Architectural Description with Integrated Data Consistency Models

Peter Tabeling
Hasso-Plattner-Institute for Software Systems Engineering
P.O. Box 90 04 60, 14440 Potsdam, Germany
tabeling@hpi.uni-potsdam.de

Abstract

The focus of typical architectural models is the description of large systems. Even though these systems are usually distributed, aspects of distributed systems are only addressed in a rudimentary way. While typical approaches pay attention to technical issues like deployment, data consistency problems are mostly ignored. This paper presents a modeling approach which integrates assumptions about data consistency into architectural models. The concepts of abstract locations and operations play a central role in this context. They allow transactions and snapshots to be implicitely defined by mapping high-level architectural models to low-level models. The decision about the application of transactional techniques and snapshot algorithms can then be based on architectural models. Furthermore, the approach can be integrated into programming languages and platforms. In this case, a runtime environment could automatically detect and process both transactions and snapshots.

Keywords: Software Architecture, Data Consistency, Transaction, Snapshot, Fundamental Modeling Concepts, FMC, STAGE.

1. Introduction

Architecture oriented methods of describing software systems generally aim at „large“ systems. Beside being heterogenous and complex, such systems are often distributed in some form, e.g. they heavily rely on the multitasking capabilities of the underlying operating system(s) or they are even deployed on a physically distributed hardware platform.

Therefore, an architecture description method should be capable of addressing typical, inherent features of distributed systems. In particular, this applies to architecture descriptions from which design and implementation should be derived in a systematic way, by following defined rules.

1.1. Data Inconsistency in Distributed Systems

A typical feature of distributed systems is the need to deal with various types of „inconsistency“. For example, transactional concepts must be implemented to avoid generating or accessing „inconsistent data“. Another problem is the limited possibility to observe a „consistent global state“. In this context, two basic types of inconsistency should be distinguished:

- *Shared data inconsistency*
  Conflicting accesses (reading, writing, modifying) to shared data by concurrently operating components is a common problem, because these accesses are usually implemented by sets of lower-level accesses. When scheduling these accesses, certain rules must be obeyed in order to avoid „dirty reads“, „unrepeatable reads“ or „lost updates“. Well-known solutions are given, ranging from simple locks to various types of transactions [1][2][3].
- *Distributed data inconsistency*
  A nondistributed sequential system can be modeled as a state machine, relying on the idea of a defined global system state, being observable by the system itself. According to Petri [4], this idea is a questionable assumption if we deal with concurrent, distributed systems. A distributed system shows non-neglectable, rarely predictable latencies when reading „remote“ data. This, in combination with concurrent data changes at different locations, makes it impossible to reliably observe the system‘s global state [5] - only „local“ state can be observed. Again, solutions have been found to address this problem, at least partially. For example, snapshot algorithms allow causal consistent observation of the system state [6][7].

1.2. Typical Focus of Architecture Descriptions

When „software architecture“ and architecture description languages started to be research topics, their main focus soon became clear. As survey papers show, they are aimed at capturing the high-level view(s) of a large system, typically as a structure of „components“ and „connectors“ [8][9][10]. Further typical research topics in the field of software architecture are reference architectures, architectural styles and architectures for product lines.

However, in case of distributed systems, only „technical“ aspects are addressed by architecture oriented descriptions. For example, „deployment diagrams“, as defined in UML [11][12], allow to define hardware nodes as installation targets of „components“ (e.g. database tables, libraries, executables etc.). The „code view“ from [13][14] has a similar focus while the „execution view“ describes the use of processes and other ressources of the (software) platform.

1.3. Open Questions

While deployment and the use of processes are important aspects of distributed systems, data consistency aspects, as described in section 1.1., are rarely addressed by architectural models.

For example, there is not enough information to decide which (part of a) data structure (e.g. a set of shared memory cells) must be accessed in a transactional manner (e.g. by applying locking mechanisms). Answering this question requires explicit information about which data elements form such a data structure, and, if conflicting accesses to it are actually possible.

Another problem is setting a transaction‘s boundary (or identifying a critical section), i.e. to combine the „right“ set of accesses to a data structure in order to be scheduled adequately. Obviously, information is required for grouping accesses.

