* and *self-healing*. They basically refer to the main properties of autonomic systems as stated in [11] which is fundamental for the vision of AC.

For performing the autonomous management of a software system, the vision of AC is based on the concept of the *Control Loop* [6]. This loop consists of four main stages as shown in figure 1 inside the *Management Layer*. The first stage (*Monitoring*) addresses information collection, laying the foundation for the subsequent stages. Because of the various aspects of autonomous management, there might be different kinds of information relevant regarding the managed system itself as well as its environment. The figure does not imply any type of information provision. There are aspects conceivable where a *push oriented* approach is appropriate, e.g. to inform the *Managed Layer* about the occurrence of an event, while others demand for *pull oriented* information collection, e.g. for structural inspection. The collected information is processed by the *Analysis* stage. It is responsible for identification of situations demanding for reaction of the *Management Layer*. These might span from identifying a single exception, directly provided by the *Monitoring* stage, to complex evaluation and aggregation tasks, e.g. deducing workload shifts or identifying attack patterns. If the need for re-configuration is identified, the *Planning* stage is addressed. It is responsible for the generation of reaction plans, leading to better fulfillment of the overall goals of the managed system. A broad variety of reactions is conceivable, spanning from fine grained and isolated reactions, e.g. preventing a malicious call from reaching the managed application, to complex re-configurations, including a re-organization of the interconnection structure. Finally, the generated plans must be realized, which is the task of the *Execution* stage. Internally, the different stages might make use of different types of internal *Knowledge*, e.g. for validation of alternatives during *Planning* against the overall goals. The control loop concept does not imply an unidirectional flow through its stages. It might e.g. be necessary to request additional information from the *Monitoring* stage during *Planning* or to perform an *Analysis* of the effects of re-configuration during *Execution*.

3.3 mKernel overview

With the *mKernel* system [4], a generic infrastructure for the *Managed Layer* of EJB based systems is provided. For being manageable by the *Management Layer*, the system provides appropriate facilities for information discovery and manipulation. Those are included in figure 1 as *Sensor*s and *Effector*s. Consequently, the provided facilities determine the opportunities for autonomous management. While *Sensor*s set up the space for system observation, *Effector*s define options for re-configuration. The internals of *mKernel* are discussed in [4] regarding their realization. *mKernel* basically consists of a preprocessing tool for EJB modules, a set of enterprise beans realizing different facilities for introspection and re-configuration, and an *Application Programming Interface* (*API*) providing comprehensive *Sensor*s and *Effector*s while abstracting from the realizing facilities. An overview of the constituting parts of *mKernel* is provided in figure 2.

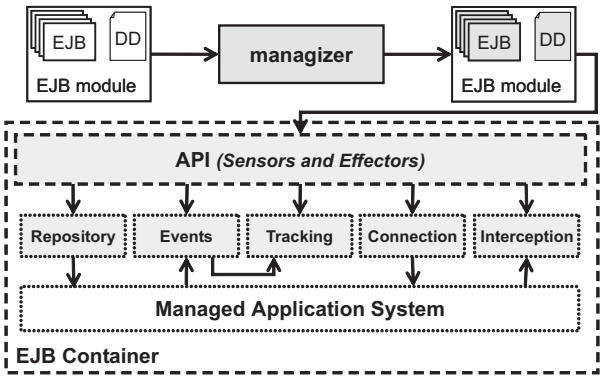

Figure 2. Parts of the *mKernel* system

The preprocessing tool, called *managizer*, accepts a standard compliant EJB module and enriches it for autonomous management. Preprocessing is executed without any need for additional information provision or intervention. Developers do not even have to consider the management of their beans during implementation or follow special guidelines for allowing manageability, because the *managizer* solely relies on requirements of the EJB standard for analysis and extension of EJB modules. As result, a standard compliant EJB module is generated, ready for integration into an *mKernel* managed system. A set of beans establishes the basic infrastructure for autonomous management. They were realized based on the *Glassfish Application Server* [1] which provides, amongst others, a standard compliant EJB container. The beans can be used without any extension of the application server implementation being necessary. This infrastructure can be divided into five major facilities:

Repository: The repository facility represents the interface for integration and removal of EJB modules. It accepts preprocessed modules and constructs inspectable representations of them. Modules can be configured and deployed into the managed container. If not needed anymore, the repository also allows their undeployment and complete removal.

