

# A Generic Model for the Use and Specification of Software Architectures

Wilhelm Rossak, Vassilka Kirova, Leon Jololian  
New Jersey Institute of Technology  
Department of Computer and Information Science  
University Heights  
Newark, NJ 07102, USA  
rossak@pluto.njit.edu

Harold Lawson  
Lawson Förlag och Konsult AB  
S-181 33 Lidingö, SWEDEN

Tamar Zemel  
RAFAEL  
Missile Division  
P.O. Box 2250  
Haifa, ISRAEL

## Abstract

This paper presents an architecture-based approach to the development of domain specific large and complex software systems. The wider context of our approach is the Engineering of Computer Based Systems (ECBS), as specified as a research and application domain by the IEEE TC on ECBS. We start with a short discussion of the ECBS framework, identifying the devel-

opment process and architecture specification as the two major critical elements. Based on these prerequisites, we then present our current work in this area to develop a Generic Systems Integration Framework (GenSIF). We discuss our model-based approach to the specification of software architectures and the use of architecture models in an integrated and enhanced development process. The proposed first draft of a generic architecture reference model is the guiding principle and major contribution

in our presentation.

**Keywords:**

software architecture, architecture reference model, architectural elements, software development

## 1 Introduction

The development of complex computer-based systems is a complicated task that can be made simpler by means of new engineering strategies. To support this task, a comprehensive model of the ECBS (Engineering of Computer Based Systems) discipline has been proposed by the IEEE ECBS TC <sup>1</sup>. This model identifies the major artifacts in the development of computer-based systems and their interrelationships [13]. The constituents of this model are: Process, Architecture, Information, and Methods and Tools, as portrayed in Figure 1.

For quite some time, the software-research community has spent a lot of effort to identify and specify advanced process models, especially models of the systems development process. Therefore, we have access to a wide variety of process models, supporting tools and related information models. However, the critical point is that complex systems, covering larger and also more complex application domains, imply the need for a higher level of coordination and integration that is related to the conceptual level of software design [21]. This aspect of high level software design and specification has been neglected for too long and has only

---

<sup>1</sup>IEEE Technical Committee for Engineering of Computer-Based Systems

lately been recognized under the term “domain specific software architectures”.

Placing the process and the architecture element on the basis of the ECBS model signals their outstanding importance for the engineering of computerized, software-driven systems. Tools and methods, as well as information models, can be successful only if a clear process and a corresponding compatible architecture model are specified first. Even more important, process and architecture must have a balanced relationship to ensure that the framework does not dip over to one side and enforces process standards without integrating well defined architectural concepts.

Once a domain wide architecture is in place, issues like project coordination, interoperability, design reuse, maintainability, extensibility, and long life-cycles can be easier tackled and have a better chance to be successful [5], [18], [22].

The idea of building a corporate culture by establishing a common level of “best practice”, well know and widely used in other industries, is also directly supported by the architecture concept and provides for possible improvement in problem areas that are dominated by organizational and social issues [8], [11].

As systems become more complex, the high-level organization and behavior of the system, the system architecture, and its balanced relationship with the development process become the critical aspects of the ECBS model. Recognizing this fact, the IEEE ECBS TC has, among other things, been working on a specification of the architecture constituent within the structural model and published its preliminary re-

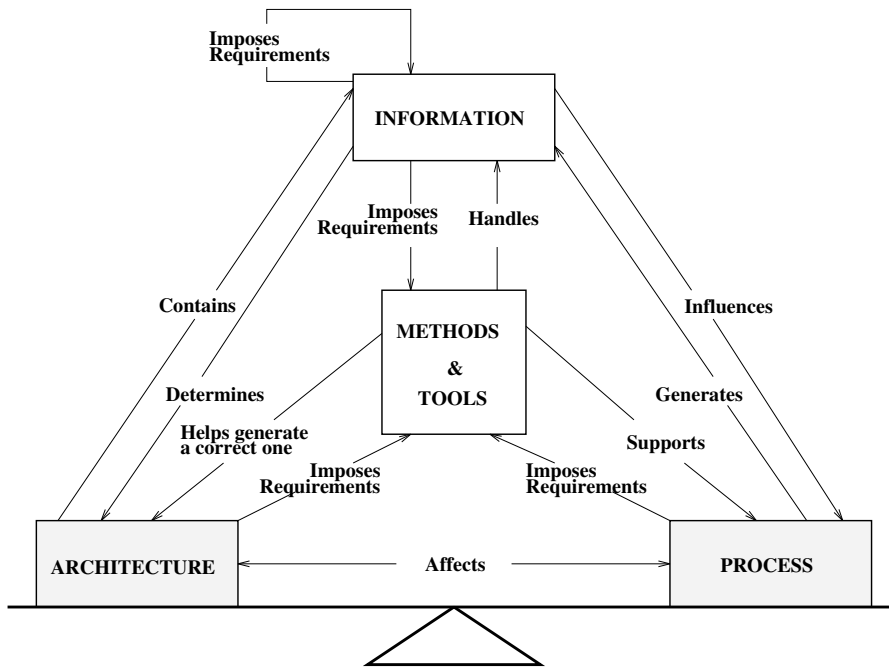


Figure 1: The ECBS Constituents

sults within the structural model. Furthermore, the use of an architecture during software and systems development has been discussed.

In the current paper, we build upon these first results by further refining the existing characterization of the ECBS architecture element and exemplifying the usage of the modified approach by integrating the architecture concept into an enhanced development process. While we work within the official ECBS framework and are actively involved in the ECBS architecture working group, we want to point out that the models discussed in the following sections are the results of work on our own Generic Systems Integration Framework (GenSIF) and, therefore, go beyond the currently officially adopted IEEE ECBS TC point of view and may not be compatible in all aspects.

The rest of the paper is organized as follows: In

section two we give a short introduction to the principles of the above mentioned GenSIF framework and its process model. This introduction sets the stage for a more elaborate discussion of our generic architecture reference model in section three.

