Integrative Architecture Elicitation for Large Computer Based Systems

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

Abstract

Large and complex computer based systems are the result of an evolution process which may take many years. Heterogeneity is an important characteristic of such systems: During the evolution process, different technologies were used for development; different third-party products were integrated into the system; information about the system components is often incomplete, scattered over a company and comes in various formats. To support the development of large systems, this information has to be collected and integrated into architectural models.

Hence, architecture elicitation of a large computer based system has to cope with the heterogeneity of technology and information sources. Some information can be extracted and prepared by tools, but the majority is usually inside documents and the heads of the developers. Therefore, in addition to tool support, this task requires a lot of abstraction which can only be done by humans.

This paper presents concepts and abstractions for integrative modeling, based on the Fundamental Modeling Concepts. It provides the necessary abstractions to combine the information about the different system components, including third-party products and hardware, into one integrated architecture model. The approach reflects experiences from many architecture elicitation projects. It covers modeling patterns and typical abstractions, supported by guidelines for the elicitation process.

The terminology and notation focuses on the communication about technical systems between persons being involved in system development. As a benefit, integrative models foster the efficient evolution of large systems, the introduction of new developers, or the handing over of a project.

Keywords: Modeling, Architecture, Heterogeneous Systems, Fundamental Modeling Concepts, FMC

1. Introduction

1.1. Large and Heterogeneous Systems

Today, the software part of commercial computer based systems is typically very large, i.e. a program size of several million lines of code is not unusual.

Largeness implies evolution. A complete redesign of a system of that size would introduce too many errors with the new code and reimplementing the existing functionality would be too expensive and time-consuming. Therefore, these systems are often maintained in an evolutionary way, i.e. by keeping most of the software and changing or adding only a few percent of code [1].

Evolution implies heterogeneity. As an unavoidable consequence of the evolutionary maintenance, large systems contain significant amounts of 'legacy' code. For example, SAP's Netweaver$^{\text{TM}}$ platform comes with several 10 Million lines of code being written in the 90s as part of the R/3$^{\text{TM}}$ system. Because technology and programming 'paradigms' change significantly over time, such systems inevitably incorporate a wide variety of languages, libraries, frameworks, protocols, etc. being designed by different teams and third parties. Heterogeneity is even greater in case of large *embedded* systems, where a mixture of hardware and software components ist given.

1.2. Architecture Elicitation

Systems of the size and complexity as described above can only be developed and maintained on the basis of a well understood, adequately described and maintained *overall* architecture.

Roughly following the architectural views from [2][3], the following types of architectural structures are distinguished:

- *System structures* in terms of conceptual or physical components of the running system which interact to offer the required functionality. These can be *high-level, conceptual models* close to the application domain (conceptual view) or *low-level, platform-related models* (execution view and hardware architecture).
- *Software structures* in terms of *source code modularization*, e.g. as classes, interfaces, etc. (module view), or software structures in terms of *deployable code units*, e.g. libraries, executables, etc. (code view).

There are several scenarios where (re-) establishing and modeling such architectural structures from code, documents or other sources of information is needed:

- Documentation

  If the system reaches a certain complexity, at the latest, its overall architecture should be consolidated and described.
- Evolution

  If a system is updated evolutionary, software structures are changed while system structures should mostly remain stable. In order to avoid architectural erosion, architectural models must exist and be updated regularily to reflect changes.
- Porting

  When porting a system, its platform or significant parts thereof are exchanged, i.e. not only software structures are affected, but also platform-related models. However, high-level system structures should remain valid. Therefore, these structures must be identified and described in a first step.
- Reengineering

  In contrast to porting, even high-level structures are subject to change. In order to identify and discuss the neccessary changes, architectural models are needed.

1.3. Resulting Problems

All these scenarios have in common that a „big picture“ has to be extracted from a variety of information sources, such as existing programs, hardware structures, documentation etc. Due to the size and heterogeneity of the systems being discussed here, the integration of the many different concepts requires a very flexible approach. Abstraction and presentation techniques are needed which allow, for example, to combine objects, assembler code and hardware components within a single model. These models are mandatory means of communication, used by architects, chief developers and project management [4][5].

Heterogeneity of system components. As pointed out above, real-life systems use a variety of techniques in terms of hardware or programming languages. The architecture of a system therefore comprises not only the software written by the own developers, but also other components bought and used as 'black box', like server hardware or operating system.

Heterogeneity of information sources. The heterogeneity of system components implies that information about them can not be found in one place. Typical information sources are documentation, source code (if available) or developer's knowledge.

Tools can support the extraction of information from some sources, typically from source code. Unfortunately these tools can hardly analyze a mixture of different programming languages and 'paradigms'. Even a tool being capable of doing so can never retrieve the architectural structures being buried in third-party binary components, hardware, documents and - last not least - the heads of persons being involved in a project.

Integrative models. The heterogeneity of system components and information sources results in a big number of different information pieces which have to be integrated to an architectural model. This integrative model should provide abstractions which can be mapped to all used techniques, at least to hardware and software as well.