As already mentioned, the general possibility to observe the „global“ state of a system must not be assumed in case of distributed systems. Reliable state observation is limited to „local data“ while reading „distributed data“ generally results in a „snapshot“ reflecting a state which has possibly never been valid. The latter case is often tolerable, but sometimes causal consistency is required, at least. In this context, some means are neccessary to distinguish between „local“ and „distributed“ data, and, to define the required type of consistency of observed data.

In general, a description method seems desirable which covers data consistency constraints as an integrated aspect of architectural models.

2. Binding Consistency Constraints to Architectural Models

The modeling approach presented in the following primarily aims at the conceptual view of a system‘s architecture. At this level, it allows to describe the „conceptual distribution“ of the system as a structure of abstract active components and abstract locations. These locations represent passive system components which allow the active components to exchange and store data. Additional architectural models describe the mapping of these components to lower-level components and the refinement of high-level operations at lower levels. Because certain consistency constraints generally apply to the different elements of a model, the models and their interdependencies carry the information to answer the questions from section 1.3..

The approach discussed here is based on the Fundamental Modeling Concepts (FMC) [15][16][17], with several extensions to address model mapping and consistency aspects [20][21][22].

2.1. Basic Architectural Structures

As a very basic idea, FMC supports the description of three elementary types of structures within each architectural model:

- *Compositional structure*
  This reflects the (mostly) static structure of the system, consisting of active and passive components, called agents and locations.
- *Behavioral structure*
  Describes the behavior of the agents and the observable dynamics caused by it.
- *Value structures*
  All locations in the compositional structure carry data items. A data item can be a simple, unstructured value or a structured value, like an array, a tree or even a database.

FMC offers three distinguished diagram types, one for each of these structures. These diagrams rely on a semiformal notation, being optimized for intuitive understanding by human readers. In the context of this paper, the concepts behind these diagrams and the relationships between them are more important than the notation itself. (Readers interested in the notation should refer to [15] and [17] for the details.)

With respect to the questions in the introduction, the elements of compositional structures and behavioral structures are the most important ones. Hence, they are dealt with in more detail below.

2.2. Compositional Structure Elements

On the background of software systems, architecture is typically seen as a high-level view of the system, represented as a structure of „components“ and „connectors“. While components primarily perform the required processing, connectors may simply transport, filter or transform messages, and they can mediate between components or even control their interaction.

From this point of view, all parts of a system are active in some way. There seem to be no explicit elements for simply carrying state, i.e. containing data. Szyperski even defines components as system parts without “externally observable state“ [18]. State and the corresponding data storages are often treated as implementation details of components, accessible only indirectly via interfaces.

At first sight, hiding data storages behind interfaces seems to be a good or even mandatory way to foster information hiding. However, data storages must be explicit parts of a model if we want to discuss data accesses, observation and consistency.

Compositional structures, as described in FMC, contain data storages at all levels of abstraction. In general, a compositional structure consists of *agents* and *locations*. Agents are *active components* which perform all data processing and transport. At a higher level, an agent can be a purely conceptual component providing a required functionality. At lower levels, an agent can be an object, a process or even a hardware component. An agent is not limited to purely sequential behavior (as, for example “active objects” [19]) but can also perform multiple concurrent activities.

Agents are never connected directly, but via *storages* or *channels*, which are *passive components* of a compositional structure. Storages and channels are called *locations* because they represent physical or conceptual „places“ where data can be stored or exchanged. A non-shared storage is used by only one agent to hold some state, while shared storages (which are connected to different agents) can also be used for communication. Like an agent, a storage can be purely conceptual - in this case, it only represents an abstract container for an also abstract data value. Only at lower level models, a storages‘ implementation in terms of „technical“ storages (like files, a set of memory cells, etc.) is revealed. This approach allows to reason about data accesses and consistency in high-level architectural models (see sections 2.4. and 2.5.) without sacrifying the idea of information hiding.