## 2 GenSIF - A Generic Framework

### 2.1 The GenSIF Principles

We are developing GenSIF [18], [19], [20], [23], within the wider context of the ECBS problem specification as a generic framework for understanding the software development and integration issues in domain specific, large scale systems development. GenSIF is focused on the software aspect of computer-based systems and

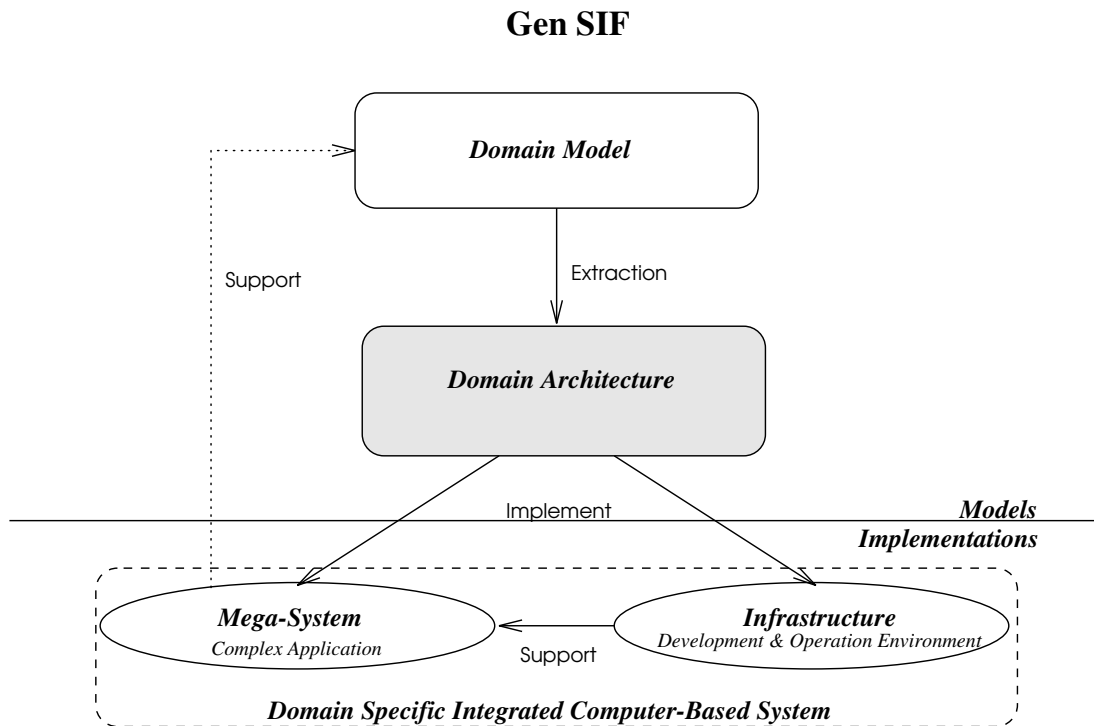


Figure 2: The GenSIF Framework

addresses mostly the architecture and process components of the ECBS structural model.

We define a business domain as a part of the real world which is a self-contained space of human action with well defined functions and services offered to society. Banking, avionics, and telecommunications are all examples of business domains. The richness of activities and the diversity of operations in the domain bring to life a number of computer-based systems, e.g., office systems, command and control systems, information services, image processing, automation, etc. The combined functionality of these systems constitutes a so called “system of systems”, one type out of many possible mega-systems [23].

The system of systems concept has originally been defined by H. Eisner [5]. The basic architectural prin-

ciple, as used in GenSIF, is that of a distributed set of independently developed and functioning systems, so called “building blocks”, which interoperate within the context of an overall, coherent mission in the given business domain. Interoperation can be achieved by a variety of means, e.g., transparent message passing or using a blackboard element, and must be specified in the architecture description.

Our research is focused towards the delineation of new techniques and artifacts in the development process which will assure that a domain specific collection of independently developed or already existing building blocks and systems will interoperate in such a large and complex system of systems. As a consequence, software development in GenSIF is based on three elements of coordination and domain standardization

[19], see Figure 2:

- **Domain model** - a semi-formal description of the domain that forms a comprehensive knowledge base and influences the outcome of all development and integration initiatives in the domain [20];
- **Domain architecture** - a unifying and coherent, multilevel software architecture specification which is used as a guiding plan in any development effort within the domain. It limits the design choices at lower design levels and supports interoperability and reuse in the presence of separate projects [9], [10];
- **Infrastructure** - a uniform development and operations-support environment. It provides a set of generic services to the building blocks and implements a set of common functions defined by the architecture [23].

The core of the framework is the domain architecture and, as such, has a strong effect on the entire development process. The domain architecture promotes technological standardization in the domain and acts as de-facto application standard. It usually supports the philosophy of an open and distributed system of systems while safeguarding integrity and semantic consistency at the same time. Such an architecture serves as a frame of reference and conformity and in this way guarantees that the target megasystem and its constituents will meet their functional and non-functional requirements from a domain-wide perspective.

To achieve these goals, an architecture is based on the analysis of a comprehensive domain model and imposes requirements on a so-called infrastructure, a standardized set of tools and common services. The domain model and the infrastructure are, respectively, prerequisite and consequence of an architecture. They cater to the needs of the architecture and are, in a typical feedback process, driven by the architecture's structure and concepts [18].

Domain architecture in GenSIF is, therefore, a domain-specific design model targeted at systems integration. It is specified at two levels as a

- **Conceptual architecture** and derived
- **Application architectures.**

The conceptual architecture is a multi-layered, generic design model, constrained by the specifics of the domain. It is specified by using a generic architecture reference model (see Section 3).

A conceptual architecture prescribes the cooperative behavior of the target systems in terms of generic system structures and behaviors, but does not include any application specific elements. In each project within the given domain, the conceptual architecture is then used as a design template and instantiated into an application architecture.

In the case of a system of systems, the conceptual architecture would just prescribe the form of communication and control used, the types of components that can be identified as being useful on a domain wide basis, etc. Application machines [12] and the

OSCA <sup>2</sup> architecture by Bellcore [16] are examples for this type of conceptual architectures.

To illustrate some of the ideas of a conceptual architecture, let us shortly look into the OSCA architecture. (We would like to point out at this point in time that we use the OSCA architecture and other examples to illustrate some of our concepts and to show that they are already, even so informally, used in daily practice. However, the reader is advised that these examples constitute our own interpretation of published documents and applied test-cases.)

The OSCA architecture, one of the few published cases of an industrially applied conceptual architecture model, is targeted at the domain of telecommunication services and information systems in general.

