Space-based Architectures as Abstraction Layer for Distributed Business Applications

Richard Mordinyi, eva Kühn
Space-based Computing Group
Vienna University of Technology
1040 Vienna, Austria
{*rm,ek*}*@complang.tuwien.ac.at*

Alexander Schatten
Complex Systems Desing & Engineering Lab
Vienna University of Technology
1040 Vienna, Austria
{*alexander.schatten*}*@tuwien.ac.at*

***Abstract*—For constantly changing businesses, it is essential that the underlying software architecture is capable of managing agile business processes and meeting future business needs. Decoupling between applications and services in distributed systems is addressed by e.g., service-oriented architectures. On the other hand, applications and its underlying middleware are still tightly coupled with respect to the middleware's architectural style. As a result of the tight coupling middleware adaptations introduced due to e.g., new business requirements can affect the application as well. In this paper we propose the concept of space-based architecture (SBA), that allows decoupling distributed applications with respect to the underlying middleware architecture by combining the characteristics and properties of state-of-the-art middleware architectural styles captured in a simple API. The benefit of our approach is minimal application adaptations in case of changing the underlying middleware architectural style, which allows for more efficient realization of new business requirements.**

***Keywords*-Architectural Styles, Agile Business Requirements, Abstraction, Space-based Computing**

I. Introduction

Business constantly changes. Therefore, software architectures should be able to manage agile business processes and need to have the ability to meet future changes and business needs. It is essential for distributed systems to make use of a flexible and adaptable platform that can respond to new requirements in an efficient way. Consequently, the usage of appropriate architectural styles for the design of software systems is a challenge.

A common approach towards creating flexible, dynamic business processes and agile applications is the service-oriented computing paradigm (SOA) [1], [2]. In [3] this kind of paradigm is also referred to as the pipes-and-filters architectural style belonging to the category of dataflow architectures. For instance, the Enterprise Service Bus (ESB) [4] promises to interconnect and route services in a loosely coupled manner for a clear separation of business logic and integration logic. However, an ESB routes service data from one application to another and usually does not keep the history of messages and service interaction, i.e. does not maintain a global state.

Although, software systems are usually not built based on a single architectural style [3], there is a tight coupling [5], [6], [7] between the application and the used style. Thus, middleware adaptations introduced due to e.g., new business requirements (section II) can affect the application as well. However, concerns not related to the application's business logic should be entirely encapsulated in the middleware, which is not possible in case of rearranging its architectural paradigm. This implies that changes to the architecture can result in significant adaptations of applications, thus those concerns have to be considered and implemented in the application as well. Hence, in case e.g., new business requirements cannot be met by the underlying infrastructure, a so called *architecture breaker* has been introduced demanding for costly and time-consuming re-evaluation and changes of the entire architecture. In the same way an *architecture limiter* restricts a simple and efficient solution for new requirements.

In this paper we propose the so called space-based architectural (SBA) style for agile business processes that allows decoupling distributed applications with respect to middleware architectural styles. Our approach combines and includes the characteristics and properties of the major architectural styles found in distributed middleware (section IV) captured in a simple API. SBA can be seen as an abstraction layer between applications and architectural styles, and as such it provides loosely coupling between the applications and the styles, and therefore hides the complexities of distributed systems from the application. Consequently, changes to the architecture can be solved within the architecture and do not affect application implementation. SBA describes an active, coordination model based architectural style. It allows the registration of subscribers like in event-based systems [8] and the concurrent deployment of various coordination models to describe the ways of interaction of the distributed applications. This way SBA is capable of supporting data-driven coordination among clients, while preserving state inside the middleware. Additionally, SBA provides Aspect-oriented programming (AOP) functionality with runtime weaving for deployment

of cross-cutting concerns.

The benefits of the proposed SBA approach allow a) the efficient realization of changing business requirements affecting the underlying architecture; and b) adaptations of the architecture transparent to the application resulting in less complex application logic since it can entirely focus on its business process.

The remainder of this paper is structured as the following: section II pictures the use case, section III defines research questions, section IV summarizes related work, section V describes the concept and the architecture, while section VI discusses the proposed concept. Finally section VII concludes the paper and proposes further work.

II. Scenario

This scenario is based on an insurance company and its agents in field services and demonstrates the terms "architecture breakers" and "architecture limiters". The scenario refers to two examples requiring a change in the architecture due to new business requirements triggering additional adaptations in the applications.