At first sight, UML [6][7], with its profiling and extension mechanisms, seems to be a possible solution. However, adding all the neccessary profiles to the already hard-to-handle complexity of UML's thirteen diagram types makes UML a bad candidate for the intended usage of the models. A recent study among software companies [4] shows that UML is primarily used in the context of coding and low-level software design but does not play a significant role in the modeling of an overall, high-level architecture.

Sharing knowledge. The need for division of labor does not only apply to the development, but also to the architecture elicitation of large computer based systems. Different specialists may extract architecture information from the various information sources. They then need to share their knowledge in order to create an integrative model. It is obvious that the integrative model implies a common language for the specialists and should therefore be optimized for the human communication of technical issues.

Experience from several modeling projects showed that revealing architectural structures from large, heterogeneous systems requires significant human effort and strong abstractions. This must be supported by:

- a conceptual basis which is flexible enough to enable the integrative description of a heterogeneous system
- modeling and abstraction techniques which help to further reduce the system's complexity

1.4. Outline

The following sections present a modeling approach and a set of approved abstractions which have been developed over several years and reflect experiences drawn from a variety of modeling projects. These projects addressed both academic and industrial systems, such as the Apache HTTP server [8][9][10] and SAP's R/3^{TM} system. The methodology has been successfully applied at Siemens, SAP, Alcatel, BMW and others.

Section 2 gives a short introduction to the terminology and notation and discusses its focus. Typical abstractions needed for architecture elicitation will be discussed in section 3. The additional guidelines in section 4 deal with the practical application during architecture elicitation. Section 5 gives some examples of architecture elicitation projects which have also influenced the terminology and notation.

2. Integrative Modeling with FMC

2.1. Basic Concepts

The modeling approach is based on the Fundamental Modeling Concepts (FMC), an approach for describing architectural structures of computer based systems, using a semiformal graphical notation [11] [12].

In order to support a wide variety of systems, FMC distinguishes three basic types of system structures which are fundamental aspects of any computer based system:

- *Compositional structure*, i.e. the static structure consisting of the interacting components of the system.
- *Dynamic structure*, i.e. the behavior of the components.
- *Value structure*, i.e. the data structures found in the system.

The corresponding conceptual and notational elements will be discussed below. However, the presentation will focus on elements of the compositional structure since these are the most important in the context of this paper. More information about FMC can be found in [11] and [12].

Compositional structure. Any system can be seen as a composition of collaborating components called *agents*. Each agent serves a well-defined purpose and communicates via *channels* (or shared storages, see below) with other agents. If an agent needs to keep information over time, he has access to at least one *storage* where information can be stored. Channels and storages are (virtual) *locations* where information can be observed.

The agents are drawn as rectangular nodes, whereas locations are symbolized as rounded nodes[1]. In particular, channels are depicted as small circles and storages are illustrated as larger circles or rounded nodes (see Figure 1). The

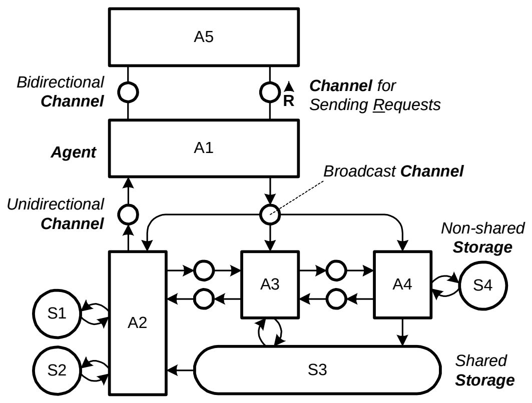

Figure 1. FMC Compositional Structure Diagram - Basic Elements

possibility to read information from or write information to a location is indicated by arrows. Types of agents and locations are identified by descriptive textual labels.

Arbitrary complex structures can be described because agents can be connected to multiple locations and locations can be shared by multiple agents. For example, it is possible to describe *unidirectional, bidirectional channels* (connecting only two agents) as well as *broadcast channels* (connecting more than two agents — see Figure 1) and *channels for sending requests* (bidirectional, with an "R"-arrow indicating the request direction). *Shared storages* can be used for buffered communication.

In general, agents and locations are *not* necessarily related to the system's physical structure. The compositional structure facilitates the understanding of a system, because one can *imagine* it as a physical structure (e.g. as a team of cooperating persons). Nevertheless, on lower levels of abstraction a direct mapping to physical parts of the system might be possible. As sections 3.1. to 3.6. will show, agents and locations can represent both hardware- and software-related elements.

Dynamic structure. One of the fundamental concepts for describing system behavior is the *event*, a value change occurring at a certain location and at a certain point in time. An *operation* is the "smallest" activity an agent can perform — an agent reads values from several locations, pro-

1. This notation originates from an industry standard [13]

cesses these values and writes the result to a certain location. This covers state changes (writing storages) and communication (writing or reading channels). Operations can be triggered by events and in turn produce events. This leads to causal dependencies of events.

FMC diagrams for dynamic structures are based on Petri nets [14], thus providing a sound basis for describing both sequential and concurrent systems. Labeled transitions symbolize event types, operation types or even complex activities, whereas places (mostly) symbolize control states. Additional places and edges are introduced to describe causal dependencies resulting from communication, see Figure 2. Branch conditions are expressed as predicates at edges (see Figure 2, below "receive request"). Another extension allows the description of *recursion*, see [15].