Channels, in contrast to storages, lack the ability to carry (non-transient) state. They can only be used by agents for communication purposes. Agents can output messages (i.e. transient data) to channels which can, but need not, be received by another agent. If the message is not read by an agent waiting for it, it is lost and the channel becomes „empty“. Again, channels can be purely conceptual or represent low-level components.

Each agent has one or more ports, which define the possible connections to locations. Depending on the number of ports, an agent can be connected to any number of locations. In a similar way, locations can be connected to any number of agents (i.e. their ports), but locations do not have ports.

Figure 1 shows a simple example of a compositional structure with three agents (a1,a2 and a3), three storages (s1,s2 and s3) and three channels (c1,c2 and c3).[1]

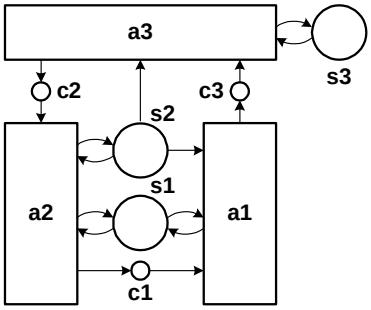

Figure 1. Compositional Structure - Example

2.3. Behavioral Structure Elements

Corresponding to the clear distinction between active and passive components of the compositional structure (see section 2.2.), we distinguish between activities and their effects.

Each activity consists of elementary activities, called operations. When performing an *operation* an agent assigns a new value (the operation result) to a certain location (the operation target). This value is derived from values which are read by the agent from one or more locations. In this way, every agent performs a sequence of operations for each location it uses as output location or state storage. In case of the operation target being a storage, the operation’s result is the new state of the storage. In case of a channel, the operation’s result is the message being sent.

The elementary effects of operations that can be observed at locations are called events. An *event* is a value change taking place at a certain location and a certain point in time. (A value change may also take time - in this case, the “value” “undefined” is given in a time interval, which implies that two events are observable.) This implies that,

1. Storages and channels are always depicted with large and small rounded nodes. Arrow directions indicate if an agent can read or write a location (or both). For simplicity reasons, ports are not shown grafically.

at a single location, there will always be a sequence of events. Of course it is still possible that separate values within a data structure change simultaneously at a single location. Nevertheless, these changes are not to be seen as independent events but as “parts” of one single event. An event is not only outcome of an operation but may also be a trigger for an operation, i.e. an agent can wait for a certain event (or multiple, simultaneous events) before it performs an operation. Such a triggering event can be a certain value change at a shared storage or the arrival of a message at an input channel, for example.

2.4. Data Access, Observation and Consistency

Compositional structure and behavioral structure in combination provide the basis for reasoning about data access and consistency. Locations as explicit elements of a model allow to discuss where data is being observed or produced during system activity.

Hence, an *access* can be defined as follows: For each operation an agent performs and for each location which is observed or affected during that operation, there is exactly one access. An access can be a reading, writing[1] or modifying access.

Depending on the type, accesses (to the same location) may overlap or not. A stand-alone read access may overlap with another stand-alone read access and a stand-alone read access may overlap with a read access being part of a modifying access. All other combinations may not overlap and must be temporally ordered.

Figure 2 illustrates an example operation where agent a1 from Figure 1 performs the operation: s1 := s1+s2. It is assumed that the storages s1 and s2 contain two-dimensional vectors, whose time-dependent values are shown in the diagram. The operation is triggered by a request (“add”) being sent by agent a2 via channel c1. Agent a2 also provides the values of s1 and s2 before triggering the operation. During that operation, agent a1 performs three accesses, namely two read accesses (R_{c1} and R_{s2}) and a modifying access (M_{s1}) - the latter being a combination of a read (R_{s1}) and a write access (W_{s1}).

As discussed in the introduction, behavioral modeling of distributed systems should not rely on the assumption of an observable „global“ state. Instead, observability of state may only be assumed for „local“ state. The modeling approach presented here takes these ideas into account (see [20] and [22] for a detailed discussion).

The definition of accesses reflects the idea that data at a location can be changed „in one single action“ and

1. Producing a message at a channel can be seen as a special type of write access, which includes a second, implicit access for resetting the channel to the “empty” state.