It identifies only three types of components/building blocks, namely user-layer (ULBB), processing-layer (PLBB), and data-layer building blocks (DLBB). It knows basically only one type of communication, i.e., invoking a service offered by a building block of any type by activating a so-called contract (an enriched interface).

Data-layer building blocks manage the semantic integrity of data that is of importance to the whole domain (company), so-called corporate data. They are the stewards of globally relevant information and have the exclusive right to provide access and update functionality for corporate data. (All types of building blocks can have also harbor private data, not accessible to other building blocks.)

---

<sup>2</sup>OSCA is a trademark of Bellcore - Bell Communications Research

User-layer building blocks provide interaction with the system's environment, especially humans. They allow an end-user to access functions provided by the system of systems that is formed by all accessible building blocks.

The processing layer, formed by the processing-layer building blocks, implements all functionalities that do not belong to either the data-layer or the user-layer. This includes typically functions to support management and operation of business.

Once a generic conceptual architecture has been specified, it can be used to develop a multitude of application systems (of systems), each one of them adhering to the conceptual architecture, but refining it at the same time into a project specific application architecture. This is done by instantiating the generic concepts, rules and principles defined in the conceptual architecture into application specific structures and interaction patterns.

An example would be to develop a system of inter-operating building blocks that provides telephone directory services, connection establishment, and billing in New Jersey and Delaware. Each of the externally visible functions might use a set of building blocks of varying type, e.g., a DLBB to steward the telephone directory data for Morris County and a ULBB to make that information accessible via touch-tone dialing.

As all projects develop their solutions as a system of systems, or to be precise, a system of building blocks, using the same conceptual architecture to derive their application architectures, the domain is served by a steadily growing, open network of interop-

erating building blocks instead of by isolated, monolithic systems with a monopoly on certain junks of data or functionality.

For a detailed discussion of conceptual and application architectures see [9], [10], [23]. However, the principle is simple: Conformance to the conceptual architecture guarantees that building blocks will exhibit the required properties that allow them to fit into a system of systems within the business domain that offers “more than the sum of its parts”.

For the rest of this paper, we will focus on the use and the specification of conceptual domain architectures. In the next subsection we go into some more detail regarding the software development process and the role of a conceptual architecture in this process.

## 2.2 A Brief Overview of the Process Model

GenSIF is oriented towards a solution for a complete business domain, while systems development is project oriented and focuses on single customer groups within the domain.

To bridge this gap, the development process in GenSIF strives to leave the project teams as much freedom as possible but to provide at the same time domain wide integration measures for project coordination. Each project can then pursue its own organization and optimization effort, as long as it complies with the domain architecture which guarantees the interoperability of the developed systems/building blocks.

The focus is on planned-for integration which prepares systems for later integration without assuming

to know all (future) parts of the mega-system nor all possible requirements in the domain. This is opposed to other approaches, like post-facto integration, which strive to resolve problems in systems integration of uncoordinated, non interoperable systems, and complete pre-facto integration which assumes to be able to plan everything in advance and, thus, relies on complete requirements and fully detailed system design.

Variations of post-facto integration are seen as a part of GenSIF in resolving the problem of legacy systems in the application domain, but are not further discussed in this paper.

Keeping projects as independent as possible and coordinating and integrating them at the same time, leads to a development approach that has two levels:

- **Domain level** and a sub-ordinate
- **Project level.**

While projects look only at their limited problem space inside the domain, the domain level takes the whole application domain into account: What requirements are for a project, that is the domain model for the domain level. While single systems are traditionally implemented according to a locally optimized design, building blocks in GenSIF derive their design from the domain architecture. While tool support in uncoordinated projects is centered around local environments, a domain architecture allows the engineer to provide a standardized infrastructure as a common tool platform.

Figure 3 describes the process model in the form of an extended flow diagram, relating tasks, models, and

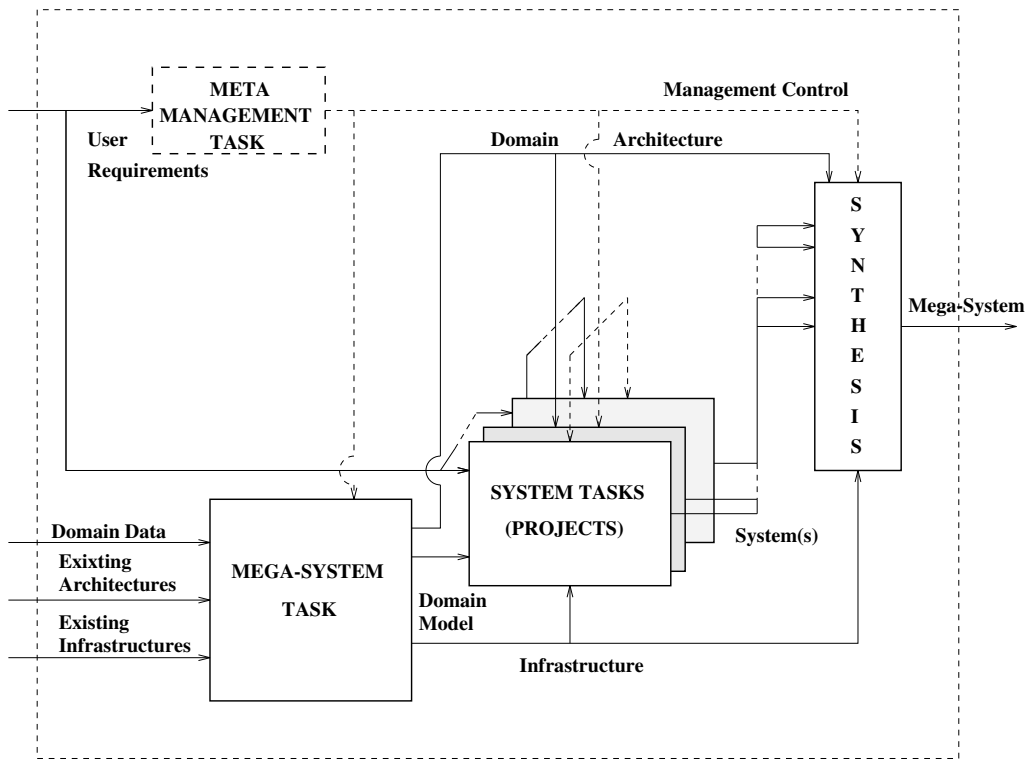


Figure 3: An Architecture Driven Development Process

results. The process model, as presented in this paper, is a simplified version of what is discussed in more depth in [23] and [19]. It includes only a discussion of the most important tasks and concepts involved in the development framework and has been brought into a compacted format regarding data and control flows.