Events: For observation of a system, the event facility provides *push oriented* sensors. Aspects addressed reach from the integration and removal of EJB archives performed through the repository facility, over events for state transition of EJB modules during their deployment lifecycle down to lifecycle observation of bean instances.

Tracking: Based on the events facility, the tracking facility analyzes behavioral aspects of a managed system. It captures events and generates a comprehensive image regarding the lifecycle of bean instances and interactions among them, including the opportunity to analyze call chains.

Connection: This facility provides effectors for reconfiguration of connections among deployed beans. It allows for connecting declared receptacles of beans to interfaces of session beans, also covering *Dependency Injection*. Moreover, it is possible to re-route names of the container managed namespace to new targets.

Interception: The EJB standard includes an interception facility. Its configuration for beans is limited to the deployment time. In contrast, this facility of *mKernel* provides the opportunity to re-route method calls arriving at bean instances through interceptors realized as session beans. The set of these session beans can be manipulated at any given time during the deployment lifecycle of the affected beans.

4 System Model

According to the EJB standard, instances of session beans can directly be referenced by clients. In case a certain EJB module should be replaced by another one, the original module must be undeployed first. Afterwards, the new module can be deployed into the container, binding the included session beans to the corresponding mapped names of the replaced session beans. This leads to situations where the affected mapped names are temporary unbound. To allow a seamless re-configuration, *mKernel* divides the EJB modules, deployed in a container, into two layers. The *Access Layer* consists of modules, providing beans directly accessible by external clients. They act as proxies, provide no application logic but only forward method invocations to beans of the second layer, the *Managed Layer*. Those are the providers of the original application logic. In case a module of the *Managed Layer* should be replaced, the new module can be deployed into the target container, binding the included beans to arbitrary mapped names and performing all configuration. Afterwards, the affected proxies of the *Access Layer* can be instructed to forward requests to instances of the replacing session beans. Finally, the replaced module can be removed from the system. The same holds for re-configurations affecting the *Managed Layer* internally, because *mKernel* has full control over connection establishment among managed beans. Through the separation of layers, *mKernel* provides the opportunity to seamlessly re-configure the managed system. Additionally, *mKernel* provides two types of re-routing. The default policy of re-routing is to also replace established connections. It might be feasible, if the re-configuration should immediately take effect. If *lazy re-routing* is performed, existing connections are used further on. Only for newly established connections, the new targets are chosen. This type of re-routing does not loose state and provides the opportunity to end up existing sessions.

Elements of a system are considered at three different levels, namely *Type Level*, *Deployment Level* and *Instance Level*. The API of *mKernel* provides model based access to the *Managed Layer* of a system by means of direct inspection and manipulation of representations of its elements. The figures, being part of the following discussion, provide simplified overviews over the different levels. They only cover the relevant elements due to clearness reasons.

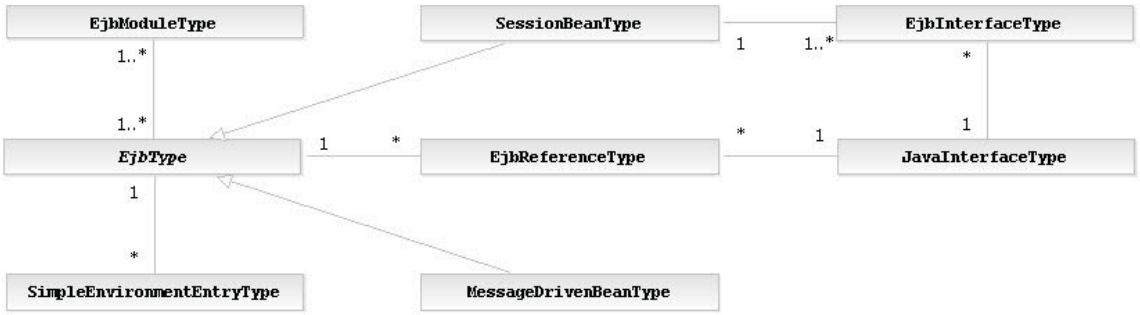

Figure 3. *Type Level* **view**

Moreover, they only include state aspects relevant for the discussion. For the elements at each level, the corresponding elements of the neighboring levels are reachable through associations which are not depicted in the figures.