As a starting point let's assume that agents visiting potential customers fill in insurance related forms at the customer. Due to technical and economical reasons the mobile agent needs a permanent connection to the main insurance server of the company, both physically via e.g., UMTS and logically to its services. However, the required permanent connection between the agents and the main server hinders the agents to work efficiently with their customers. The agents cannot be sure whether the transmission capabilities of the provider cover the area where the customer lives, leading to an unreliable customer information management. This brings in a new requirement demanding the agent capability of working offline as well, without being dependable on a permanent connection. However, this leads to a break in the architecture in the sense that data stored before on the main server only, has to be partially replicated to the agents' mobile devices and persisted there. Therefore, both the server and the application need to manage their own data and need to have the capability of synchronizing data changes.

Another requirement refers to the efficiency of how orders are processed between the insurance company and the agents, and thus introduces an architecture limiter. In order to increase customer satisfaction orders are not propagated to and handled by a single agent only, but by a group of agents. The order refers to a group of agents with the capabilities needed to process the order. The agent who wants to process the order is granted a specific amount of time to do so. In case that time expires the order is automatically reassigned to the group again, giving other agents the opportunity to work with the customer's order. However, this business requirement introduces coordination issues between the agents the architecture has to be capable of dealing with. Since e.g., an ESB implementation is not capable of such time-triggered re-routing and coordination issues, the applications have to be adapted accordingly.

III. Research Questions

In this paper, we propose the concept of space-based architecture (SBA)[1], which allows the combination of several different architectural styles. Based on recent projects with industry partners from telecommunications and on the limitations of traditional middleware technologies with respect to introduction of new business requirements and their affects on client applications, we wanted to investigate a) whether the SBA approach is capable of representing the characteristics of different architectural styles at the same time, b) the advantages and limitations of the proposed approach with respect to changing business requirements, c) how to realize changing business requirements transparent to the participating clients, d) how much complexity can be shifted to the abstraction framework, e) how much affects on client applications still remain in case the underlying architectural style has to be changed, and f) whether the proposed concepts allows decreasing development and migration time by reducing the effort needed to adapt the current system to new business requirements and therefore saving costs while improving adaptability and re-usability.

IV. Related Work

In distributed systems there are many architectural styles to be chosen. Distributed middleware are mostly based on either dataflow style, such as pipes-and-filters, on data-centered style, i.e. a repository, or on implicit invocations, like publish-subscribe or event-based [3], [9].

A. Dataflow Architectural Style

Pipes-and-filters, representing the dataflow style, define independent components (filters) that can be connected with each other but which do not know about the existence of other filters [10]. The connections between filters determine the pipeline. Sharing data between filters is only possible by passing it from one filter to the next, even if it is not needed in an intermediary step. SOA [2] typically makes use of the pipes-and-filters style. Services can be implemented as filters and the way of routing messages determines the pipeline that represents the business logic. The ESB [4] is the major platform used in SOA offering the necessary functionality in order to make use of SOA. The ESB discards any service-relevant data after message delivery. Thus, it cannot offer a shared repository that clients can use in order to coordinate themselves.

[1] an implementation demonstrating the concept of SBA can be downloaded at http://www.mozartspaces.org

B. Data-centered Architectural Style

The essence of data-centered styles is that multiple components have access to the same data store, and communicate through that data store. A shared repository does provide its clients with access to shared data. Databases are the typical representation of this data-centered architectural style. They support data distribution and therefore allow their clients to coordinate processing of distributed shared data. Active repositories tie together the shared repository with another architectural style, which are event-based systems [8]. An active repository is able to notify registered clients about changes [10]. A repository does not provide the means for specifying in which order its shared data needs to be processed by its clients. Thus, repositories cannot offer routing capabilities in order to determine the processing sequence among its clients. Thus, it is irrelevant for the usage in pipes-and-filters.

Another data-centered architectural style is the blackboard based one, in which the state of the information on the blackboard determines the order of execution. A representative of the style is e.g., the Linda coordination model by David Gelernter [11]. It describes the usage of a logically shared memory, called tuple space, by means of simple operations (out, in, rd, eval) as a communication mechanism for parallel and distributed processes. In principle, the tuple space is a bag containing tuples with read/write access. Unlike Linda, SBA allows e.g., storing shared data in a customizable structured way (section V). This facilitates the efficient implementation of coordination concerns among middleware clients. Compared with other Linda implementations, which completely rely on tuple matching only, this concept allows the efficient realization of more complex coordination patterns [12]. Besides its inherent shared repository nature, SBA also supports the pipes-and-filters style (section V-B) by using the concept of containers. Containers would represent the filter components that can be interconnected [13] in order to define the pipeline.