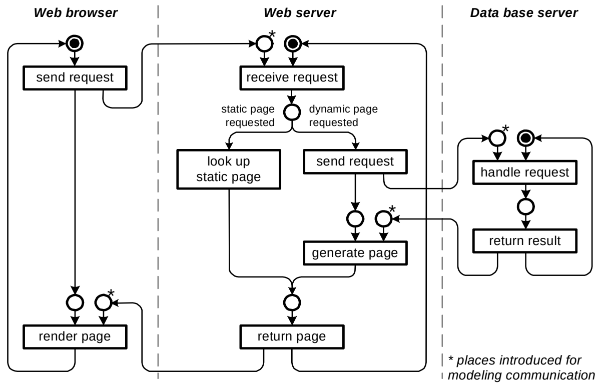

Figure 2. FMC Dynamic Structure Diagram

Value structure. Each location of the compositional structure holds a unit of information, called a value. A value can be a simple, *unstructured value* such as a bit or integer as well as a *structured value* like a tree or the whole content of a database. FMC offers a dedicated diagram type for value structures, an extension of entity/relationship diagrams [16]. More information can be found in [11].

2.2. Scope and Focus

The strict separation of system structures and software structures as discussed in section 1.2. is an important element of FMC, because FMC is focused on system structures. Hence, the modeling of a software system means more than describing structures of the self-written program code. Instead, it covers *runtime structures* of the whole system, at the intended level of abstraction. This can include runtime object structures, operating system services, a virtual machine, and other third-party components. The basic idea is that these elements are needed to provide the complete functionality of the system — hence they have to be reflected in a model of the overall system architecture.

In case of third-party components, the model should describe as much functionality or control flow as needed to understand the self-contributed parts of the system. For example, using a framework with a pre-built event dispatcher partially dictates the compositional structure and control flow of an application. In order to illustrate the overall behavior, both the pre-built event dispatcher and the self-defined event handlers have to be combined in one model (see also Figure 4).

3. Typical Abstractions

3.1. Modeling of Software Artifacts

Even though object-oriented methods seem to be the predominant development technique, there is often a need to adopt a non-object-oriented view on software.

Third-party components. These are often given as binary code with accompanying *documentation* being the *only* source of information.

- *Libraries*

  They often describe general or domain specific data types, including the related operation types and implementation. Since the code primarily provides types missing in a given programming language, it is usually *not* appropriate to introduce a 'library agent' into the system model. Instead, the library can be reflected in the model by instances of storages of the library data types actually being used in the system. Operations being defined in the library show up as abstract operations in the behavior model of (application-related) agents accessing these storages. This corresponds to the *data type view* as described in section 3.2.
- *Frameworks*

  A Framework imposes parts of the overall compositional structure and behavior. Thus it must be reflected in both types of models. This can be done, for example, by describing a GUI framework's event dispatcher as an agent, controlling the (self-written) event handlers by sending requests to them via channels Figure 4. For the dynamic structure, the event loop yields the template of the overall control flow, with event handler activities filling the gaps.
- *Subsystems*

  Stand-alone products like a database system or platform services like a persistency service or a request broker offer basic functionality to other parts of the system. Each functionality is typically represented as a single agent being capable of receiving and *processing* certain requests (e.g. a persistency service) or *transporting* requests (request brokers, event services).

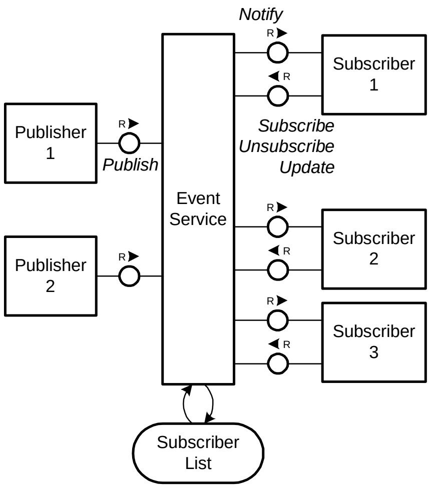

Figure 3. An Event Service as Part of a System Model

Generally, agents, storages and channels can each be derived by answering one of the following questions, based on the component's documentation:

- What data processing functionalities or services are provided by the component?
- Which implicit or explicitely named information must be 'kept inside' or 'remembered' by the component? Which agents need to access that information and how (read, write, modify)?
- Which implicit or explicitely named requests, messages or event notifications are exchanged between which agents and in which direction?

Platform-related elements. In case of platform-related models (the 'execution view' [2][3]), the compositional structure can also be derived from the following elements:

- *Processes and threads*

  These can be modeled as agents, either one agent per process (or thread) or multiple processes (or threads) can be mapped to one agent. The latter can be done if the processes co-operate in order to realize a certain functionality and the model should hide this co-operation.

- *Memory areas and files*

  These should, by nature, be represented as storages. If files or memory areas are accessed by several processes (or threads), this is reflected by a shared storage with corresponding access arrows.