4.1 Type Level

The *Type Level* addresses information regarding types of constituting elements of a managed system. The covered elements correspond to artifacts being the result of development. Figure 3 presents an overview over the *Type Level* model as provided by the API. After finishing development of a component, the corresponding *Java Archive* (*JAR*) is – after being preprocessed by the *managizer* – integrated into an *mKernel* based system through the API. It is represented as `EjbModuleType` containing at least one `EjbType` which correspond to a `class`-file of a single bean. Due to the fact that one and the same class file can be integrated into more than one archive, the corresponding cardinality is `1..*`. For indicating opportunities to configure an `EjbType` during deployment through parameters, the `SimpleEnvironmentEntry-Types` are used, covering all necessary information. In line with the EJB standard, an `EjbType` can either be a `SessionBeanType` or a `MessageDrivenBean-Type`. For both types, needed references to implementations of certain `JavaInterfaceTypes` can be declared indirectly through `EjbReferenceTypes`. At *Type Level*, those are used to indicate configuration demand in case a bean of the corresponding type is planned to be used within a system. `SessionBeanTypes` provide their functionality through `EjbInterfaceTypes` based on `Java-InterfaceTypes`, too. Note that `JavaInterface-Types` are only considered by *mKernel* if they are connected to at least one `EjbReferenceType` or `Ejb-InterfaceType`. For identification of `EjbModule-Types`, `EjbTypes` and `JavaInterfaceTypes`, hash values of the underlying files are used. Through this proceeding, errors based on naming conflicts are avoided, e.g. if the same name for a bean class is used for more than one implementation.

4.2 Deployment Level

While options for configuration are addressed at *Type Level*, the *Deployment Level* concentrates on the concrete configuration of the managed system. The corresponding elements of the API are shown in figure 4. An `EjbModule` represents a deployed instance of an `EjbModuleType` from *Type Level*. The included `EnterpriseBeans`, and the corresponding specializations respectively, are accessible through a *`mappedName`* inside the namespace of the underlying container. Concrete *`value`*s of parameters of beans are accessible through `SimpleEnvironment-Entrys`. The association between `EjbInterface` and `EjbReference` represents a concrete binding between the requestor of a certain `JavaInterfaceType` and a provider. At *Type Level* it can be evaluated, if an `Ejb-Interface` is matching the demand of a certain `Ejb-Reference`. Not every provided interface must actually be used. The other way round, there need not exist a connection to an interface for each reference, although this might be an indicator for a faulty configuration. Note that at *Deployment Level* each bean is bound to exactly one `Ejb-Module` of which it is an integral part. Nevertheless, it is possible, that a certain `EjbModuleType` is deployed more than once or that an `EjbType` is deployed as part of different `EjbModules`. During its lifecycle an `EjbModule` can pass through different deployment *`state`*s:
`EXISTS`: The module is not integrated into the container, but only exists as representation inside *mKernel*.
`STOPPED`: The module is deployed in the underlying container, but the constituting beans are not yet accessible.
`STARTED`: The included beans are accessible for clients.
While the last two states are deduced from the *JSR88* [7] which builds the technical foundation for deployment level configuration performed by *mKernel*, the first state is special to *mKernel* for provision of additional configuration options as described in section 5.

Each `EjbReference` has a corresponding *`state`* for indicating its usability:
`DISCONNECTED`: A disconnected reference is not associated with an `EjbInterface`.

