A Generic Model for Software Architectures

Complex systems require an especially high level of coordination and integration. To address this need, the authors developed GenSIF, a generic framework to help developers understand software and integration issues in domain-specific, large-scale systems development.

WILHELM ROSSAK
Friedrich-Schiller Universitat Jena

VASSILKA KIROVA AND LEON JOLOLIAN
New Jersey Institute of Technology

HAROLD LAWSON
Lawson Förlag och Konsult AB

TAMAR ZEMEL
Rafael

The software research community has expended much effort to identify and specify advanced process models, especially models of the systems development process. This has given us a variety of process models, supporting tools, and related information models. However, complex systems require an especially high level of coordination and integration—one that is related to software design concepts.[1] Only recently have these concepts been recognized under the "domain-specific software architectures" label.

The Engineering of Computer-Based Systems model, developed by the IEEE Technical Committee on ECBS, identifies the major artifacts and their interrelationships in computer-based systems development.[2] Figure 1 shows the constituents of this model: process, architecture, information, and methods and tools.

0740-7459/97/$10.00 © 1997 IEEE

Process and architecture form the base of the ECBS model. Tools and methods, as well as information models, can be successful only if you first specify a clear process and compatible architecture model. Even more important, process and architecture must have a balanced relationship to ensure that the framework does not tip to one side and enforce process standards without integrating well-defined architectural concepts. Once a domain-wide architecture is in place, you can more easily tackle issues like project coordination, interoperability, design reuse, maintainability, extensibility, and long life cycles, and thus gain a better chance of success.[2-4]

The IEEE ECBS TC has developed a specification for the ECBS model's architecture constituent and published its preliminary results. We are building upon the ECBS work by further refining the existing characterization of the ECBS architecture element into a generic framework: the Generic Systems Integration Framework, or GenSIF.[4-6] Here, we discuss the ongoing development of GenSIF and show how it works in an enhanced development process. Although we work within the official ECBS framework and are actively involved in the ECBS architecture working group, the models discussed here are our own and go beyond the official viewpoint of the IEEE ECBS TC.

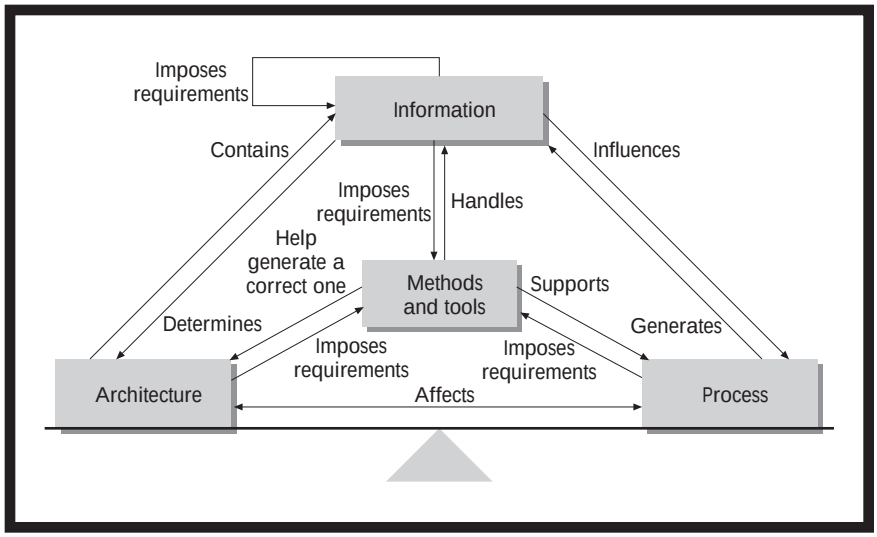

Figure 1. *The ECBS model identifies the major artifacts of computer-based systems development and the way these constituents interact.*

GENSIF—A GENERIC FRAMEWORK

We are developing GenSIF within the context of ECBS problem specification, focusing on its architecture and process components. GenSIF's purpose is to serve as a generic framework to help developers understand software and integration issues in domain-specific, large-scale systems development in a specific business domain.

We define a business domain as a self-contained space of human action in the real world that has well-defined functions and services—such as banking, avionics, and telecommunications. The richness of activities and diversity of operations in this domain have led to the development of many computer-based systems: office systems, command and control systems, information services, image processing, automation, and so on. The combined functionality of these systems constitutes a "system of systems."