In [14] an extension to the pipes-and-filters style was proposed, where a shared repository is also supported. However, the hybrid framework does not offer the abstraction of the pipes-and-filters style but rather adds shared data to the pipeline.

C. Implicit Invocation Architectural Style

This style is characterized by calls that are invoked indirectly and implicitly as a response to a notification or an event. The group is represented by the publish/subscribe [15] and event-based [16] architectural styles.

V. Architecture

This section pictures the idea of space-based architecture in detail. It describes the components, interfaces, supported operations, the various ways of executing operations, and how architectural styles are represented.

A. Space-based Architecture Framework

In contrast to the original Linda model, the SBA architecture consists of Internet addressable containers [12]. A set of containers span the so called space. In the first place, a container is a collection of entries accessible via a basic simple API. The difference to Linda is that a container may be bounded to a maximum number of entries, and allows the usage of so called coordinators with each having its specific and optimized view on the stored entries. Figure 1 shows the basic components [12] of a container.

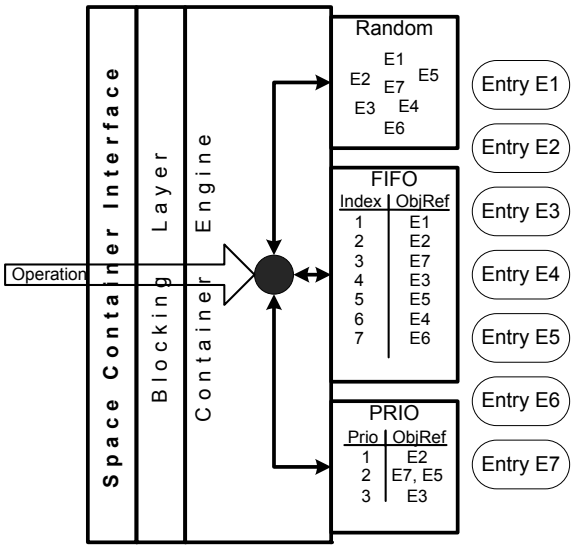

Figure 1. The main components of a container.

The data items that are stored in a container are called entries. An entry can be either of type Tuple or of type AtomicEntry. A Tuple contains other Entries, which can be either AtomicEntries or other Tuples. An AtomicEntry is a Generic Java class, so when it is instantiated, the class that is contained within the AtomicEntry can be defined. An Entry may also be a reference to other containers.

Coordinators are the programmable parts of the container and are responsible for managing their view on the entries in the container. The aim of a coordinator is to represent a coordination model and to structure and organize the entries in the container for efficient access. Each coordinator has its own internal data structures which help him to perform its task. Since the coordination model to be realized is known beforehand, the coordinator can be implemented in an efficient way. Additionally, any number of coordinators can be added to a container, but it has to have at least one.

The container's interface provides a simple API for reading, taking, and writing entries, but extends the original Linda API with the methods `destroy`, `shift` and `notify`. Destroy removes an entry from the container, while shift writes an entry after it has removed one. Containers support bulk operations as well, so that it is possible

to insert multiple entries into a container resp. to read/take multiple entries out of it within one operation. The number of entries to be retrieved or to be written is specified in the so called selector that is used for the operation.

For every available coordinator there is a certain selector that represents the counterpart to this coordinator. Selectors contain parameters (like a counter for the minimum number of entries to be retrieved) for queries in case of a read, take, destroy access. In case of writing entries they contain a) the parameters specifying the appropriate coordinators and influencing it with special values, and b) the entry to be added to the container.

Comparing the concept of containers with databases, the drawback of databases is that they need a static data model of the entries they have to store, while containers allow the usage of several different coordinators at the same time, enabling dynamic data models, and thus being schema-free. In case of db4o[2], accessing an entry is performed via query-by-example, like in Linda. However, in [12] it has been shown that containers allow an optimized realization of queries and coordination models.

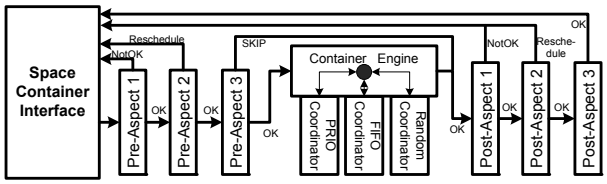

Figure 2. Pre- and post-aspects of a container.