- *Semaphores, locks*

  Showing processes (threads), files and memory areas can help to illustrate access conflicts and synchronization between processes or threads when sharing files or memory. In this case, the model can be enhanced by including locking mechanisms. This can be done by adding a lock (semaphore) manager as agent, which receives and processes lock-related requests sent by processes or threads.

- *Network connections*

  The simplest way to cover network connections is to introduce a channel for each connection. However, in some cases it might be appropriate to describe the network service of the platform as a dedicated agent, with the processes being connected to it. This can be useful if more aspects than message transport have to be shown.

An agent or storage can further be refined based on the different functionalities a process performs or the different data structures being stored in a file or shared memory.

Non-object-oriented code. Many computer-based systems rely on non-OO software for various reasons. In this case, modules have to be classified for modeling instead of classes. Modules which serve controlling purposes, e.g. containing a dispatcher loop, should be depicted as agents. However, modules containing data structure definitions which only implement abstract data types should be illustrated as storages — typically one storage per data structure instance. This again corresponds to the *data type view*, see section 3.2.

In case of a large system, mapping each module to a separate agent or storage would produce a model of too fine granularity. This can be reduced by combining closely related modules to a single agent or storage. For example, all modules dealing with persistency can be mapped to a single agent named “persistency service”.

3.2. Mapping of Objects and Classes

There are various possibilities to map objects to compositional structures [17][18]. In simple cases, an object is viewed as an agent or storage.

Object agent view. Considering an object to be an agent is a common interpretation in object-oriented design. Calling a method is interpreted as sending a message to the object which carries out the desired operation. Attribute data is mapped to an internal storage of the object agent, methods describe the object agent's behavior.

A typical example for this view is an event dispatcher object, see Figure 4. GUI toolkits usually use a singleton object to dispatch GUI events (e.g. mouse moved or button pressed) to other objects which have been registered as handlers for certain event types.

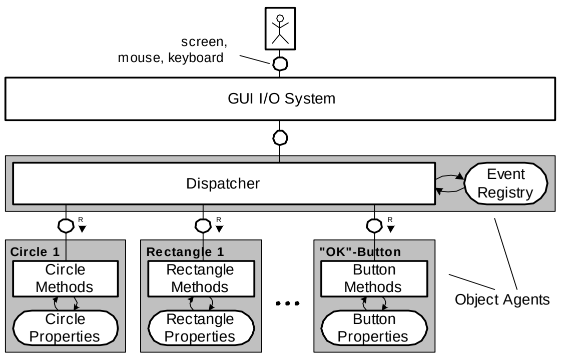

Figure 4. Object Agent View

Abstract data type view. This is another common interpretation where an object is seen as a storage for an abstract data type being described by the corresponding class — Bertrand Meyer defines a class as „an abstract data type equipped with a possibly partial implementation“ [19]. The class not only lists the operation types (i.e. the method signatures), it may also provide the implementation of the data storage (i.e. the attribute storages) and the implementation of the operation types (i.e. the method bodies).

Figure 5 shows a typical example, namely a ”Vector“ class defining an abstract data type ”Vector“ with corresponding operation types (scale, add, rotate etc.). Following the data type view, an instance of the ”Vector“ class can be seen as a storage which holds an abstract ”Vector“ value.

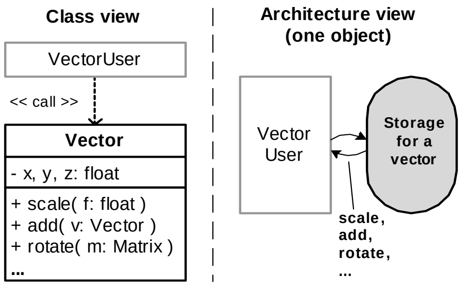

Figure 5. Abstract Data Type View

Platform-related view on objects. From the platform-related *execution view* [2][3] perspective, an object is just a data record in memory. This *data record view* explicitly shows the inner structure of objects, i.e. the individual storages for attribute data. In case of the ”Vector“ example (see above), the storages for the vector components now become visible, see Figure 6.

In contrast to the abstract data type view (see above), the discussion of inconsistency problems in the context of multithreading is now possible. The architecture model makes clear that object data is shared between threads, see Figure 6. In order to avoid inconsistency of attribute data, each thread's operations on object data must be synchronized with the others, e.g. by using a central lock manager. This view also provides an elegant way to describe object persistency. A persistency manager can access object data directly, for example, to increase performance, see Figure 6 below.

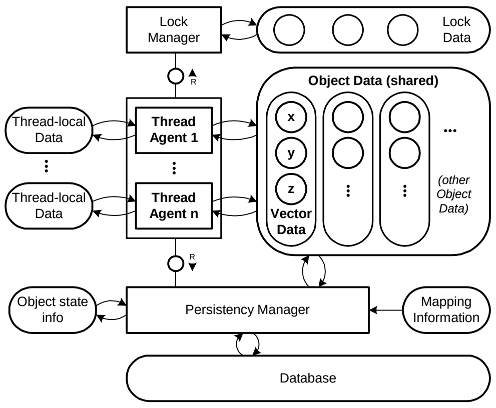

Figure 6. Objects as Data Records

3.3. Modeling of Hardware Elements

A mapping of hardware elements to elements of a compositional structure is straightforward. Processing elements like an arithmetic-logical unit or combinatorial circuits can be shown as agents. Memory units and registers can be simplified as storages — in case of dual ported RAM, as a shared storage. Wires can be mapped to channels, with busses being a potential exception. These may somtimes be shown as a communication service (agent), in analogy to communication services in software systems.