The basic architectural principle used in GenSIF is that of a distributed set of independently developed and functioning systems, so-called "building blocks," that interoperate within the context of an overall, coherent mission in the given business domain. Interoperation can be achieved by a variety of means—such as transparent message passing or using a blackboard element—and must be specified in the architecture description.

Our research focuses on delineating new techniques and artifacts that will assure that independently developed or pre-existing building blocks and systems will interoperate in a large and complex system of systems. Software development in GenSIF is thus based on three elements of coordination and domain standardization,[5] as Figure 2 shows.

- The *domain model* is a semiformal domain description that creates a comprehensive knowledge base and influences the outcome of all development and integration initiatives in the domain.
- The *domain architecture* is a unifying, coherent, multilevel software architecture specification used as a guiding plan in any development effort. It limits the design choices at lower design levels and supports interoperability and reuse.
- The *infrastructure* is 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.

Because it is the core of our framework, the domain architecture has a strong effect on the entire development process. The domain architecture promotes technological standardization and acts as a de facto application standard. It usually supports the philosophy of an open and distributed system of systems while safeguarding integrity and semantic consistency. Such an architecture serves as a frame of reference and conformity and thus guarantees that the target system and its con-

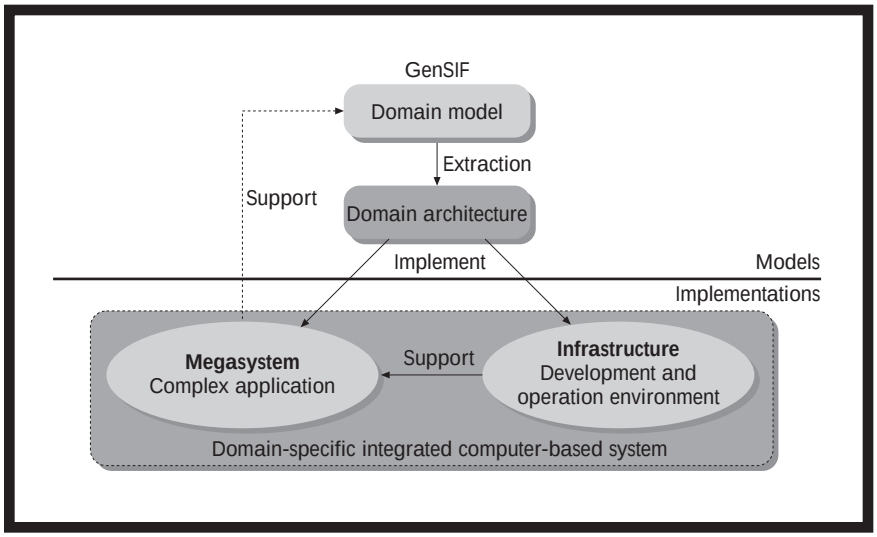

Figure 2. *The GenSIF framework is based on three elements of coordination and domain standardization: the domain model, domain architecture, and infrastructure.*

stituents will meet their functional and nonfunctional requirements from a domain-wide perspective.

To achieve these goals, we base the architecture on an analysis of a comprehensive domain model; the architecture, in turn, imposes a standardized set of tools and common services that constitute the infrastructure. The domain model and infrastructure are, respectively, prerequisite and consequence of the architecture. They cater to the needs of the architecture and are, in a typical feedback process, driven by the architecture's structure and concepts.[4] The domain architecture in GenSIF is thus a domain-specific design model targeted at systems integration. It is specified at two levels—as a conceptual architecture and the derived application architectures.

Model specification. The *conceptual architecture* is a multilayered, generic design model, constrained by the domain specifications. It is specified using a generic architecture reference model, which we discuss below. 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 used as a design template and instantiated into an *application architecture*.

In a system of systems, the conceptual architecture prescribes the communication and control you should use, the types of components that will work across the domain, and so on. Application machines[7] and Bellcore's OSCA[8] are examples of this type of conceptual architecture.

Our analysis of literature on the OSCA architecture showed that some of our concepts are already used in daily practice. The OSCA architecture—one of the few published cases of an industrially applied conceptual architecture model— is targeted at the telecommunications and information systems domains. OSCA identifies three types of building blocks: data-layer (DLBB), user-layer (ULBB), and processing-layer (PLBB). It communicates by activating a contract (an enriched interface), which invokes a building block service.

DLBBs manage the semantic integrity of corporate data—data that is of importance to the whole company. 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 also harbor private data, not accessible to other building blocks.) ULBBs provide interaction with the system's environment, especially humans. They enable users to access functions provided by the system of systems. The processing layer, formed by the PLBBs, implements all functionalities that do not belong to either the data layer or the user layer. This typically includes functions to support business management and operation.