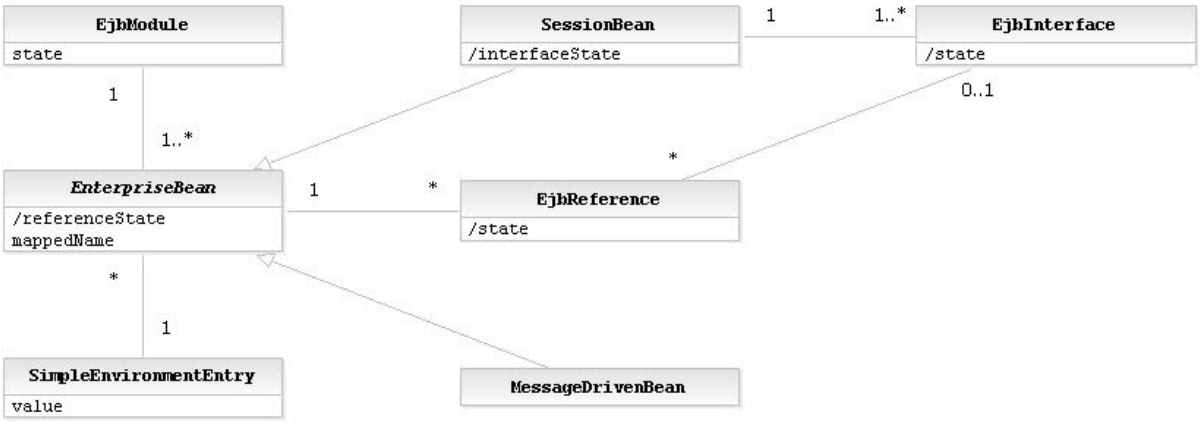

Figure 4. *Deployment Level* **view**

CONNECTED: This state is given, if the reference is associated with an EjbInterface, but the corresponding session bean can not be used properly. This might e.g. be the case, if it is itself missing connections for references.
ACTIVATABLE: An activatable reference is associated with an EjbInterface, and the EjbModule, the reference belongs to, is in state STOPPED or STARTED. Additionally, all EjbReferences of the session bean belonging to the associated interface are either in state ACTIVATABLE or ACCESSIBLE recursively, if any. Finally, at least one module being part of the transitive closure, given through the reference-interface-connections, is in state STOPPED. Consequently, the usability of the corresponding reference can be established solely through starting a collection of modules.
ACCESSIBLE: For this state the same conditions hold as for ACTIVATABLE, but all affected modules are in state STARTED. The reference can be used directly.

Each enterprise bean provides an aggregated *referenceState* which is derived from the states of all of its required references. According to the order of reference states from above, each state is analyzed if at least one reference is in it. If this is the case, the corresponding state is given. Consequently, the aggregated state indicates the most evident need for action for making the corresponding bean usable.

Analogue to EjbReferences, each EjbInterface has a corresponding referencing *state*:
NOT_REFERENCED: No EjbReference is connected to the interface. Regarding the interface, changing the state of the corresponding module would not have any effects on other modules.
PASSIVELY_REFERENCED: Only references are connected to the interface of which the corresponding modules are in state STOPPED. Changing the state of the module, the interface belongs to, would only have indirect effects, e.g. at least one EjbReference would become DISCONNECTED in case of removal. Because of the affected modules not being in state STARTED, this would not have any effect on accessible beans.
REFERENCED: At least one reference is connected to the interface of which the corresponding module is in state STARTED. Therefore, a state transition of the module, the considered interface belongs to, would have direct impact on beans accessible by clients. The same holds for changes regarding the references of the corresponding session bean.

Session beans grant access to a derived *interfaceState*. It represents an aggregated state over all provided interfaces. Its calculation is analogue to the way the *referenceState* of beans is derived. It allows for analysis regarding the most evident impact of changes affecting the session bean either through state transition of the corresponding module or through changes of references the session bean requires.

4.3 Instance Level

At *Instance Level* the instances of beans are considered regarding interactions among each other. Figure 5 covers the relevant elements. An EnterpriseBeanInstance represents a concrete instance of an EnterpriseBean from the *Deployment Level*. Interactions bean instances take part in, are represented by Calls which correspond to method invocations. State transitions during the lifecycle of a bean instance are accompanied by LifecycleCalls performed by the container. Therefore, the lifecycle *state* of an instance can be deduced from the observed LifecycleCalls. For detailed information regarding the different states and the corresponding transitions, we refer to [5]. The arrival of a message at a MessageDrivenBeanInstance is represented by an instance of MessageCall, while BusinessCalls represent the invocation of a certain method on a SessionBeanInstance. During the execution of an arbitrary