The two views at processors. Processors (stand-alone or as part of a micro-controller) can be described as an agent with access to program and data storage (RAM or registers) being capable of processing the loaded program code. However, processors are used to implement a component being defined by its program. Therefore, it is often appropriate to replace the processor by an agent representing the component to be realized.

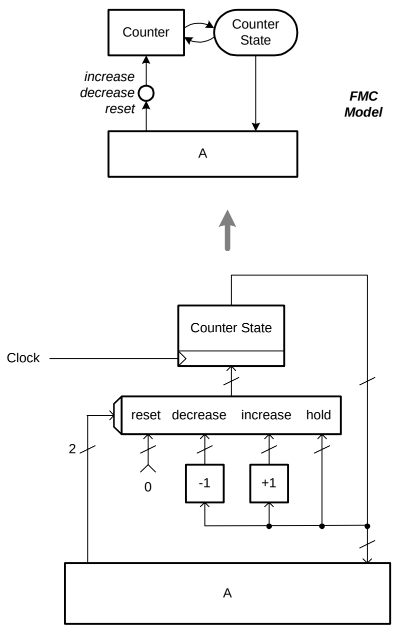

Figure 7. FMC-based Model of Harware Elements

3.4. Hiding Software-related Platform Structures

The mappings discussed above allow the transformation of various technical solutions into FMC-based model elements. However, they often result in a model of too high granularity. An obvious way to reduce this complexity is to integrate partial compositional structures of agents or storages into higher-level agents or storages.

However, experience shows that this purely composition-based abstraction alone is not sufficient for establishing high-level architecturals models. Additional techniques are needed which help to hide typical, implementation-related aspects of large systems.

Subsystems. Subsystems like a persistency service, an event service or a request broker provide basic functionality in a generic way. For example, a request broker essentially allows communication. Because this generic solution is not bound to a certain application domain, it should not be part of a high-level application-related model.

For example, a request broker (or event service) can be eliminated from the model and be replaced by the abstract point-to-point (or broadcast) connections it realizes, see Figure 8 (left).

Persistency services and database systems are generic means for providing persistent storage. They can be omitted from a model, where the database or persistency clients have direct access to abstract, persistent storages, see Fig-

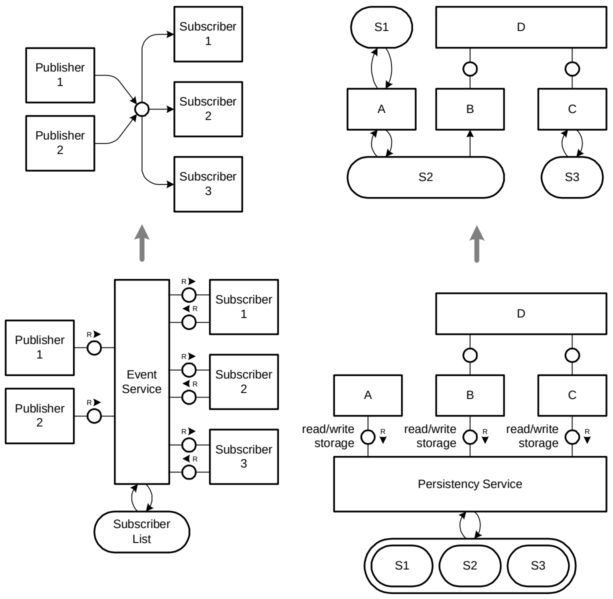

Figure 8. Hiding an Event Service / Persistency Service

ure 8 (right). In this view, the individual accesses can be shown (e.g. only A accesses S1, B does not write S2, etc.).

Synchronization mechanisms. Sychronization is needed when accesses to shared data structures are in conflict and a transactional set of accesses is used to implement a (virtually) atomic, single access. If a model describes the system at a higher level, with abstract shared storages and abstract atomic operations instead of the implementing data structures and non-atomic activities, there is no need for synchronization in this idealized view, see also [20][21]. Consequently, synchronization mechanisms like locks and semaphores should be concealed completely at this level, see Figure 9.

Caching and replication. Caching and replication replace storages and agents with multiple storages or agents in order to decrease access latency or increase safety, respectively. Again, this should not be shown in a higher-level model, i.e. the corresponding structures can be reduced to a single, abstract storage or agent.

Multiplexing and pooling. In order to reduce resource consumption, a single worker (for example, an operating system process) or even a pool of workers are used to realize a high and changing number of virtual service agents. Therefore, at a higher level model, these virtual service agents can replace the worker (or worker pool), including supporting mechanisms like context management or pool management.