Concepts in practice. Once a generic conceptual architecture has been specified, you can use it to develop numerous application systems. You do this by instantiating the generic concepts, rules, and principles defined as application-specific structures and interaction patterns within the conceptual architecture.

For example, you might develop a system of interoperating building blocks that provide telephone directory services, connection establishment, and billing in New Jersey and Delaware. Each of the externally visible functions would use a set of building blocks, such as a DLBB to steward the telephone directory data for Morris County and a ULBB to make that information accessible via touch-tone dialing.

All projects develop their solutions as a system of systems—or to be precise, a system of building blocks—and use the same conceptual architecture to derive their application architectures. The domain is thus served by a steadily growing, open network of interoperating building blocks instead of by isolated, monolithic systems with a monopoly on certain chunks of data or functionality.

The principle is simple: When building blocks conform to the conceptual architecture, they are also guaranteed to fit into a system of systems within the business domain that offers more than the sum of its parts.

PROCESS MODEL OVERVIEW

GenSIF is oriented toward 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 GenSIF development process strives to give project teams as much freedom as possible while also providing 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—and thus guarantees building block interoperability.

Our focus is on planned-for integration, which prepares systems for integration without assuming to know all potential megasystem parts or domain requirements. This is opposed to approaches such as post-facto integration, which strive to resolve problems of integrating uncoordinated systems; or complete pre-facto integration approaches, which assume you can plan everything in advance and thus rely on complete requirements and fully detailed system design. (We used variations of post-facto integration in part of GenSIF to resolve the problem of legacy systems in the application domain.)

In trying to keep projects as independent as possible while at the same time coordinating and integrating them, we decided upon a two-tiered development approach: the domain level, which considers the application domain, and the subordinate project level, which focuses on a limited space within the domain. What requirements are to a project, the domain model is to the domain level.

Traditionally, single systems are implemented according to a locally optimized design, with tool support centered around the local environment. With GenSIF, the building blocks derive their design from the domain architecture, which provides a standardized infrastructure as a common tool platform.

Figure 3 shows the process model in an extended flow diagram, relating tasks, models, and results.

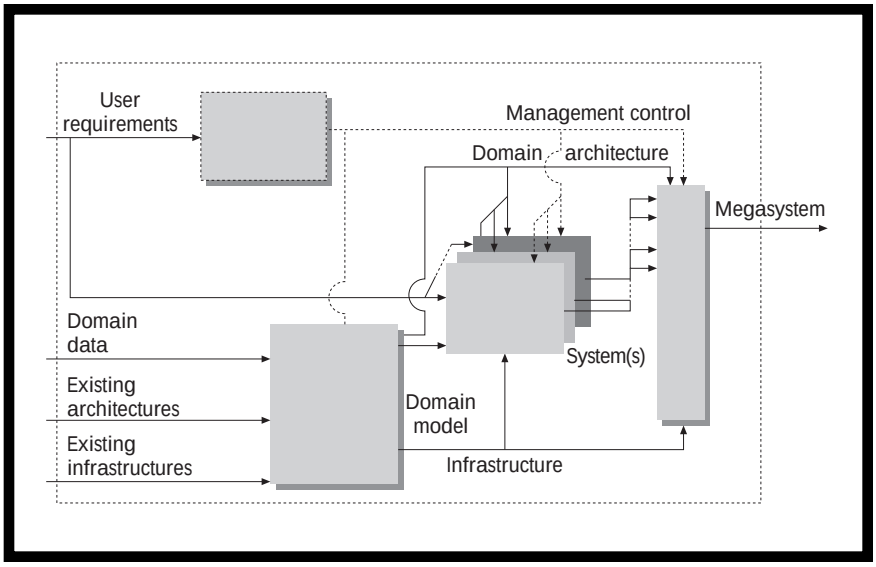

Figure 3. *The architecture-driven development process used in GenSIF.*

Projects represent the traditional system development process, transforming requirements into user-specific systems. 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 time, can overlap in their timing, or can be related in a sequential manner. To support these projects and to guarantee an integrated result (a system of systems and not a set of unrelated systems), we use the domain model, domain architecture, and infrastructure, and control all projects with a metamanagement task activity.