Building blocks, or closely related sets of building blocks, are developed in projects. Due to the nature of domain wide thinking and development, and the architectural concept of a system of systems, multiple projects can be active at the same point in time, can overlap in their timing, or can be related in a sequential manner.

Projects represent the traditional system development process, transforming requirements into user-

specific systems. To support these projects and to guarantee an integrated result (a system of systems and not a set of unrelated systems), the domain model, domain architecture and infrastructure are used. All projects are controlled by a meta-management task activity.

The domain model allows the project team to handle requirements and to communicate knowledge about the domain on the basis of a standardized common model. The domain architecture provides a design reference model used to guide projects during their internal design activities. The technical infrastructure supports the implementation phase of the project by providing a standardized set of tools and technologies which are shared by all projects.

Domain architecture, domain model, and infrastructure are specified and maintained by the mega-system task. This mega-system task complements the individual projects by establishing coordination on the domain level.

A synthesis activity derives a consistent system of systems from the set of single systems produced by the projects and takes care of issues like processor assignment, redundancy management, network load coordination, etc.

### 3 A Generic Architecture Reference Model

#### 3.1 Architectural Characterization

Having briefly discussed the use of architectures in an integrated development process, the next step is to tackle the problems of architecture specification and description.

To facilitate this task, we propose the use of an architecture reference model. Beyond the already discussed use of an architecture reference model during systems development, such a model also helps to foster the understanding and analysis of software architectures by establishing a standardized terminology (ontology), providing a common theory of architectures in general, supporting the categorization and comparison of architectures, facilitating reverse engineering and reuse of architectures, and providing a basis for the development of architecture support tools [17], [1], [7], [18], [9], [10].

In the context of engineering of computer-based systems (ECBS - Figure 1) and as used in the GenSIF framework, an architecture model must address:

- The relations that bind a system architecture to the corresponding development process.
- The relations to the information model employed.
- The relations to supporting tools.
- The corresponding body of applicable engineering knowledge and design rationale (concepts and rules).
- The set of constructive elements (and notations) used.

The generic architecture reference model, as the basis for a domain specific conceptual and application architecture, must be compatible with and integrated into the corresponding development process (see Section 2.2 for the GenSIF approach to this problem).

Information models are useful to make knowledge and structure explicit that is incorporated in the process and the architecture model. An example for this type of information model, in the ECBS context, is given in [14]. However, as the architecture model is typically underspecified in most existing frameworks, as compared to the process model, more work on architecture models will be necessary before clear relationships to the information model can be established.

Another type of information that is vital to architecture specification is domain knowledge, as captured in GenSIF in the domain model [20]. In this case, the domain model supports the architecture specification

by providing the characteristics of the business domain which can be used as parameters in identifying a compatible domain architecture [19], [20].

In such a model-driven approach, tools are the logical consequence of the domain architecture, while in many cases existing tools impose a de-facto architecture without appropriate architectural design or documentation. To support the tool aspect, every architecture and, therefore, every complete architecture reference model, must provide an interface that imposes clear requirements on tool developers, e.g., in listing typical architectural elements that need to be supported and by specifying notations to capture them.

Beyond the just listed relations to other components of a larger framework, like the one provided by ECBS, an architecture reference model must capture the goals, the design rationale and the major design decisions that have been set as guidelines for the development of software in a specific business domain. As an additional bonus, they can also serve as a medium to communicate engineering knowledge by providing a focus and a framework to organize and describe what is usually called “good practice”.

In other terms, an architecture model must be able to specify on an abstract, generic level the basic properties that must be guaranteed, and system organization and behavior in terms of components, interactions (connectors), and static and dynamic configurations. Similar definitions can be found in [6], [15], [4].

Based on the above characterization of architectures, we propose a generic software architecture reference model for GenSIF. This model is a refinement

of the current ECBS architecture model and an integration of the concepts and ideas we have considered and formed during the many hours of discussion with colleagues from academia and industry.

The most important influences on the GenSIF model have been, without any doubt, the OSCA architecture [16], as proposed by the Bellcore team around John Mills, and the application machine concept, as put forward in [12]. Both architectures have been (and are) widely used in practice and have been tested in case-studies in our labs. The GenSIF architecture reference model is, thus, to a certain extent a reverse-engineered model, based on a set of existing architectures and studies.

The suggested GenSIF architecture reference model, see Figure 4, has two parts:

- The first part captures the design rationale and is a collection of **concepts, rules, principles and guidelines** [2], [3], [15], [10].
- The second part introduces the constructive portion of the model, the **architectural elements** [9], [10].

### 3.2 Concepts, Rules, Principles and Guidelines

The notion of concepts, rules, principles and guidelines has been established in the ODP [15] and ANSA [2] projects. Their specifications in the context of GenSIF are as follows:

- **Concepts** include the required system properties the architecture must support and the set of