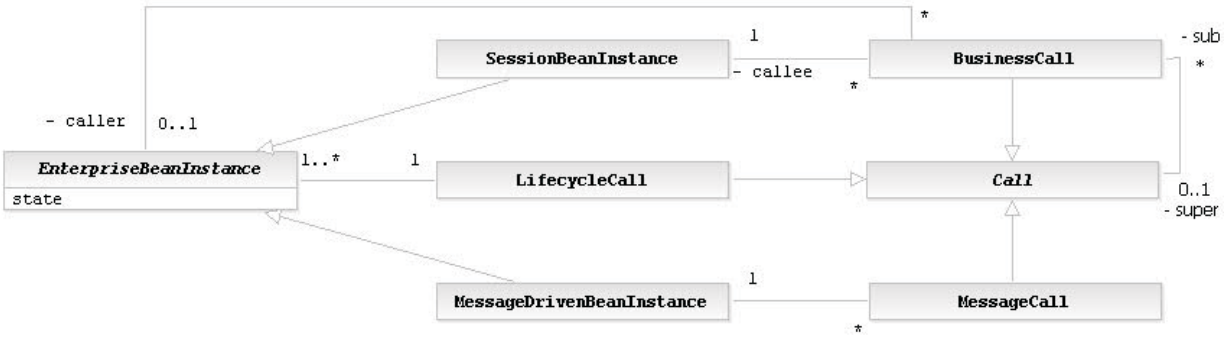

Figure 5. *Instance Level* **view**

`Call`, `BusinessCalls` on `SessionBeanInstances` might be invoked. This is represented through the corresponding association. In this context, the invocation of a *`sub`* call was performed during the execution of a *`super`* call. This leads to the opportunity to analyze call chains inside the *Managed Layer*, spanning multiple bean instances. Moreover, all calls on a given instance are ordered according to their arrival. This allows for inspection of the history of calls executed by a certain instance. In combination, complex interaction scenarios can be analyzed through navigation along call chains as well as through the call histories of the participating instances. Note that bean instances are non-reentrant by definition. Additionally, thread management for beans is prohibited by the EJB standard. Therefore, the aforementioned analysis is facilitated.

5 Lifecycle Administration

At *Type Level*, `EjbModuleTypes` can be created through uploading the corresponding archive through the API. A module type can only be removed from the system, if there exists no `EjbModule` derived from the type. This is because otherwise, there would be a missing link to the type of a considered module and the included elements. At *Deployment Level*, an `EjbModule` can be created from an `EjbModuleType`. Based on the state of its lifecycle, different operations can be performed.

As discussed in section 4, a module in state `EXISTS` is not deployed in the target container. For the included beans it is possible to set mapped names and values for simple environment entries. An existing module can either be deployed into the container, transferring it to state `STOPPED`, or it can be removed from the system. Summarizing, this state allows for configuration of aspects which can not be changed after deployment.

For beans being part of a `STOPPED` or `STARTED` module, it is possible to set targets for required references. This is equivalent to integrating the module into the system architecture. References can only be connected to beans of modules which are either in state `STOPPED` or `STARTED`, too. Additionally, *mKernel* can be instructed to trace method invocations on instances of a particular bean or of all beans of a certain module. As result, *Instance Level* inspection becomes possible. Moreover, lifecycle events are distributed asynchronously over a *JMS* topic, making them observable in a push-oriented fashion. A stopped module can be transferred to state `EXISTS` or `STARTED` while a started module can only be transferred to state `STOPPED`.

In addition to lifecycle based events at *Instance Level*, it is also possible to catch events for state transitions of elements of the other levels. At *Type Level* the upload or removal of `EjbModuleTypes` are published through a well known *JMS* topic, e.g. to allow monitoring and analysis instances respectively, to identify new opportunities for configuration, or to trigger the revision of information kept outside of *mKernel*. Moreover, each state transition of an `EjbModule` at *Deployment Level* is also published via *JMS*. This provides managing instances with the opportunity to observe configuration changes or to supervise the execution of re-configuration.

It is possible to register *mKernel* based interceptors for instances of a particular bean or for all beans of a certain module if the corresponding module is either in state `STOPPED` or `STARTED`. These can engage in the control flow of method invocations. Therefore, they are equipped with a rich set of opportunities, e.g. to manipulate parameters and return values, to initiate a rollback of a transaction, or to avoid the call from being executed. To intercept method calls, a special interface must be implemented as session bean by the interceptor provider. Afterwards, the session bean must be deployed in the same container as the beans of which invocations should be intercepted. It is possible to intercept methods before they are reaching the target instance or after its execution, or both. Additionally, it is possible to specify if method parameters and return values should be transferred as part of the interception or not. This leaves interceptor providers the freedom to provide specific interceptors for a concrete set of beans or to implement generic ones. Otherwise, if parameters and return values are always submitted, the corresponding classes,

e.g. for transferred data, for all intercepted methods must be in the class path of the interceptor.