Another component of a container are aspects [17] (Figure 2), realizing some parts of Aspect-oriented Programming (AOP) [18] by registering them at different points of a container. Aspects represent additional computational logic and are executed on the peer where the container is located. Aspects are triggered by operations either on a specific container or on operations related to the entire set of containers, the space, rather on the according impact. Aspects can be located before or after the execution of an operation, indicating two categories: pre and post. Aspects are executed sequentially in the same order in which they where added. Adding and removing aspects can be performed at any time during runtime.

Figure 2 shows a container with three pre and three post aspects installed. The accessing operation is passed immediately to the first pre-Aspect, before it is executed on the container. The operation contains the parameters of the operation, like transactions, selectors and timeout. If all aspects return *OK*, the container implementations interprets the selectors of the operations and executes the operation [12]. Afterwards, all post-Aspects are executed. Beside an *OK*, an aspect may return different values changing the container operation accordingly. In case of *NotOK* the execution of the operation is stopped and the transaction is rolled back. In case of *SKIP* the operation is neither performed on the container, nor on any following pre-Aspects. The post-Aspects are executed immediately afterwards. The return value *Reschedule* indicates to stop the execution of the operation and reschedule it for a later execution.

Containers are Internet addressable using an URI of the form "xvsm://mycomputer.mydomain.com:1234/ContainerName". The protocol type *"xvsm"* makes the possible communication protocols transparent to the application [19]. Depending on the application domain, or the underlying network infrastructure, *"xvsm"* may be translated to e.g. tcp+java, specifying that communication takes place via a tcp-connection using java objects. The default communication is based on an XML based protocol that defines all operations in a platform neutral way. A lookup mechanism (e.g., DHTs [13]) resolves published container names to its URL. In [13] it has also been explained how to replicate containers transparent to the application in order to achieve load-balancing and increased fault-tolerance.

B. Architectural Styles realized with SBA

This section describes how SBA is capable of combining several different architectural styles, such that the complexity remains in the middleware rather than in the application. Figure 3 shows the approach for decoupling application from architecture style in distributed systems. It proposes to separate the application itself into an application layer and an architecture layer representing the architectural style. Between these layers a new layer the space-based architecture is introduced as an intermediate layer that is able to combine and switch between different software architecture styles transparent to the application layer.

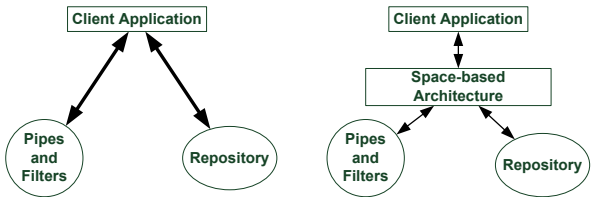

Figure 3. SBA as an abstraction framework for architectural styles for reducing complexity in client applications.

As shown in Figure 4 the SBA concept combines the characteristics of different architectural styles on several layers. Each layer is responsible for a specific task and as such it is capable of representing a specific architectural style on its own or in combination with another layer.

1) SBA resource representation: This layer (Layer 1) is the one that can be used by the client applications. Here, the concept of organizing data in containers by means of coordinators and the various container access methods

[2]http://www.db4o.com/

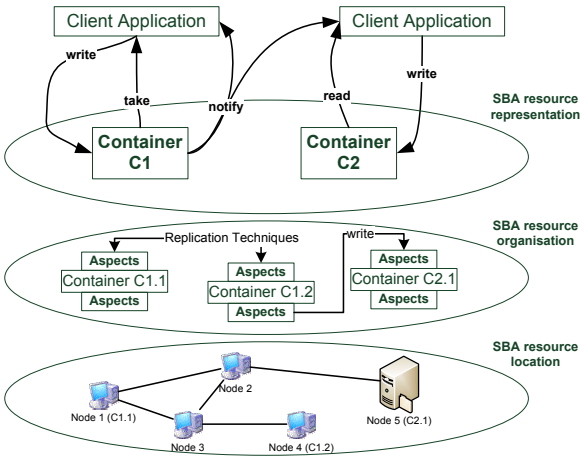

Figure 4. Layering of SBA competence.