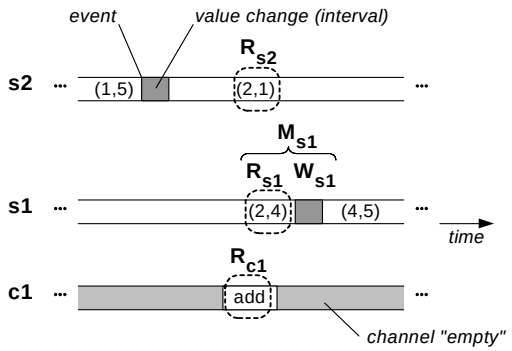

Figure 2. Accesses to Locations (High Level)

observed „at a single glance“, which resembles our intuitive understanding of „being located at one place“. In general, read accesses to a location are considered to guarantee *temporal consistency*:

A sequence of read accesses to a location yields a sequence of observed data. This observation is temporal consistent, i.e. it does not contradict the real sequence of values at the observed location, i.e. the order in which these values have been generated.

However, temporal consistent observation is only guaranteed for data at one location, i.e. „*local* state“ can now be defined as *data being held at one location* (storage or channel).

As an intended consequence, data being read from *different* locations may be a temporal *inconsistent* observation. This is also true for data which is read during *one* operation, because each agent must be considered as an internal observer of the system. However, an operation - as an elementary activity - should at least guarantee *causal consistency*:

Data being read from different locations during one operation represents a causal consistent observation, i.e. the observed values do not contradict their causal dependencies. If an operation is triggered by a set of simultaneous (i.e. simultaneously observed) events, these events must be causally unrelated.

While this requirement is based on „consistent snapshots“ as described in [6] and [7], the additional constraint regarding triggering events actually makes it stronger.

It should be pointed out that temporal and causal consistency are *implicit* assumptions about *all* abstract operations. Of course, they may require additional effort at the implementation level (see section 2.6.).

2.5. Model Mapping and Implementation Dependency

The example shown in Figure 1 and Figure 2 models an idealized, high-level system view where the implementa-

tion of the vector data type (i.e. the realization of the vector storages and the add operation) is not visible. At a lower level, each of the vector storages s1 and s2 may be implemented by two storages for the vector components, {s1.x, s1.y} and {s2.x, s2.y}, respectively - see Figure 3. (For simplicity reasons, components a3, c2, c3 and s3 have been omitted.)

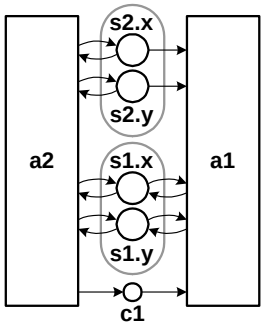

Figure 3. Compositional Structure (Low Leve)

On the other hand, the abstract add operation might be implemented by the following operation sequence:
(1) s1.x := s1.x + s2.x
(2) s1.y := s1.y + s2.y

In general, a lower-level model contains elements which are implementations of corresponding elements of the higher-level model, i.e. locations are implemented by locations, operations by operations, and agents by agents. (This dependency can be seen as a generalization of Hoare's „abstraction function" [23].)

In most practical cases, this *implementation dependency* (see [20] for details) is a static one-to-many mapping of higher-level elements to lower-level elements. In case of agents or locations, this is often called "containment". For example, each of the storages s1 and s2 from Figure 1 is implemented by two storages, namely {s1.x,s1.y} for s1 and {s2.x,s2.y} for s2 (see Figure 3). A one-to-many mapping is also given if an abstract agent is composed of several "inner" agents at the lower level.

However, the implementation dependency need not be a one-to-many mapping in all cases - it can be a many-to-one mapping as well. For example, several abstract agents of the same type can be realized by the same one agent at a lower level, following the principle of time division multiplex.

The implementation dependency can also be a dynamic mapping, e.g. if an abstract storage for binary trees is implemented by a dynamically changing set of lower-level storages.

As a direct consequence of mapping locations or operations, there is an implicit mapping of accesses. Every time a location or operation has to be implemented by multiple locations or operations, a high-level access must generally be „splitted". Figure 4 shows the low-level accesses for the example above: $R_{s2.x}$ and $R_{s2.y}$ correspond to R_{s2} (cp. Figure 2), $M_{s1.x}$ and $M_{s1.y}$ correspond to M_{s1}.