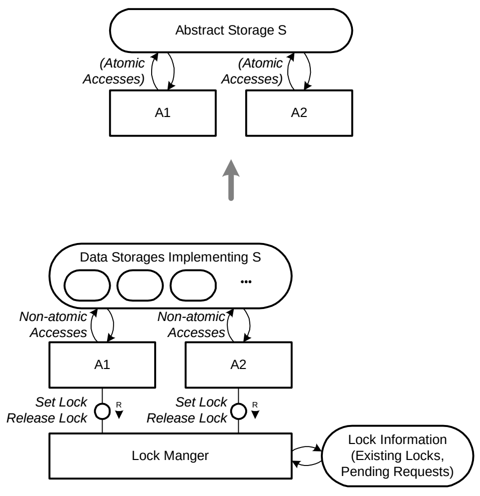

Figure 9. Hiding Synchronization

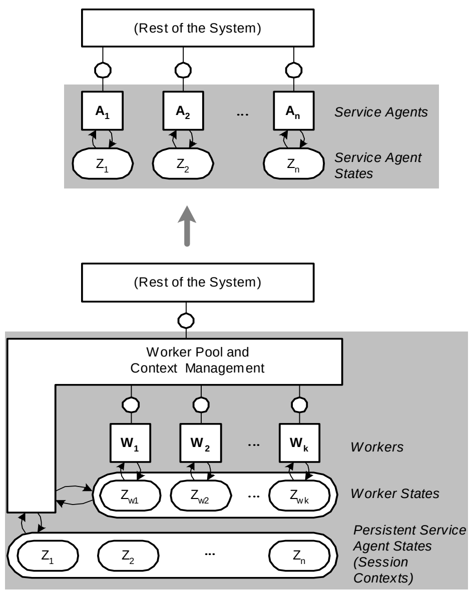

Figure 10. Hiding Pooling

3.5. Hiding Hardware-related Platform Structures

Similar to software-based system models, hardware structures can be simplified by combining closely-related hardware parts (i.e. part of a compositional structure) to a single abstract agent. Beside this widespread abstraction, there are implementation-related structures and elements which can be removed from a high-level model.

Physical distribution. The physical distribution of a system often depends on the individual installation or a predefined hardware infrastructure, i.e. it is often a deployment and installation aspect. This should be separated from higher level models, i.e. these models should not reflect the physical, but the *conceptual distribution* (compositional structure) of the system.

Physical packaging. The integration and placing of hardware components is often determined by manufacturing aspects. For example, components with quite different functions are integrated into a single unit to decrease size and costs. In a more abstract model, the compositional structure should not reflect the physical parts but the different functionalities.

Subsystems. Similar to software systems, hardware systems contain elements which provide basic, application-independent functionality and can therefore be omitted in higher-level models.

Simple examples are clock generators or driver units. Both do not serve application-related purposes and can therefore be ignored in abstract models.

In analogy to a request broker in a software system, a bus, including the bus arbiter, can be eliminated from a model. In this case, the hardware components being connected indirectly via the bus are shown as agents with direct connections, see Figure 11.

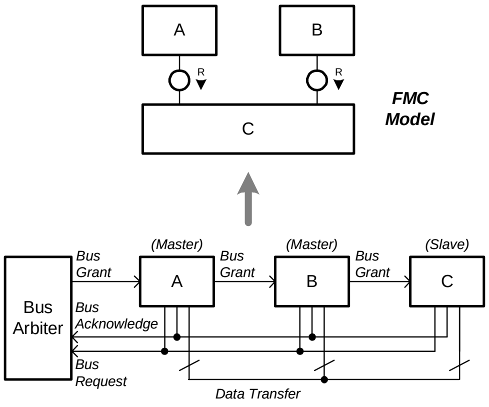

Figure 11. Hiding a Bus

3.6. Hiding Object Structures

The object mappings discussed in section Figure 3.2 actually do not reduce complexity. Reflecting each object as a dedicated element in an architectural model yields models of extreme granularity — large systems with thousands or millions of objects cannot be modeled this way. Hence, mappings are needed where a collection of objects can be mapped to a single, high–level architectural element [17][18].

High–level abstract data type view. This view extends the idea of the abstract data type view (see section 3.2.). It is applicable when a set of objects is used to implement an abstract data type. For example, a data type ”tree“, representing expressions to be evaluated, could be descibed by several classes as shown in Figure 12. The basic idea behind the classes is to provide the possibility to store trees. Hence, at a higher level, an instance of a corresponding object set can be viewed as a single storage for an abstract data type ”Tree“ — see top of Figure 12.

High–level object agent view. In many cases, object sets can be described as one agent. For example, a persistency service (agent) may consists of a singleton object of a generic persistency manager class and a set of singleton objects, each of them managing persistent objects of a certain class — see Figure 13. The persistency service creates persistent objects, restores or saves their state from/to the database, etc. In a high-level model, the inner structure of the persistency service *would be omitted.*

This architecture has been chosen for an project called ’Object Services’ at SAP in 1999 whose goal was to add a new persistency and transaction service for ABAP Objects [22].

The two abstractions presented above have in common that many objects are mapped to a single element of an architectural model. While this helps to reduce complexity, there are still other scenarios where a hierarchical one–to–many mapping is not sufficient. In this case, further abstraction techniques are neccessary, see [17][18].