The domain model lets the project team handle requirements and communicate domain knowledge using a standardized model. The domain architecture provides a design reference model to guide projects during their internal design activities. The technical infrastructure supports project implementation by providing a standardized set of tools and technologies that all projects share. Domain architecture, domain model, and infrastructure are specified and maintained by the megasystem task, which complements the individual projects through domain-level coordination.

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, and so on.

A GENERIC ARCHITECTURE REFERENCE MODEL

Beyond its usefulness during systems development, an architecture reference model can help us analyze and understand software architectures. It does this 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.[4]

An architecture model must address

- the relations that bind a system architecture to the corresponding development process,
- the relations to the information model,

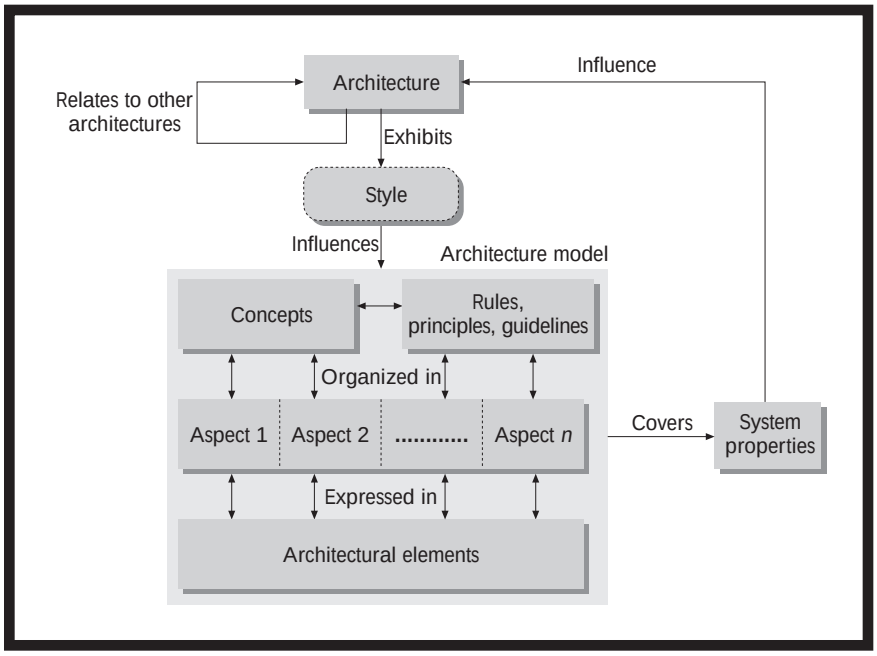

Figure 4. *GenSIF's generic architecture reference model has two parts: the first captures design rationale; the second introduces the architectural elements.*

♦ the relations to supporting tools,

♦ the corresponding body of applicable engineering knowledge and design rationale (concepts and rules), and

♦ the set of constructive elements (and notations).

Because it is the basis for a domain-specific conceptual and application architecture, the generic architecture reference model must be compatible with and integrated into the corresponding development process.

Information models are useful because they make the knowledge and structure of both the process and the architectural model explicit. However, because the architecture model is typically underspecified in most existing frameworks, more work on architecture models will be necessary before clear relationships to the information model can be established. D.W. Oliver provides an example of an information model in the ECBS context.[9]

Another type of information that is vital to architecture specification is domain knowledge, as captured in GenSIF's domain model.[6] In GenSIF, the domain model supports the architecture specification by providing domain characteristics that can be used as parameters in identifying a compatible domain architecture.[4,5]

In our model-driven approach, tools are the logical consequence of the domain architecture; in other approaches, existing tools often impose a de facto architecture without appropriate architectural design or documentation. To support the tool aspect, every architecture—and thus every complete architecture reference model—must provide an interface that imposes clear requirements on tool developers. It might, for example, list typical architectural elements that must be supported and specify notations to capture them.

Beyond its relations to other components of a larger framework (like the one provided by ECBS), an architecture reference model must capture the goals, design rationale, and major design decisions that guide software development in a specific business domain. It can also communicate engineering knowledge by providing a framework to organize and describe "good practice." An architecture model should also generically specify the required basic properties, as well as the system organization and behavior in terms of components, interactions (connectors), and static and dynamic configurations.

For GenSIF, we use a generic software architecture reference model, which is a refinement of the current ECBS architecture model. The most important influences on the GenSIF model have been the OSCA architecture[8] and the application machine concept.[7] Both architectures are widely used in practice and have been tested in case studies in our labs. Thus, GenSIF's architecture reference model is to a certain extent a reverse-engineered model, based on a set of existing architectures and studies. The GenSIF architecture reference model has two parts, as Figure 4 shows. The first part captures the design rationale and is a collection of concepts, rules, principles, and guidelines. The second part introduces the constructive portion of the model, the architectural elements.