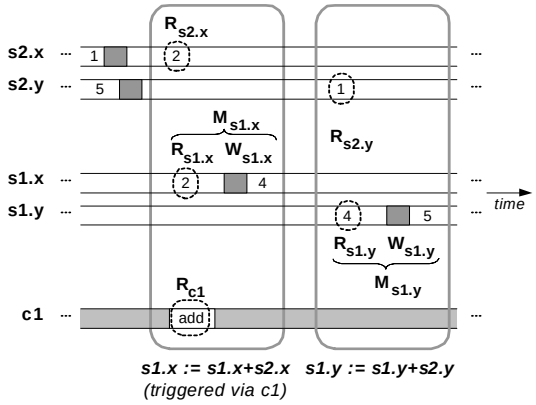

Figure 4. Accesses to Locations (Low Level)

2.6. Implicit Transactions and Snapshots

Locations, operations and implementation dependency actually establish the conceptual basis for answering the questions discussed in section 1.3..

As already mentioned, the mapping of locations and operations according to the implementation dependency implicitely defines a mapping of each high-level access to a set of lower-level (implementing) accesses. However, as a result from the high-level view and the assumption of temporal consistency (see section 2.4.), each access set must actually fulfill the atomicity, consistency and isolation requirements of the "ACID properties" [1]:

A set of accesses must be performed completely or not at all, because an incomplete set of accesses cannot be interpreted as a (successful) abstract access (→atomicity).

Intermediate values occurring between two lower-level accesses have no meaning at the high-level view (where a value change interval is given). Hence they must not be observable in the context of another access set. When all low-level accesses are completed, the resulting value(s) must represent (be consistent with) the abstract value after the high-level access (→consistency).

Conflicting abstract accesses may only occur in a defined temporal order (see section 2.4.). The related lower-level access sets must be serializable in a corresponding order (→isolation or serializability).[1]

1. Non-conflicting accesses may overlap. In this case, serialization is automatically possible.

The above three properties are requirements which must be met at the lower level in order to comply with the high-level model. The fourth "ACID property", namely durability, would only be mandatory if fail-safety would be given as an additional requirement.[1]

If we ignore durability, transactions can be derived solely on the basis of architectural models and their relationship as defined by the implementation dependency:

For each abstract operation affecting n abstract locations, there are n sets of implementing accesses which must (potentially) be managed as transactions.

In the example above (see section 2.5.), two potential transactions can be derived, namely access set $\{R_{s2.x},R_{s2.y}\}$ and $\{R_{s1.x},W_{s1.x},R_{s1.y},W_{s1.y}\}$.

If transactional techniques (e.g. locking) are actually neccessary depends also on the compositional structure. If an abstract storage is not subject to conflicting accesses (i.e. if only one agent performs sequential accesses) no special effort is needed.

The interpretation of transactions discussed above allows to distinguish between different types of transactions as well:

- If an abstract operation is implemented as an operation sequence an access sequence is given for each affected abstract location. Each of these access sequences represent a *flat transaction.*
- If a storage S is implemented as a set of storages $\{s_1,s_2,...s_n\}$ and an operation accessing S is realized as a set of concurrent operation sequences, each of them accessing a corresponding low-level storage s_i, the resulting accesses form a *distributed transaction.*
- A *nested transaction* can be derived if the mapping of accesses spans multiple levels, i.e. if lower-level operations or storages themselves have to be implemented by operations or storages of the next lower level, etc.

The need for snapshot algorithms [6][7] can also be interpreted as the outcome of a model mapping. If an abstract operation reads multiple storages but must be implemented by a set of operations (e.g. because the locations cannot be read within one operation), a *snapshot* must be implemented in order to guarantee the causal consistency required for the abstract operation's read accesses (see section 2.4.). Therefore, the implementing operations must be coordinated via a snapshot algorithm.