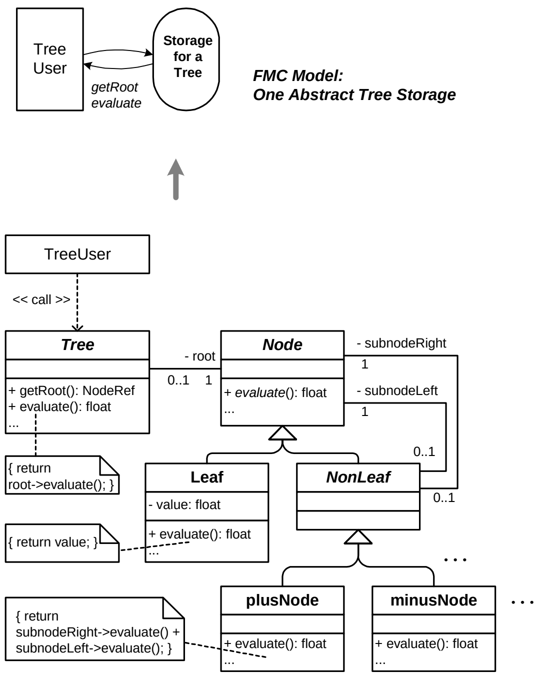

Figure 12. High-level Abstract Data Type Viev

3.7. Further Abstractions

Beside the abstractions presented here, there are further techniques to build integrative models and reduce their complexity. One method is to limit a model’s scope to a certain *scenario*, for example, a use case. Another way of abstraction is *aspect-oriented modeling*, where a model

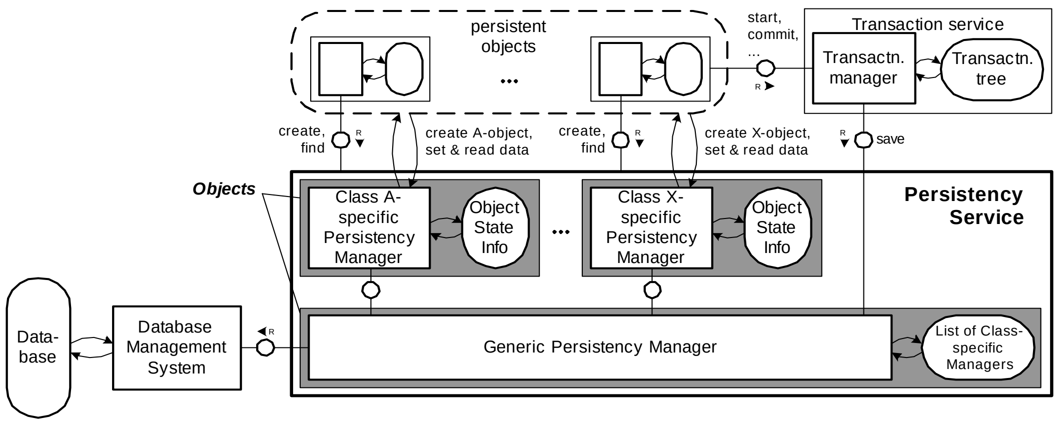
persistent objects
start, commit, ...
Transaction service
Transactn. manager
Transactn. tree
create, find
create A-object, set & read data
create, find
create X-object, set & read data
save
Objects
Class A-specific Persistency Manager
Object State Info
Class X-specific Persistency Manager
Object State Info
Persistency Service
Data-base
Database Management System
Generic Persistency Manager
List of Class-specific Managers

Figure 13. High-level Object Agent Viev

only covers a certain aspect like the start-up behavior, error handling or persistency.

4. Additional Guidelines for Practical Application

The concepts discussed above represent a set of well-proven modeling techniques. This section provides additional hints for architecture elicitation, based on experience from modeling large industrial systems (SAP R/3^{TM}, the Apache HTTP server and others).

Information sources. A very important aspect is the type and relevance of the different sources of information:

1. *Interviews*

   Architectural models cover what certain stakeholders in a project view as important: Fundamental software and hardware elements. As Fowler states in [23], these components are significant "because the experts developers say so.“ Therefore, it is recommended to ask chief developers, architects and project managers about their 'big picture'. By doing so, one can often gain information which is unreachable otherwise and gives valuable input where to search for additional information. The latter is often a mandatory basis for the next steps because studying all documents and program files is simply not feasible in case of large systems.

2. *Documentation*

   As a complement to interviews, existing documentation provides important details, even if it is incomplete or outdated. As stated in [24], "software engineers typically do not update documentation [on a regular basis but] out-of-date software documentation remains useful in many circumstances." By nature, statements about architectural structures are most probably still valid while most changes affect low-level details.

3. *Running system, program code, design specifications*

   Remaining uncertainties can be solved by analyzing the running system, selected program sections and specifications. Here, tools support searching and navigating.

Iterative modeling. The priority list above should not be interpreted as a three-phase process model. Instead, after creating or updating a model, it should *again* be presented to the stakeholders in additional interviews, in order to reveal misunderstandings and missing elements. Updated documentation should first be released to 'beta readers', i.e. other persons with *little* knowledge about the described system. This helps optimizing models and related documents for the customer and newly hired personal joining a project team. If neccessary, this process can be *repeated* several times.