The ANSA project has established the notion of concepts, rules, principles, and guidelines.[10] In GenSIF,

♦ *concepts* include the required system properties and the elements available to build the system that the architecture must support;

♦ *rules* limit the set of possible interactions and structures and impose constraints on how elements may be combined to form systems;

♦ *principles* list and evaluate useful structures and patterns that help you meet particular needs while staying within the boundaries of concepts and rules;

♦ *guidelines* provide general hints and techniques to help you stay out of trouble when using concepts, rules, and design principles; and

♦ *concepts* are decisions about the many different aspects of an architecture, which you can structure by separating properties and constructive elements.

Conceptual elements. Properties are goal

statements derived from the domain characteristics and should be reflected in all systems developed in the domain. The conceptual architecture, driving all design efforts in software development projects, is responsible for upholding and implementing these properties. Typical properties include security, performance, reliability, robustness, user friendliness, and so on.

Generally, properties are the nonfunctional, quality aspects of systems. However, if the constructive property is both a supporting quality element and a basic concept in the architecture's characteristics, it can state a more constructive goal. For an open system of systems, for example, "location-transparent message passing" is a basic concept.

When you specify rules for the architecture, you can further refine the properties. The architecture can thus implicitly enforce properties by limiting use of constructive elements to a set of approved structures. "Security of invocation" in an open, distributed system is a typical example of a property that might be mapped into rules.

Another way to realize properties is to implement them directly in the supporting infrastructure. In this case, you must document the model's required properties to avoid losing an architectural concept by implementing it on the code level without any higher-level documentation.

Constructive elements. These elements cover the classical principle of system design by identifying general design elements. We base the classification system of the GenSIF architecture reference model on the ECBS architecture working group's agreed-upon elements: components, connectors, and types of abstraction.

♦ *Components* are the elements that form the designer's mental model of a functioning system. They are the sources of activity and organize a system's architecture. Components can come in a variety of types and specializations, such as building blocks, objects, data capsules, filters, subroutines, and tasks. They can be of a very general nature, such as the building blocks in the OSCA architecture, or highly specialized, as in application machines. Even when based on a simple concept, architecture components are often subdivided into different types with the same layout but with different capabilities and specialities.

♦ *Connectors* facilitate interaction between components and form executable structures. As with components, the types and characteristics of connectors can vary widely and include solutions like messages, procedure calls, remote procedure calls, buffers, event broadcasts, external interfaces, and contracts. In GenSIF, we use a basic classification of connectors divided into data-carrying connectors, control-oriented connectors, and hybrid forms that exhibit both characteristics. Messages, for example, mainly carry information, but also encompass some control aspects, triggering services in the receiving building block.

♦ *Abstraction types* let you handle more complicated architecture models. Typically, there are two relationships that describe the abstraction aspect: aggregation/decomposition and generalization/specialization. These two types of abstraction let you construct hierarchies based on basic "Part-Of" and "Is-A" relationships. To be useful in an architecture model, you must specify precisely the characteristics of the relationship.

In the conceptual architectures we studied, the use of hierarchies was less extensive than it is in basic software system design. The reason for this might be that some architectures are specifically designed to be very simple and flat. They also typically exhibit few component types and connectors. However, in architectures like Doris,[11] hierarchically organized levels of abstraction and the transformation between levels are a central issue.

Rules, principles, and guidelines. Not all aspects of our reference model are appropriate for every architecture. However, if you establish a grid to specify the possible alternatives and typical solutions for concepts, it can function as a checklist for architecture completeness and support the development of rules, principles, and guidelines that complement the given concepts.

♦ *Rules* simply specify constraints on concepts, limiting a designer's choices when constructing a system while still allowing reuse of high-level design concepts and facilitating tool support via a standardized infrastructure. Rules can be targeted toward the constructive, structural aspect of concepts, addressing possible combinations of components and connectors or how to layer structures based on certain types of relationships. A typical structural rule, for example, specifies the type of connector that can be used to achieve communication between different components. Another rule might refine the classification hierarchy concept with a single-inheritance restriction. Rules can also focus on an architecture's properties, translating abstract properties directly into attributes of components and connectors.