1. A detailed discussion of the ACID properties in this context can be found in [20]

3. Application and Further Research

An important motivation behind the modeling approach discussed here is to establish a better basis for designing transactional distributed systems. Transactions or lock requests no longer need to be introduced as additional concepts in later design stages. They can be derived from architectural models in a "natural" way, because abstract storages, abstract operations and implementation dependencies are integrated concepts of these models. For example, a shared abstract storage could be realized as an object, i.e. the abstract storage is implemented by the object's attribute storages. The types of operations which affect the abstract storage would be mapped to methods of that object. However, the high-level model requires the method calls to be serialized, e.g. by locking the object.

Furthermore, the "conceptual distribution" in terms of abstract storages can help to determine the adequate physical distribution of a system. Storages which implement the same abstract storage should be kept "local", i.e. on the same hardware node, because otherwise distributed (and expensive) locking mechanisms were needed in order to guarantee temporal consistent reads. In contrast, storages which realize different abstract storages are good candidates for being implemented on different nodes, because weaker consistency (i.e. causal consistency) is required in this case - which is easier to achieve.

As already discussed, architectural models according to our approach establish a basis for identifying transactions and snapshots. Therefore, it appears useful to integrate the relevant concepts into a corresponding programming language and platform. This idea is being pursued by us in an ongoing research project, STAGE [24]. A virtual machine and an accompanying intermediate language are under development. The intermediate language, as a basis for a high-level "architecture oriented" language, allows the definition of compositional structures, including abstract storages, abstract operations and implementation dependencies. Because this information will be available at runtime, the runtime environment can transparently detect and process transactions[2] and snapshots.

In this case, explicit transaction boundaries or lock requests are no longer needed in the program. Instead, the platform includes a lock management based on abstract locations and operations, which are defined in the intermediate language. Every time an abstract operation has to be performed, each affected abstract location is identified and, if neccessary, locked in an appropriate lock mode. However, locking is done only at the highest possible level. For

2. Orca [25] follows a similar idea but is built on a simpler conceptual basis which does not support the concept of locations.

example, if there is an implementation hierarchy of abstract storages (abstract storages being implemented by storages which, in turn, are implemented by storages, etc.), the most abstract storage is locked, which implicitely locks all subordinate storages as well.

The actual need for locking also depends on the (shared) usage of an abstract storage. Whether a storage is accessible by more than one agent or not can be decided on the compositional structure which, again, is available at runtime.

As discussed in section 2.4., causal consistency is required if multiple locations are read during one operation. This requirement can be met in different ways, depending on the physical distribution of the abstract locations. Setting a single lock for all locations to be read would be a simple but suitable solution if these locations are implemented at the same computer: The lock would guarantee temporal consistency, which also implies the required causal consistency. However, this solution appears inappropriate if the observed locations are implemented at different computers, because distributed locking could cause too much unneccessary idle time. In this case, a snapshot protocol can be used to collect copies of the locations' contents and temporarily store them in a snapshot cache at the machine where the operation will actually be processed. This approach is being implemented at the VM level of the STAGE platform [24], invisible for the programmer.

In the context of implicit transactions, controlling durability in an efficient way is one of the open questions. Demanding each implicit transaction to yield a durable state would require very frequent safepoints, which might be too expensive. As an alternative solution, only selected abstract operations could be marked as operations with durable results and/or selected abstract storages could be marked as durable.

Beside data consistency, there are further aspects of distributed systems which should also be addressed by architectural models, such as partial failure of hardware nodes or latency and unreliability of network connections. Our approach takes this into account, because the elementary building block for communication is the asynchronous send operation with potential message loss (see section 2.2., channels). This mechanism does not neccessarily require a certain agent as receiver, which allows for a variety of scenarios. A receiver can be (temporarily) unavailable or unobservant, multiple receivers can be connected to a channel for broadcast communication, and synchronous communication can be built on a pair of asynchronous messages (a request and a corresponding response).

4. Conclusion

The modeling approach presented in this paper integrates a certain aspect of distributed systems, namely data consistency. By doing so, architectural models not only become more expressive, they also form a basis for a more systematic design of distributed transactional systems and automatic runtime support for transactional techniques.