Models reflect consensus. A frequent observation is that different people have different views of a system. In order to create a meaningful model of the overall architecture, the different views must be integrated. This can even produce additional, important input to a project, because the stakeholders must agree to a common view of the system. Secondary, complimentary models can cover information which is only relevant to certain persons or groups.

5. Experiences

Development of FMC was started in 1974 during an architecture elicitation project of Siegfried Wendt for the Siemens AG and has been refined in the following years in many other projects with industry partners [12].

One of the biggest architecture elicitation projects was the modeling of the SAP R/3^{TM} basis system between 1990 and 1996. As it turned out, the most important information source were the leading developers. The elicitation process typically started with an interview to get a rough idea of the concepts, followed by examination of the running system (if possible) and inspection of the source code. Because the R/3 system relies on several, partially proprietary programming languages (C, RSYN, CCB, ABAP, DYNP) and various platform technologies, heterogeneity was a major challenge. Many iterations led to more details on the one hand and to better abstractions on the other hand. The result was a series of technical reports, the 'SAP Blue Books'.[1] They are not only used inside SAP. For example, IBM used some of these reports when porting the R/3^{TM} basis to the AS/400^{TM} platform.

During another architecture elicitation project, a widely used open source system has been analyzed: Between 2001 and 2003, research assistants and students of the Hasso-Plattner-Institute examined and modeled the Apache HTTP server and published the results [8][9][10].

6. Concluding Remarks

The notational concepts and abstraction techniques presented above have been developed during a series of projects in academia and industry. Due to the independency from individual platforms, programming languages and paradigms, FMC has proven to be a very good conceptual basis for describing architectural structures of large and heterogeneous computer based systems.

1. Unfortunately, these reports can't be published because they contain intellectual property which may not be disclosed.

In addition to this, the abstraction techniques foster model integration and consolidation, a prerequisite for building high-level architectural system models.

The techniques complement analysis tools and can be utilized to varying degrees, depending on the intended level of abstraction being suitable for the modeling context: Porting, reengineering, evolutionary development, etc.

Further research is neccessary to integrate the methodology with model based aproaches.

7. References

[1] Gio Wiederhold, *The Product Flow Model, CS Dept.* Invited Talk, Hasso-Plattner-Institute, Potsdam, Germany, October 2004

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

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

[4] Frank Keller, *Über die Rolle von Architekturbeschreibungen im Software-Entwicklungsprozess,* PhD Thesis, University of Potsdam, Germany, August 2003

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

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

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

[8] Bernhard Gröne et. al. *Architecture Recovery of Apache 1.3 - A Case Study,* Proceedings of The 2002 International Conference on Software Engineering Research and Practice, Las Vegas, 2002

[9] Bernhard Gröne et. al. *The Apache Modeling Project,* HPI Technical Reports of the Hasso-Plattner-Institute, Vol. 5, ISBN 9-937786-14-7, Germany, July 2004

[10] Bernhard Gröne et. al. *The Apache Modeling Project Homepage,* http://fmc.hpi.uni-potsdam.de, as fetched in November 2004

[11] 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

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

[13] Deutsches Institut für Normung e.V., *Betrieb von Rechensystemen — Begriffe, Auftragsbeziehungen,* DIN 66200, Beuth, Berlin, 1968.

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

[15] Siegfried Wendt, *Modified Petri Nets as Flowcharts for Recursive Programs,* Software — Practice and Experience, vol.10, 1980, pp. 935-942.

[16] Peter Chen, *"The Entity-Relationship Model — Towards a Unified View of Data",* ACM Transaction on Database Systems, vol. 1, no. 1, 1976, pp. 9-36.

[17] Wolfram Kleis, *Konzepte zur verständlichen Beschreibung objektorientierter Frameworks,* PhD Thesis, University of Kaiserslautern, Germany, 1999

[18] Peter Tabeling, Bernhard Gröne, *Mappings Between Object-oriented Technology and Architecture-based Models,* Proceedings of the 2003 International Conference on Software Engineering Research and Practice, Las Vegas, 2003

[19] Bertrand Meyer, *Object-Oriented Software Construction,* 2nd Ed., Prentice Hall, 1997

[20] Peter Tabeling, *Architectural Description with Integrated Data Consistency Models,* IEEE Symposium and Workshop on Engineering of Computer Based Systems, 2004

[21] Peter Tabeling, *Der Modellhierarchieansatz zur Beschreibung nebenläufiger, verteilter und transaktionsverarbeitender Systeme,* PhD Thesis, University of Kaiserslautern, Germany, 2000

[22] Bernhard Gröne et al. *Object Services für R/3 Release 99 - Konzepte,* SAP AG Walldorf, Germany, 1999

[23] Martin Fowler, *Who Needs an Architect?* IEEE Software, vol. 20 no. 5 pp. 11-13, 2003

[24] Timothy C. Lethbridge et. al. *How Software Engineers Use Documentation*, IEEE Software, vol. 20 no. 6 pp. 35-39, 2003