Concepts and rules are the backbone of an architecture. You must have concepts and rules if you want to describe a complete architecture model, and they must be obeyed by the system designer.

When you specify rules for the architecture, you can further refine the properties.

♦ *Principles* are of a very different nature; they are compiled design experience that express how to be effective while remaining within the rules. Principles describe what is considered to be "good practice," a notion used in many other engineering disciplines. It is not mandatory to follow principles, but their (re)use reduces design effort and guarantees practical solutions of good quality. In

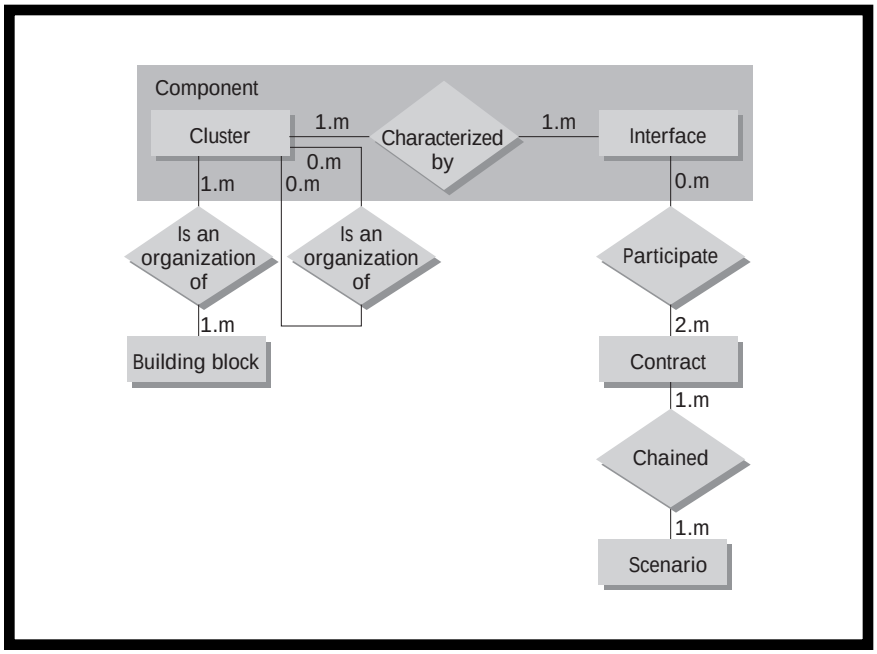

Figure 5. *The GenSIF architectural elements serve as a basis for the architecture specification.*

software development, principles correspond to software design patterns. They offer solutions to application or situation-specific problems in a generic, adaptable way.

♦ *Guidelines* try to anticipate design problems you might encounter within the given set of concepts and rules. In many cases, guidelines are subsumed in the architecture principles. Whereas rules tell you what you should do, guidelines are more concerned with helping you avoid known pitfalls. Thus, they often read like the problem-reason-solution tables presented in manuals. Another way to look at guidelines is to regard them as the accumulated knowledge and "folklore" that build up during the 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.

Architectural elements. Concepts and rules give you 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, they exhibit a clear focus on system semantics and nonconstructive, rule-based characteristics.

Two major groups that have a vested interest in the conceptual model: project designers and tool manufacturers who want to supply the architecture's development and execution environment.

Both groups use the conceptual architecture as a generic template that lets them instantiate their designs. In both cases, the focus shifts from rule-based thinking, which puts semantics first, to a more constructive thinking that connects predefined elements into an application-specific solution. This also includes a shift to a more syntax- and notation-oriented view.

Just as typical programmers organize their knowledge about a specific language around the syntax diagram and look at semantics specifications only if there is a problem, typical system designers will organize their 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, we suggest providing a representation of the architecture that hides rules and calculus behind an easy-to-use, point-and-click, icon-driven interface. The user can then choose whether to use the rule-driven or the icon-driven view.

It can be argued that notations should be domain-specific and application-oriented, supporting the application engineer rather than the architecture engineer.[4] If we accept this, we must allow multiple notations for one conceptual architecture. Concepts and rules represent the architecture's essence, while a constructive, icon-driven view caters to the field engineer. To support the development of domain- and customer-specific architecture notations we can provide an interface that makes it easier to achieve the transition from concepts and rules to a notation with construction-oriented characteristics. As Figure 5 shows, GenSIF provides a set of architectural elements for this purpose.