Inside the source code of beans or attached interceptors based on the EJB standard, context information can be obtained regarding the current state of method calls, e.g. a unique identifier for the call. Thus, an entry point to the *mKernel* model for a concrete context is provided. This enables the development of applications making use of context information during the execution of method calls.

6 Conclusion and Future Work

With *mKernel*, a generic infrastructure for supporting the autonomous management of EJB based enterprise application systems is given. On the one hand, integration of manageability into components is automated by a tool, freeing developers from the need to address management aspects during implementation. On the other hand, context information from inside the source code of enterprise beans or interceptors can be obtained. This allows for explicitly considering management from inside managed entities, if desired. For inspection and manipulation of an application system, *mKernel* provides an unified API which represents a system in a model based fashion while abstracting from the realizing facilities. A rich set of opportunities for inspection and manipulation is provided, taking platform specific aspects of the EJB standard into account.

The view *mKernel* provides to managing instances focuses on entities of the *Managed Layer*, covering structural and behavioral aspects. While this is sufficient for obtaining information common to the intended application areas, the provision of a facility to enrich the provided model with meta data, specific to different contexts, would be desirable. One advantage would be, to free users from the need to keep their specific data synchronized with *mKernel*. To facilitate the application of *mKernel* for autonomous management, additional tools are planned which e.g. allow the automated execution of the *managizer* as part of the development process. Currently, *mKernel* is evaluated in a project addressing self-healing and self-protection of managed systems. Further projects are planned to identify potential shortcomings and desirable extensions.

References

[1] The Glassfish Application Server. http://glassfish.dev.java.net.

[2] T. Abdellatif and A. Danes. A simple approach to autonomic J2EE servers. In *IEEE International Conference on Self-Organization and Autonomic Systems in Computing and Communications (SOAS'2006)*, Erfurt, Germany, 2006.

[3] S. Bouchenak et al. Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters. In *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 13–24, Washington, DC, USA, IEEE Computer Society, 2005.

[4] J. Bruhn and G. Wirtz. mKernel: A manageable kernel for EJB-based enterprise applications. In *Proceedings of the First International Conference on Autonomic Computing and Communication Systems (Autonomics 2007)*, 2007.

[5] L. DeMichiel and M. Keith. JSR 220: Enterprise JavaBeans, Version 3: EJB Core Contracts and Requirements. http://jcp.org/aboutJava/communityprocess/final/jsr220, 2006.

[6] Y. Diao et al. Self-Managing Systems: A Control Theory Foundation. In *ECBS '05: Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, pages 441–448, Washington, DC, USA, IEEE Computer Society, 2005.

[7] J. Dochez. JSR 88: Java Enterprise Edition 5 Deployment API Specification, Version 1.2. http://jcp.org/aboutJava/communityprocess/mrel/jsr088/index.html, 2006.

[8] X. Dong et al. Autonomia: An Autonomic Computing Environment. In *Proceedings of IEEE International Conference on Performance, Computing, and Communications (IPCC)*, pages 61–68, 2003.

[9] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. In *IBM Systems Journal*, volume 42, pages 5–18. IBM, 2003.

[10] D. Garlan et al. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, volume 37(10), pages 46–54, 2004.

[11] P. Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf, IBM Corporation, 2001.

[12] G. Huang, H. Mei, and Q. Xiang Wang. Towards software architecture at runtime. *SIGSOFT Software Engineering Notes*, volume 28(2), page 8, 2003.

[13] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer Magazine*, volume 36(1), pages 41–50, 2003.

[14] P. Lin, A. MacArthur, and J. Leaney. Defining Autonomic Computing: A Software Engineering Perspective. In *ASWEC '05: Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)*, pages 88–97, Washington, DC, USA, IEEE Computer Society, 2005.

[15] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, volume 38(8):114–117, 1965.

[16] R. Sterritt and D. Bustard. Towards an Autonomic Computing Environment. In *DEXA '03: Proceedings of the 14th International Workshop on Database and Expert Systems Applications*, page 699, Washington, DC, USA, IEEE Computer Society, 2003.

[17] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[18] J. White, D. C. Schmidt, and A. Gokhale. Simplifying the Development of Autonomic Enterprise Java Bean Applications via Model Driven Development. In *Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS / UML 2005)*, 2005.