(especially blocking operations [11]) are the most important ones. SBA allows customized coordination that determines the access to the data. This kind of architectural style is inherent, since the origins of SBA come from the Linda model. However, by means of coordinators and selectors, data stored in a container can be stored and queried for in an efficient way taking into account domain specific requirements. The main difference to other repository like databases is that it can block operations allowing queries for future data states.

For the application this layer looks like a virtual server with a structured data store, already fulfilling the requirements for a data-centered architectural style. The application itself needs to focus on its business aim only, i.e. needs to specify the coordination models needed within the distributed applications derived from the business aim.

2) SBA resource organisation: This layer (Layer 2) introduces aspect capabilities, which can be added and removed transparent to client applications. In order to fulfill the requirements of the pipes-and-filters architectural style, it is not enough to deploy a container using a FIFO coordinator at Layer 1. The containers in the chain have to be interconnected, which can be done by aspects. The FIFO coordinator makes sure that a container behaves like a queue. Thus, the container is the abstraction for a filter manipulating a message in a service pipeline. Pre- and post aspects on those containers can be used to determine message routing.

As stated in [20] aspects can also be used to notify applications. There, various ways of how applications can be notified and how containers have to be interconnected is explained. This implies the representation of the blackboard architectural style, but also allows to support publish/subscribe and event-based architectural styles.

3) SBA resource location: Layer 1 hides the fact where a container is physically stored in the network. From an client application point of view it is not known whether the container is stored on a single server or in a P2P network. Therefore, similar to an ESB, SBA allows to migrate a single container executed on a server to a replicated one deployed in a P2P network transparent to the client application. In such case, aspects in Layer 2 would define the replication technique between the distributed containers. How replicated containers in a P2P network are realized is explained in [13] and [21].

VI. Discussion

In this paper, we propose the concept of space-based architecture (SBA) as an technology in order to combine different architectural styles. The SBA approach is capable of representing the characteristics of different architectural styles at the same time because it offers a generic interface, the concept of containers and aspects. Therefore, e.g. writing data to a container allows the simultaneous use of data flow styles as well as data-centered styles.

The advantages of the proposed approach with respect to changing business requirements are that the applications do not have to consider the underlying architectural style. However, an abstraction technology placed between middleware and client application causes an additional overhead which affects overall performance. Changing business requirements as described in the scenario can be applied transparently to participating clients because they communicate via the same interface with the middleware. Thus, complexity can be shifted to the abstraction framework by adding aspects to the middleware that implement the requirement details. However, client applications still might have to be adapted if new requirements do not include distribution details only but also business process changes. Nevertheless, the proposed concept allows decreasing development and migration time by reducing the effort needed to adapt the current system to new business requirements and therefore it saves costs while improving adaptability and re-usability. Summing up, we take a look at the quality attributes defined in [22]: Performance is decreased due to the additional layer. Security is supported transparently to client applications by adding security-relevant aspects to the SBA middleware. A space can be transparently replicated in order to improve availability and fault tolerance. According usability, SBA offers a generic interface. Components always act upon the same interface, which improves modifiability, modularization and encapsulation. By abstracting the underlying architectural style the SBA can be ported to many different middleware technologies, such as inter-process communication (e.g. RMI), SOA, etc. SBA supports loose coupling facilitating re-usability. Since client applications do not need to be adapted, they can be tested in the same manner which improves testability. In [23] several properties have been compared in the implementation of a use case that requires an algorithm to perform circular word shifts of a given input. This was implemented on one hand with a shared repository

and on the other hand with pipes and filters. The conclusion was that changes of algorithm and functionality as well as re-usability are better supported with the pipes-and-filters style. Direct data access was better supported by the shared repository. SBA as an abstraction framework is able to offer all these advantages to its applications.

VII. Conclusion and Future Work

In this paper, we described the concept of Space-based architecture as an abstraction framework for middleware architectural styles in order to allow the realization of new business requirements with minimal affect on client applications. We derived research questions and answered them based on a scenario from the insurance domain.

Based on the components SBA provides, it is capable of simultaneously support and behave like any of the major architectural styles of middleware, like pipes-and-filters (e.g. as can be found in SOA) and shared repositories (e.g. a DB).

The benefits of the approach are a decoupling between applications and architectural styles allowing changes in the architecture with minimal effect on the client, resulting in less testing of client implementations and therefore minimized time-to-market.