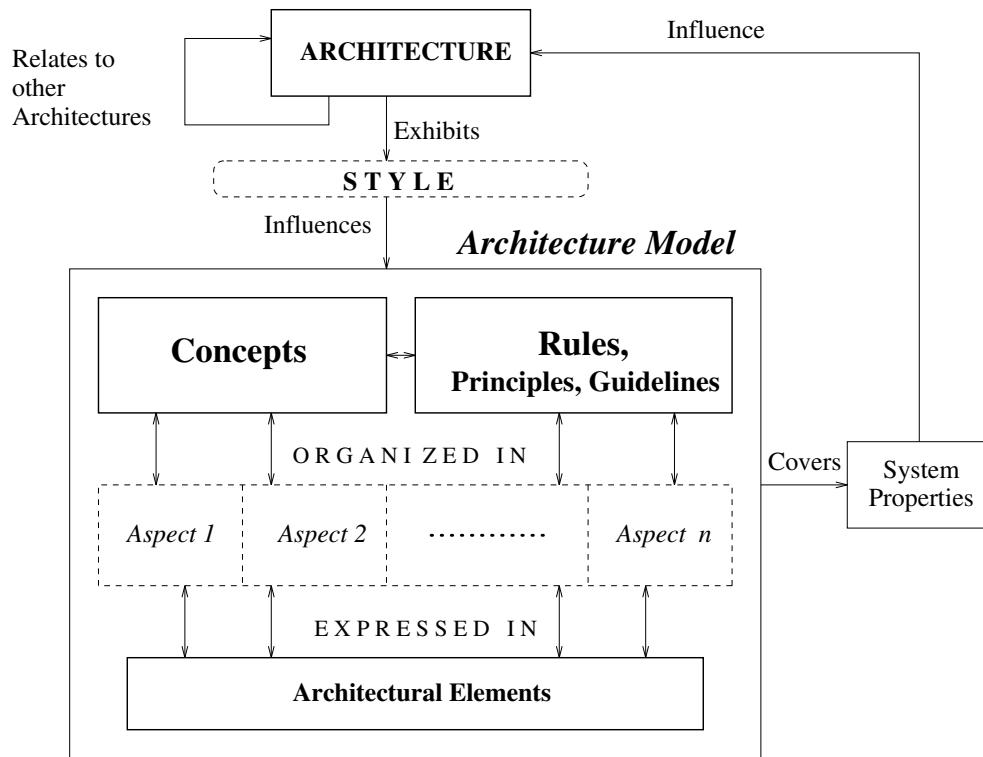


Figure 4: The Generic Architecture Reference Model

available constructive elements to form systems.

- **Rules** limit the set of possible interactions and structures. They impose constraints on how elements may be combined to form systems.
- **Principles** list and evaluate useful structures and patterns that help to meet particular needs while staying within the boundaries of concepts and rules.
- **Guidelines** provide general hints and techniques to stay out of trouble when using concepts, rules, and design principles.

**Concepts** are decisions about the many different aspects of an architecture. One way to structure these aspects is to make a separation between

- Properties and
- Constructive elements.

Properties are goal statements that are derived from the characteristics of the business domain(s) the architecture will serve. These properties should be reflected in all systems that are developed in the domain. The conceptual architecture, driving all design efforts in software development projects, is responsible to uphold and implement these properties.

Typical properties include characterizations of the following sub-aspects: security, performance, reliability, robustness, user-friendliness, etc. In other terms, we speak about the non-functional, or quality aspects of systems. A good book on software engineering can provide a very comprehensive list of these aspects and

even provide a discussion of relations and dependencies between them.

However, properties may also state more constructive goals, e.g., transparent message passing. This is necessary if the given constructive property is not only a supporting element for a quality, but is considered to be crucial to the architecture’s characteristics. It is considered to be a basic concept, and not only a means to achieve another concept. For an open systems of systems, for example, “location transparent message passing” is a basic concept.

Properties may later-on be further refined and discussed when the rules for the architecture are specified. This will allow the architecture to implicitly enforce properties by limiting the use of its constructive elements to use in a set of approved structures only.

“Security of invocation” in an open, distributed system, especially in the presence of transparent communication channels, is a typical example for a property that might be mapped into rules (see below).

Another way to realize properties is to implement them directly in the supporting infrastructure. In this case, the documentation of required properties in the architecture model becomes essential to avoid the loss of an architectural concept by implementing it on the code level without any higher-level documentation.

An example for a typical candidate for direct implementation is the concept of location transparent message passing, e.g., with the mechanisms provided by [2].

Constructive elements are the second major aspect of architectural concepts besides properties. They

cover the classical principle of system design via identification of general design elements. In the GenSIF architecture reference model we use the following classification, which is based on the current agreement within the ECBS architecture working-group:

- Components,
- Connectors, and
- Types of abstraction.

Components are the elements that form the designer’s mental model of a system in operation. They are the sources of activity and serve as units of organization of a system’s architecture.

Components can come in a variety of types and specializations, e.g., building blocks of a system of systems, objects, data capsules, filters, subroutines, tasks, etc. They can be of a very general nature, e.g., building blocks in the OSCA architecture [16], or highly specialized, as in application machines [12].

Even if based on a simple and general concept, components of an architecture are often subdivided into different types with the same basic layout but different capabilities and specialities.

As an example, let us consider Bellcore’s OSCA architecture [16], as we are using it in our case-studies at NJIT. In this architecture, components are called building blocks.

A building block is specified as a cohesive set of functions, or to be precise, a cohesive set of functions characteristic of the given (business) domain.

Building blocks are then classified into three types that serve three distinct purposes (see also Section

2.1): user-layer building blocks (ULBB), processing-layer building blocks (PLBB), and data-layer building blocks (DLBB).

The basic OSCA architecture document [16] specifies building blocks by listing (in plain English) a set of properties that must be guaranteed by all building blocks and then adds on additional properties and rules to discriminate between different types of building blocks.

Examples for properties (and associated rules) shared by all building blocks are: release independence, location independence, infrastructure independence, handling of interaction by contract only, etc.

Examples for DLBB specific properties and rules include: handling of semantic integrity of corporate data, redundancy management, openness to all authorized queries, employment of an implementation independent information model, etc.

However, components of different types are only one of the necessary constructive elements to form an architecture. In addition to components, connectors are necessary to facilitate interaction between components and to form executable structures [6].

As with components, types and characteristics of connectors can vary widely and include solutions like messages, procedure calls, RPCs, buffers, event broadcasts, external interfaces, contracts, etc.

The only type of connector used in the OSCA architecture is a so-called contract. In our flavor of the architecture, a contract is a typical data-capsule interface, enriched with additional managerial and technical elements, and activated by a message. It offers

functionality on a transparent, location independent basis and can be directly invoked by other building blocks (of any type).

Contracts guarantee features like a restricted set of syntax encodings, isolation from building block internals, release independence, location independence, recognition of invocation authorization, security audits, etc. [16].

Another, and very comprehensive, set of connectors has been identified and documented in the DORIS architecture used by British Aerospace [22]. DORIS targets architectures in the real-time and, especially, aerospace oriented domain.

The DORIS principle is that intermediate data, representing connections, should be specified explicitly. Furthermore, the only way to handle that data is to destructively or non-destructively write or read it.

This leads to four major types of connectors in DORIS: signals, pools, channels, and constants. Each one of them is seen as an available implementation of a communication protocol, where a pool, for example, is characterized by destructive writing and non-destructive reading (reference data).

A much more basic classification of connectors, used in GenSIF, divides them into data-carrying connectors, control-oriented connectors, and the many possible hybrid forms that exhibit both characteristics to some degree.