5. References

[1] Theo Härder, Andreas Reuter, *Principles of Transaction Oriented Database Recovery - A Taxonomy,* University of Kaiserslautern, 1982

[2] Jim Gray, Andreas Reuter, *Transaction Processing: Concepts and Techniques,* Morgan Kaufmann Publishers, 1993

[3] George Coulouris, Jean Dollimore, Tim Kindberg, *Distributed Systems - Concepts and Design,* 2nd Edition, Addison Wesley, 1994

[4] Carl Adam Petri, *Kommunikation mit Automaten,* PhD Thesis,Technische Hochschule Darmstadt 1962

[5] Sape Mullender (Ed.), *Distributed Systems,* Addison Wesley, 1993, pp. 55

[6] K. Mani Chandy, Leslie Lamport, *Distributed Snapshots: Determining Global States of Distributed Systems,* ACM Transactions on Computer Systems, Vol. 3, No. 1, February 1985, pp. 63-75

[7] Gerard Tel, *Introduction to Distributed Algorithms,* Cambridge University Press, 1994

[8] Mary Shaw et al. *Abstractions for Software Architecture and Tools to Support Them,* Carnegie Mellon University, Pittsburgh, 1995

[9] Paul C. Clements, *A Survey of Architecture Description Languages,* Proc. of the 8th Intl. Workshop on Software Specification and Design, 1996

[10] Nenad Medvidovic, Richard N. Taylor, *A Classification and Comparision Framework for Software Architecture Description Languages,* Dept. of Information and Computer Science, University of California, Irvine, 1997

[11] Martin Fowler, *UML Distilled,* Addison Wesley, 3rd ed., 2003

[12] Grady Booch, James Rumbaugh, Ivar Jacobson, *The Unified Modeling Language User Guide,* Addison Wesley, 1999

[13] Christine Hofmeister, Robert Nord, Dilip Soni, *Applied Software Architecture*, Addison Wesley, 1999

[14] Dilip Soni, Robert L. Nord, Christine Hofmeister, *Software Architecture in Industrial Applications*, Proceedings of the Intl. Conference of Software Engineering, 1995

[15] Frank Keller, Peter Tabeling et. al. *Improving Knowledge Transfer at the Architectural Level: Concepts and Notations*, International Conference on Software Engineering Research and Practice, Las Vegas, June 2002

[16] Frank Keller, Siegfried Wendt, *FMC: An Approach Towards Architecture-Centric System Development*, IEEE Symposium and Workshop on Engineering of Computer Based Systems, 2003

[17] Siegfried Wendt et al. *The Fundamental Modeling Concepts Home Page*, fmc.hpi.uni-potsdam.de

[18] Clemens Szyperski, *Component Software*, Addison Wesley, 2nd Edition, 2002

[19] Andrei V. Borshchev, Yuri G. Karpov, Victor V. Roudakov, *Systems Modeling, Simulation and Analysis Using COVERS Active Objects*, IEEE Workshop on Engineering of Computer Based Systems, 1997

[20] Peter Tabeling, *Der Modellhierarchieansatz zur Beschreibung nebenläufiger, verteilter und transaktionsverarbeitender Systeme*, Shaker Verlag, Aachen 2000 (PhD Thesis, University of Kaiserslautern)

[21] Peter Tabeling, *Ein Metamodell zur architekturorientierten Beschreibung komplexer Systeme*, Lecture Notes in Informatics (LNI) - Proceedings of “Modellierung 2002", Workshop of the Gesellschaft für Informatik, Tutzing, 2002

[22] Peter Tabeling, *Multi-level Modeling of Concurrent and Distributed Systems*, International Conference on Software Engineering Research and Practice, Las Vegas, June 2002

[23] C. A. R. Hoare, *Proof of Correctness of Data Representations*, Acta Informatica, vol. 1 no. 4, 1972, pp. 271-281

[24] Peter Tabeling et. al. *The STAGE Project Home Page*, stage.hpi.uni-potsdam.de, March 2004

[25] Henri E. Bal et al. *Orca: A Language for Parallel Programming of Distributed Systems*, IEEE Transactions on Software Engineering, Vol. 18, No. 3, 1992