This paper is meant to be an introduction into a series of architectural changes of distributed systems. Further work will include an investigation of independent component-, virtual machine-, and call-and-return architectural styles with respect to possible ways of representing their features by means of SBA components. Another open issue is benchmarking of the framework, i.e. to what extent does the additional abstraction layer decrease computational performance. A more comprehensive evaluation with respect to testing and development time is intended. Finally, the possibility of conflicting architectural styles and its effects on complexity in client applications has to be evaluated.

VIII. Acknowledgement

We would like to thank Thomas Frühbeck from Telekom Austria and Stefan Biffl for the valuable discussions on this topic. This work has been partially funded by the Complex Systems Design & Engineering Lab, Vienna University of Technology (http://www.informatik.tuwien.ac.at/csde).

References

[1] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

[2] M. P. Papazoglou and W.-J. Heuvel, "Service oriented architectures: approaches, technologies and research issues," *The VLDB Journal*, vol. 16, no. 3, pp. 389–415, 2007.

[3] S. Dustdar, H. Gall, and M. Hauswirth, *Software Architekuren für Verteilte Systeme*. Springer-Verlag, 2003.

[4] D. Chappell, *Enterprise Service Bus*. O'Reilly Media, Inc., 2004.

[5] N. Medvidovic, "On the role of middleware in architecture-based software development," in *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*. New York, NY, USA: ACM, 2002, pp. 299–306.

[6] L. Aldred, W. M. van der Aalst, M. Dumas, and A. H. ter Hofstede, "On the notion of coupling in communication middleware," pp. 1015–1033, 2005. [Online]. Available: http://dx.doi.org/10.1007/11575801_6

[7] H. Xiao, J. Guo, and Y. Zou, "Supporting change impact analysis for service oriented business applications," in *SDSOA '07: Proceedings of the International Workshop on Systems Development in SOA Environments*. Washington, DC, USA: IEEE Computer Society, 2007, p. 6.

[8] A. Carzaniga, E. Di Nitto, D. S. Rosenblum, and A. L. Wolf, "Issues in supporting event-based architectural styles," in *ISAW '98: Proceedings of the third international workshop on Software architecture*. New York, NY, USA: ACM, 1998, pp. 17–20.

[9] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[10] P. Avgeriou and U. Zdun, "Architectural patterns revisited - a pattern language," in *Proc. Of 10th European Conference on Pattern Languages of Programs (EuroPLoP 2005)*, 2005.

[11] D. Gelernter, "Generative communication in linda," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 80–112, 1985.

[12] E. Kühn, R. Mordinyi, L. Keszthelyi, and C. Schreiber, "Introducing the concept of customizable structured spaces for agent coordination in the production automation domain," in *AAMAS '09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 625–632.

[13] E. Kühn, R. Mordinyi, H.-D. Goiss, S. Bessler, and S. Tomic, "A p2p network of space containers for efficient management of spatial-temporal data in intelligent transportation scenarios," *Parallel and Distributed Computing, International Symposium on*, vol. 0, pp. 218–225, 2009.

[14] A. R. Franois, "Software architecture for computer vision: Beyond pipes and filters," Technical Report IRIS-03-240, Institute for Robotics and Intelligent Systems, USC, Tech. Rep., 2003.

[15] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[16] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[17] E. Kühn, R. Mordinyi, L. Keszthelyi, C. Schreiber, S. Bessler, and S. Tomic, "Aspect-oriented space containers for efficient publish/subscribe scenarios in intelligent transportation systems," *The 11th International Symposium on Distributed Objects, Middleware, and Applications (DOA'09)*, 2009.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," pp. 220–242, 1997. [Online]. Available: http://dx.doi.org/10.1007/BFb0053381

[19] R. Mordinyi, T. Moser, E. Khn, S. Biffl, and A. Mikula, "Foundations for a model-driven integration of business services in a safety-critical application domain," *Accepted for the Track on SoftwareProcess and Product Improvements (SPPI) at 35th Euromicro Conference Software Engineering and Advanced Applications (SEAA'09)*, 2009.

[20] E. Kühn, R. Mordinyi, and C. Schreiber, "Configurable notifications for event-based systems," Vienna University of Technology, (TechRep at http://tinyurl.com/oht888), Tech. Rep., 2008.

[21] E. Kühn, R. Mordinyi, M. Lang, and A. Selimovic, "Towards zero-delay recovery of agents in production automation systems," *Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM International Conference on*, vol. 2, pp. 307–310, 2009.

[22] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

[23] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.