Messages, for example, are dominantly carriers of information/data, but also encompass some control aspects by triggering execution of services in the receiving building block.

Components and connectors describe the structure of a system on one level of abstraction. To be able to handle more complicated architecture models, many levels of abstraction can be employed. There are typically at least two major forms of relationships that describe the abstraction aspect:

- Aggregation/Decomposition
- Generalization/Specialization

These two types of abstraction allow for the construction of hierarchies, based on only two basic relationships, “PART-OF” and “IS-A”. To be useful in an architecture model, it will however be necessary to specify precisely the characteristics of the relationship.

The use of hierarchies is not as extensive in many of the conceptual architectures we have been studying [16], [12], as it is in basic software system design. The reason for this observation might be that some architectures are designed on purpose to be very simple and “flat”, and exhibit anyway only a handful of component types and connectors.

However, there are also examples of architectures like DORIS [22], where hierarchically organized levels of abstraction and the transformation between levels is a central issue.

Not all of the **aspects** that have been used in this reference model to specify the possible alternatives and typical solutions for concepts will be addressed in every architecture. However, establishing a grid of aspects has a value in itself, as it can function as a checklist for architecture completeness and support the development of rules, principles and guidelines that com-

plement the given concepts.

**Rules** simply specify constraints on concepts. They make it more likely that by avoiding specific situations and prescribing a limited set of other solutions, a system developed according to the conceptual architecture will exhibit the necessary properties. They constrain the freedom a designer has to make choices when constructing a system, but support at the same time reuse of high-level design concepts and facilitate tool support via a standardized infrastructure.

Rules can be targeted towards the constructive, structural aspect of concepts, being concerned with possible combinations of components and connectors, or the formation of layered structures based on certain types of relationships:

A typical structural rule, for example, could specify what type of connector can be used to achieve communication between what types of components: Message passing between user-layer building blocks and processing-layer building blocks is ok. Message passing between processing-layer building blocks and data-layer building blocks is ok. However, there can be no direct communication (message passing) between user-layer building blocks and data-handling building blocks. (These rules do not hold for the original OSCA architecture, but were used in one of our case studies to specify a derived conceptual architecture.)

Another rule might refine the concept of building classification hierarchies of components by restricting inheritance to single inheritance (out-ruling structures with multiple inheritance), or by allowing for classification, but banning inheritance completely (A step

many system architects consider now, being insecure if inheritance will scale-up to large and complex systems and their maintenance).

Rules can also focus on the properties aspect of an architecture, by translating abstract properties directly into attributes of components and connectors.

As an example, let us consider a situation where each message that is received by a building block of any type must go through sender authorization checking before any services of the receiving building block are activated.

Such a rule would be the direct mapping of one aspect of an architectural concept, i.e. security of location transparent communication, into the world of rules. It would also have an impact on the implementation of every building block in the system, making it necessary to include sender authorization into the basic “template” each building block is derived from.

Concepts and rules are a necessary minimum to describe a complete architecture model. They must be obeyed by the system designer. Any design not complying with concepts and rules must be considered to be outside the architecture. This means also that concepts and rules must be stated first when developing an architecture and are the backbone of any further development.

**Principles** are of a very different nature. They are compiled design experience, expressing effective ways to do things while remaining within the rules. They describe what is considered to be “good practice”, a notion used in many other engineering disciplines. It is not mandatory to follow principles, however, their

(re-)use reduces design effort and guarantees practical solutions of good quality.

In the context of software development, principles correspond to what has lately been called “software design patterns”. They offer solutions to application or situation specific problems in a generic, adaptable way.

A good example is the use of a transaction handler component (a special form of processing-layer building block) to achieve atomic transaction capability in systems with otherwise distributed control structure. (It is of course possible to construct also much more elaborated patterns that cover a much wider range of issues.)

**Guidelines** try to anticipate problems that can be encountered when designing systems within the given set of concepts and rules. In many cases they can be subsumed in the architecture principles. They are of a less constructive nature than principles and are more concerned with helping to avoid known pitfalls. Thus, they read many times like the well known problem-reason-solution tables we know from most manuals.

Another way to look at guidelines is to regard them as the accumulated knowledge and “folklore” that starts to build up during actual use of a conceptual architecture. While guidelines are highly interesting and helpful, they do not exhibit the same technical completeness and rigor as rules and principles. However, who has never asked for help in one of the on-line news-groups and was happy to get at least a hint of what might be wrong or should be done?

An example for an architecture related problem situation could be that during times of high system load messages between building blocks get lost. If we have a rule that enforces sender authorization checking in each building block receiving a message (see above), that rule might very well be the reason that a bottleneck is created in most implementations of the message handling mechanism, leading to an overflow of the queue for incoming messages each building block has to maintain. Possible hints at a solution would be to increase the size of the queue, execute authorization in parallel with other building block functions, etc. (Remark: It is of course not possible to drop authorization, as it has been stated as a rule and is, therefore, mandatory!)

### 3.3 Architectural Elements

Concepts and rules provide for a complete specification of a conceptual architecture. Organized in aspects, they may be described in natural language, currently the most common solution, or in any suitable formalism. Regardless of notation, however, they exhibit a clear focus on system semantics and non-constructive, rule-based characteristics.

The question, however, is, who are the “consumers” of our conceptual architecture model and what do they expect?

There are two major groups that have a vested interest in the conceptual model: the designer working in the context of a software development project, and the tool manufacturer who wants to supply the architecture’s development and execution environment

(infrastructure).

Both groups use the conceptual architecture as a generic template that allows them to instantiate an appropriate application architecture or a tool design, respectively. In both cases, and in all mentioned activities, the focus has now shifted from a rule-based type of thinking, putting semantics first, to a more constructive type of behavior, where the usual working paradigm is that of connecting predefined types of elements into an application specific solution.

This also includes a shift to a more syntax, notation oriented world-view. Similar to the typical programmer who organizes his knowledge about a specific language around the syntax diagram and looks into the specifications of semantics only if a problem occurs, the typical system designer will organize his mental model of the architecture around a much more constructive and notation oriented perception.

Therefore, even though concepts, rules and principles alone form a complete specification of a conceptual software architecture, they are ill suited for the day-to-day handling of the architecture model and are not very intuitive to use. To accommodate a more construction oriented perception of an architecture, we suggest providing another representation of the conceptual architecture model that hides rules and calculus behind an easy to use point-and-click, icon-driven interface. It is then up to the end-user to choose which view of the architecture, the rule-driven or the icon-driven should and will be used in a specific situation.