Ontological elements. Architectural elements repeat most of the aspects captured in concepts and rules as generic elements. These elements form an ontology, listing typical elements likely to appear in a syntax and construction-oriented representation of a conceptual architecture. This ontology can help you unify vocabulary, compare alternatives, and check for completeness in notational solutions and so on, very much like the architecture reference model supports these issues for architectural models. It can also help you decide on a primitive set of icons and graphics that you can later refine.

We selected generic architectural elements to cover the architecture reference model as completely as possible.

They also had to be capable of modeling a software system's static and dynamic structure. Although there are other approaches, we believe the following elements are a useable and pragmatic choice for mapping concepts, rules, and guidelines into a user-friendly interface.

♦ A *building block* is an autonomous structural primitive of the architecture, representing processing and data management. Building blocks and components relate directly to the layering aspect of the reference model. If there is no decomposition hierarchy or other form of clustering, the building block is the only element you need. In a flat system of systems, component building blocks map directly into architectural building blocks. Types of building blocks can be mapped in the same way.

Be aware, however, that an architectural building block adds to the specification given in the conceptual architecture. It can, for example, link the rule that "all incoming messages must be checked for authorization" directly to the component concept. It can also provide a notational primitive for "building block" in an icon that symbolizes all information about component type, associated properties, and linked rules. (Such an icon can be supported by the infrastructure in a code skeleton or template.)

♦ A *component* is a cluster of building blocks and/or components with defined external behavior. The merging of information from the reference model can be documented and specified in the component interface. In many tools, building blocks, clusters, and components will become graphic icons, while the interface becomes the typical component specification (including ways to express constraints and properties!).

Component *clusters* let you map abstraction types (Part-Of, Is-A). On the level of architectural elements, this mapping lets you specify notational primitives that manage layering in system design (within the application architecture). Sometimes clustering is implementation-specific, such as when you aggregate a set of objects running on the same platform into an execution cluster. The *interface* is a component abstraction that represents a component's expected external behavior.

♦ A *contract* is a set of requirements and constraints on one or more components prescribing a collective behavior and derived from the participating interfaces. 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. That is—following up on our example of structural rules—specialized icons might express that a user-layer building block can connect to a processing-layer building block, but not to a data-layer building block. You can do this by specifying graphical icons for contracts that "fit" only if all rules are obeyed.

♦ A *scenario* is a dynamic configuration of architecture components. Scenarios let you chain many components and contracts together to form complete execution structures that model dynamic system aspects. You can also use scenarios to store complete design patterns or to show how a thread of execution runs through a complex system, making use of and obeying concepts and rules.

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. In many respects, architectural elements are "incomplete." They are not a notation, but they make it easier to reason about a good notation. Architectural elements act like a set of predefined classes that you can further specialize and finally instantiate into a solution. They also let you specify basic generic functions early in the process. You can, for example, identify the commands needed by an architecture editor to manipulate architectural elements.

Architectural elements do not cover some architectural properties. Although 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. Concepts and rules always cover all aspects of an elements-based representation. However, this representation may not be a complete mapping of the concepts and rules (even though it is easier to communicate to application engineers and designers).

In other industries, the idea of building a corporate culture by establishing a common level of "best practice" is widely known and used. The architecture concept directly supports this goal for our industry and can help us improve problem areas dominated by organizational and social issues, such as health care organizations, educational systems, and so on.[2,3]

Our proposed reference model for architecture specification and development is organized around a set of aspects that structure concepts and rules; these, in turn, specify a conceptual architecture. We have added principles and guidelines to the 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. Adding architectural elements is a step toward a more constructive type of architecture representation.

The idea of building a corporate culture by establishing "best practice" is widely known and used.

Our current research is focused on further refining these concepts and developing a formal specification of the ar-

chitecture reference model. We are continuing to test our ideas in case studies, such as applying our model to the OSCA architecture and the application machine concept. We are also developing a prototype architecture editor, and we are testing different tools to learn more about integrating them into a real infrastructure and to learn what typical services an infrastructure must provide. ◆