Furthermore, it can be argued that notations should be domain specific and application oriented,

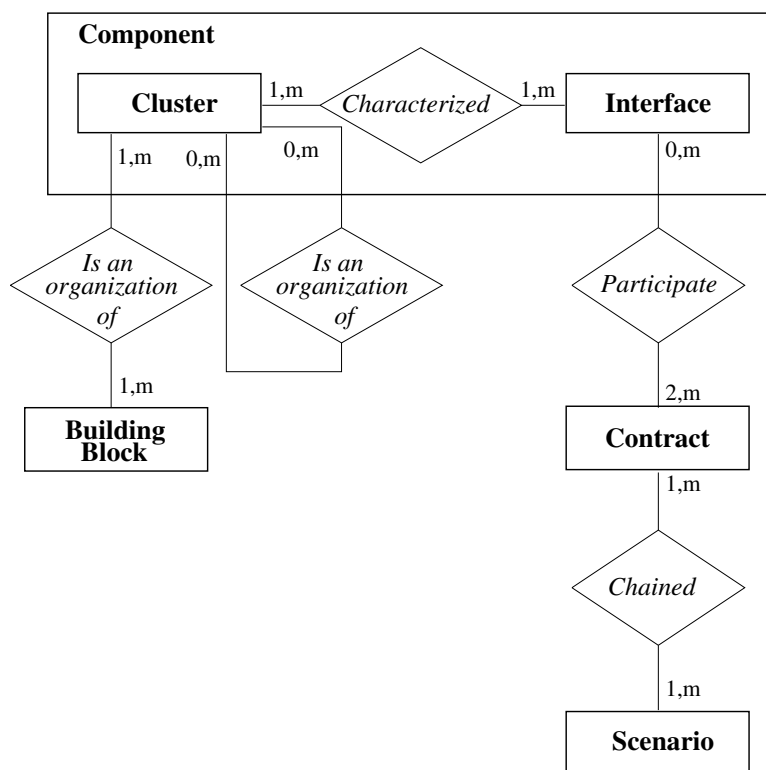


Figure 5: Common Architectural Elements

supporting the application engineer, e.g., as proposed in the application machine concept [12], and not the architecture engineer.

If we accept this concept, the logical consequence is to allow for multiple notations even for one conceptual architecture. Concepts and rules represent the architecture’s essence, while a constructive, icon-driven view caters to the field engineer.

What we can do to support the development of domain/customer specific notations for the architecture is to provide an interface that makes it easier to achieve the transition from concepts and rules to a notation with construction oriented characteristics.

In GenSIF we provide a set of so called architectural elements, Figure 5, for this purpose. Architec-

tural elements repeat most of the aspects captured in concepts and rules in the form of a set of generic elements. These elements form an ontology, listing typical elements that can be expected to participate in a syntax and construction oriented representation of a conceptual architecture.

Such an ontology will help to unify vocabulary, compare alternatives, and check for completeness of notational solutions, etc., very much like the architecture reference model supports these issues for architectural models. It is also helpful to decide on a primitive set of icons/graphics than can than be refined as necessary.

The generic architectural elements, specified below, were selected upon considerations to cover the

architecture reference model as completely as possible. Their applicability to the modeling of the static and dynamic structure of a software system [8], [9], [15] has been the other driving force. However, they still embody only one possible choice to map concepts, rules and guidelines into a user-friendly interface.

- **Building Block** - a structural primitive of the architecture, representing processing and data management. It is an autonomous unit.
- **Component** - an organization, **Cluster**, of building blocks and/or components with defined external behavior.
- **Interface** - a component abstraction that represents a component's expected external behavior.
- **Contract** - a set of requirements and constraints on one or more components, prescribing a collective behavior, derived from the participating interfaces.
- **Scenario** - dynamic configuration of architecture components.

Building block, cluster and component relate directly to the component and layering aspect of the reference model. If there is no decomposition hierarchy or any other form of clustering, e.g., as in the application machine architecture [12], the only element necessary is the building block.

In a flat system of systems, building blocks (the component concept) map directly into building blocks (the architecture element), as they are the only form of

component the architecture knows. Types of building blocks can be mapped in the same way.

Be aware so, that a building block in the sense of an architectural element now adds on to the specification given in the conceptual architecture, e.g., by linking the rule that “all incoming messages must be checked for authorization” (see above) directly to the component concept and, in a later step, by providing a notational primitive for “building block” that symbolizes all information about component type, associated properties and linked rules in one icon. (Such an icon may finally be supported by the infrastructure in the form of a code-skeleton or template, etc.)

The component interface is the place to document and specify this merger of information that stems from several aspects in the reference model. In many tools, building blocks, clusters and components will become graphic icons, while the interface becomes the typical component specification (that should include ways to express constraints and properties!).

Clusters provide for the mapping of abstraction types (PART-OF, IS-A). On the level of architectural elements, this mapping allows for the specification of notational primitives that manage layering in system design (in the application architecture). Sometimes clustering is implementation specific, e.g., by aggregating a set of objects running on the same platform into an execution cluster.

A contract is the equivalent of the connector concept. As with components, a contract packs all associated properties and rules into one unified architectural element.

Following-up on our example of structural rules, specialized icons might then express that an user-layer building block can connect to a processing-layer building block, but not to a data-layer building block. This can be done by specifying graphical icons for contracts that “fit” only if all rules are obeyed, i.e., the user-layer building block can be connected only to contracts which in turn can be connected only to processing building blocks.

Scenarios provide the ability to chain many components and many contracts together to form complete execution structures, modeling the dynamic aspects of a system. In many cases it might also be helpful to use scenarios to store complete design patterns, suggested in architecture principles, or to show how a thread of execution runs through a complex system, making use of and obeying concepts and rules.

To summarize, architectural elements do not simply repeat concepts and rules, they support the development of domain specific notations by providing another way to represent the conceptual architecture. Architectural elements are “incomplete” in many aspects.

They are not a notation, they just make it easier to reason about a good notation. Architectural elements act like a set of predefined classes which can be further specialized and finally instantiated into a solution. However, they allow for the specification of basic generic functions early on in the process, e.g., by identifying the commands necessary for an architecture editor in order to be able to manipulate the elements of an architecture.

Architectural elements do not cover all architectural properties, especially qualities. While it might be possible to “code” a lot of qualities and rules into a syntax oriented representation, the decisive model is given by concepts and rules. Even if a contract embodies a connector type **and** associated rules, other rules might not be mapped into the syntax and must be listed separately.

It always holds that the concepts and rules cover all aspects of the representation that is expressed in terms of architectural elements. The elements based representation, however, may be only an incomplete mapping of the concepts and rules (even though it is easier to communicate to application engineers and designers).

## 4 Summary

We have proposed to establish a reference model for architecture specification and development. Such a model is organized around a set of aspects which structure concepts and rules which, in turn, specify a conceptual architecture. Principles and guidelines are added on to concepts and rules to give a more complete picture of the architecture and to provide a place to store and communicate successfully applied design patterns and other knowledge related to the architecture. To make a step towards a more constructive type of architecture representation, architectural elements have been introduced.

Currently, our research is focused on a further refinement of the concepts presented in this paper, es-

pecially a formal specification of the architecture reference model. We also continue to run case studies to test our ideas, e.g., by applying our model to the OSCA architecture and the application machine concept, as we have done in the past. A prototype for an architecture editor is in the first stages of development. A multitude of tools are being tested to learn more about the prospects of integrating them into a real infrastructure and to understand what the typical services an infrastructure must provide.

## References

- [1] Abowd, G., Allen, R., and Garlan, D., "Using Style to Understand Descriptions of Software Architecture", in *Proc. of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM SIGSOFT Software Engineering Notes, Vol. 18, No. 5, December 1993, pp. 9-20.
- [2] ANSA Reference Manual, Vol.A, Architecture Projects Management Limited, UK, 1989.
- [3] ANSA Architecture Report - The ANSA Computational Model, Architecture Projects Management Limited, UK, 1991.
- [4] Boehm, B., Scherlis, W., "Megaprogramming", in *Proceedings of DARPA Software Technology Conference*, Los Angeles, California, 1992, pp. 63-82.
- [5] Eisner, H., Marciniak, J., McMillan, R., "Computer Aided System of Systems (S2) Engineering", in *Proceedings of the 1991 IEEE/SMC International Conference on Systems, Man, Cybernetics*, Charlottesville, VA, IEEE, Computer Society Press, October 1991, pp. 531-537.
- [6] Garlan, D., Shaw, M., "An Introduction to Software Architectures", in *Tutorial Notes: Architectures for Software Systems*, ICSE 15, Baltimore, Maryland, May 17-21, 1993, pp. 61-99.
- [7] Garlan, D., Allen, R., Ockerbloom, J., "Exploiting Style in Architectural Design Environments", *Software Engineering Notes*, Vol.19, No.5, December 1994, pp. 175-188.
- [8] Hammer, D., "The Development of Large and Complex Software Systems: A Purely Technical Issue?", in *Proceedings of the Computers in Engineering Symposium, ETCE'94*, New Orleans, LA, USA, January 1994, pp. 9-18.
- [9] Kirova, V., Rossak W., "Some Thoughts on Architecture Engineering", in *Proceedings of the Computers in Engineering Symposium, ETCE'94*, New Orleans, LA, USA, January 1994, pp. 59-68.
- [10] Kirova V., Rossak W., and Jololian L, "Software Architectures for Mega-System Development - Basic Concepts and Possible Specification", *Proc. of the IEEE Third International Conference on Systems Integration*, Sao Paulo City, Brasil, August 1994, pp. 38-45.
- [11] Lawson, H.W., "Philosophies for Engineering Computer-Based Systems", *IEEE Computer*, Vol. 23, No. 12, December 1990, pp. 1859-1874.
- [12] Lawson, H.W., "Application Machines: An Approach to Realizing Understandable Systems", *The Euromicro Journal*, Vol. 35. No 1-5, September 1992, pp. 5-10.
- [13] Lawson, H. W., "Introducing the Engineering of Computer-Based Systems", in *Proceedings of the 1994 Tutorial and Workshop on Systems Engineering of Computer-Based Systems*, IEEE Computer Society Press, Los Alamitos, CA, May 1994, pp. 2-8.
- [14] Oliver, D.W., "A Draft Integration of Information Models: Component Model and Oliver Model", *Proceedings of the 1994 Tutorial and Workshop on Systems Engineering of Computer-Based Systems*, IEEE Computer Society Press, Los Alamitos, CA, May 1994, pp. 44-96.
- [15] Basic Reference Model of ODP - Part 2: Descriptive Model, ISO/IEC JTC 1/SC 21, Interim revised CD text, 1992.

- [16] The Bellcore OSCA Architecture, Bellcore - Bell Communications Research, Technical Advisory, TA-STS-000915, Issue 3, March 1992.
- [17] Perry, D., and Wolf, A., "Foundations for Study of Software Architectures", ACM SIGSOFT Software Engineering Notes, October 1992, pp. 40-52.
- [18] Rossak, W., Ng, P.A., "Some Thoughts on Systems Integration - a Conceptual Framework", International Journal of Systems Integration, Vol.1, No.1, Kluwer Academic Publ., Dordrecht, The Netherlands, 1991, pp. 97-114.
- [19] Rossak, W., Zemel, T., Lawson, H., "A Meta-Process Model for the Planned Development of Integrated Systems", International Journal of Systems Integration, Vol.3, No.3, Kluwer Academic Publ., Dordrecht, The Netherlands, 1993, pp. 225-249.
- [20] Rossak, W., Zemel, T., "Integrative Domain Analysis via Multiple Perceptions", Informatica, Vol.17, No.2, 1993, pp. 117-136.
- [21] Shaw, M., "Larger Scale Systems Require Higher-Level Abstractions", in Proceedings of the Fifth International Workshop on Software Specification and Design, Pittsburg PA, May 1989, pp. 143-146.
- [22] Simpson, H.R., "Architecture for Computer Based Systems", Proc. of the 1994 Tutorial and Workshop on Systems Engineering of Computer-Based Systems, IEEE Computer Society Press, Los Alamitos, CA, May 1994, pp. 70-82.
- [23] Zemel, T., MegSDF - Mega-System Development Framework, Doctoral Dissertation, Department of Computer and Information Science, NJIT, Newark NJ, USA, 1993.