REFERENCES

1. M. Shaw, "Larger Scale Systems Require Higher-Level Abstractions," *Proc. Fifth Int'l Workshop Software Specification and Design*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1989, pp. 143-146.
2. H.W. Lawson, "Introducing the Engineering of Computer-Based Systems," *Proc. 1994 Tutorial/Workshop Systems Eng. Computer-Based Systems*, IEEE Computer Soc. Press, Los Alamitos, Calif., May 1994, pp. 2-8.
3. H.W. Lawson, "Philosophies for Engineering Computer-Based Systems," *Computer*, Dec. 1990, pp. 52-63.
4. W. Rossak and P.A. Ng, "Some Thoughts on Systems Integration—a Conceptual Framework," *Int'l J. Systems Integration*, Vol.1, No.1, Kluwer Academic, Dordrecht, The Netherlands, 1991, pp. 97-114.
5. W. Rossak, T. Zemel, and H. Lawson, "A Meta-Process Model for the Planned Development of Integrated Systems," *Int'l J. Systems Integration*, Vol.3, No.3, Kluwer Academic, Dordrecht, The Netherlands, 1993, pp. 225-249.
6. W. Rossak and T. Zemel, "Integrative Domain Analysis via Multiple Perceptions," *Informatica*, Vol. 17, No. 2, 1993, pp. 117-136.
7. H.W. Lawson, "Application Machines: An Approach to Realizing Understandable Systems," *Euromicro J.*, Sept. 1992, pp. 5-10.
8. *The Bellcore OSCA Architecture*, Bellcore/Bell Communications Research, Tech. Advisory TA-STS-000915, No. 3, Mar. 1992.
9. D.W. Oliver, "A Draft Integration of Information Models: Component Model and Oliver Model," *Proc. 1994 Tutorial/Workshop Systems Eng. Computer-Based Systems*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1994, pp. 44-96.
10. *ANSA Reference Manual*, Vol. A, Architecture Projects Mgt. Ltd., London, UK, 1989.
11. H.R. Simpson, "Architecture for Computer Based Systems," *Proc. 1994 Tutorial/Workshop Systems Eng. Computer-Based Systems*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1994, pp. 70-82.

Wilhelm Rossak is a professor of software technology at the Friedrich-Schiller Universitaet Jena, Germany. His research interests are in software and systems engineering, software architectures, process models, and information and knowledge management. From 1990 to May, 1997, he was an assistant professor of software engineering at the New Jersey Institute of Technology. From 1982 to 1984 he was lead programmer and systems analyst at a leading software house in Vienna, mostly involved in the development of large online information systems.

Rossak received his Dipl.Ing in 1983 and PhD in 1989, both in informatics, from the Technical University of Vienna, and his Habilitation from the University of Klagenfurt, Austria, in 1994. Rossack is a member of the IEEE Computer Society, the Austrian Computer Society, and ACM.

Vassilka Kirova is a member of the technical staff at Lucent Technology Bell Labs Innovations and a doctoral candidate in the Department of Computer and Information Science, New Jersey Institute of Technology. Her research interests include software architecture, architecture description languages, domain engineering, process models, and system integration.

Kirova received an MS in computer science and engineering from the Technical University of St. Petersburg, Russia. She is a member of IEEE Computer Society.

Tamar Zemel is head of the software department in the missile division at Rafael in Haifa, Israel. Her interests include software process improvement, domain analysis, and software architectures.

Zemel received an MS in computer science from Technion, Israel, and a PhD in computer science from the New Jersey Institute of Technology.

Leon Jololian is the director of computing and a PhD candidate in the Computer and Information Science Department at the New Jersey Institute of Technology. His research interests include software architecture and electronic enterprise engineering.

Jololian received a BE in electrical engineering from Manhattan College, an MS in electrical engineering from Georgia Institute of Technology, and an MS in computer science from Polytechnic University in Brooklyn, New York.

Harold W. Lawson is the managing director of Lawson Konsult AB in suburban Stockholm. He has been active in academic and industrial computing since 1958. His experience includes work in systems and software engineering, computer architecture, programming languages and compilers, operating systems, and various application domains. He has contributed to several pioneering efforts in hardware and software technologies at Univac, IBM, Standard Computer Corporation, and Datasaab, and held professorial appointments at several universities including Polytechnic Institute of Brooklyn; University of California, Irvine; Universidad Politecnica de Barcelona; Royal Technical University; University of Malaya; and Keio University. He has written and contributed to several books and numerous articles.

Lawson received a BS from Temple University in Philadelphia and a PhD from the Royal Technical University, Stockholm. He is a fellow of the IEEE and ACM, and a founding member of SigMicro, EuroMicro, and the IEEE Computer Society Technical Committee on the Engineering of Computer-Based Systems.

Address questions about this article to Rossak at Friedrich-Schiller Universitat Jena, Institut F. Informatik, Ernst-Abbe Platz 1-4, 07743 Jena, Germany; rossak@informatik.uni-jena.de.