Model-driven systems development

L. Balmelli
D. Brown
M. Cantor
M. Mott

Traditional systems development methods are designed to create a "point solution"–that is, a solution for a specific and static set of requirements. These methods result in systems that are sluggish in their response to dynamic conditions and changing requirements, expensive to maintain over extended periods of time, and prone to system failure. As an alternative to this approach, this paper describes a model-driven approach to systems development, which extends traditional systems engineering methods and is well-suited to a systems development environment characterized by rapidly changing conditions and requirements. We describe how model-driven systems development is performed by using the RUP® SE (Rational Unified Process® for Systems Engineering) architecture framework and transformation methods, which have been refined over eight years of field experience by Rational®, IBM, and our clients.

INTRODUCTION

According to conventional wisdom, software and systems projects are seldom successful because success is defined as the delivery of systems that perform as promised, on time, and within budget. According to a report published in *Crosstalk, the Journal of Defense Software Engineering* in 2003, $84 billion was spent on projects that were never finished, and $192 billion was spent on projects that were significantly late and over budget.[1]

There can be many explanations for this state of affairs, including the need for a redefinition of success in this context. This redefinition would encompass the delivery of systems that meet a variety of stakeholder needs, including content, cost, and schedule. It would allow for the likelihood of improving understanding of stakeholder needs as a program proceeds. Nevertheless, there is some consensus in a variety of contexts that fundamental change is needed in the way systems development is conducted in order to create systems that are agile in response to dynamic conditions and requirements. As stated by Alberts, Garstka, and Stein, "A fundamental lesson that has emerged from multiple domains, including business operations, product development, and defense, is that the power of a new technology cannot be fully exploited to create competitive advantage without the simultaneous co-evolution of organization and process."[2] Among the

©Copyright 2006 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the Journal reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to republish any other portion of the paper must be obtained from the Editor. 0018-8670/06/$5.00 © 2006 IBM

characteristics of the new systems required by today's environment are tighter collaboration among the engineering disciplines involved in systems development and operational collaboration among members of extended enterprises.

In business, supply chain management creates value; the more tightly supply chains are integrated, the more value they create. Hence, the participants in a supply chain strive to integrate their operations and data and, at the same time, maintain unique business processes that create competitive advantage. In warfare, providing the right information at the right time to the right participant supplies a critical advantage. In defense and industry, this creates a need for components that are loosely coupled for architectural benefits such as maintainability and extensibility, yet tightly integrated to interact efficiently. In addition, as the components become more tightly integrated operationally, functionality is naturally redistributed as the collaborations are understood, which naturally results in higher capabilities and fewer components.

The plethora of poor development outcomes is not so much the result of incompetent execution of traditional processes, but rather the result of the inadequacy of the processes themselves to deliver modern collaborative systems. Traditional systems development methods, based on a static and predictable set of requirements, are limited in their capability to address the challenges of operational integration, effective information sharing, and the increasing volatility of requirements. These methods were designed to create a "point solution," and an essential assumption of these methods is that requirements must be well understood before development can go forward. As requirements volatility increases, such methods become less viable. Further, experience has shown that traditional requirements-driven methodologies result in systems that are limited in their capability to self-modify in response to evolving mission or business needs, brittle and difficult to manage in adapting to new requirements, and expensive to maintain over an entire product life cycle.

Systems engineering methods have to be extended in order to develop collaborative and agile systems throughout a broad set of application domains, including business-structure analysis, value-chain collaboration, and warfare. This paper describes a model-driven approach to systems development that extends traditional systems engineering methods and is well-suited to address the additional systems development concerns we have described.

Systems

A *system* is a set of resources that is organized to provide services. The services enable the system to fulfill its role in collaboration with other systems to meet some useful purpose. Systems may consist of combinations of hardware, software (including firmware), workers, and data. This definition of systems is extremely general: a product, such as an automobile or a computer, is a system; a business or its components are also systems. Businesses may be organized into larger enterprises that are also systems, e.g., the health-care system.

Systems can be characterized by their attributes, which can be grouped as follows: "Black box" attributes are a system's externally observable characteristics, including the services that it provides. "White box" attributes are the resources that comprise the system. A white-box view is a representation that reveals the internal aspects of the subject under consideration. The system's white-box resources are encapsulated in its black-box services.

Systems development may be thought of as the specification of the black-box attributes of the system (i.e., its requirements) and the delivery of the integrated system components that meet the requirements (i.e., its implementation or realization). An *implementation* describes how an item is constructed or computed. For example, a class is an implementation of a type; a method is an implementation of an operation.

At a high level, a *service* is a mechanism by which the needs or wants of the requestor are satisfied. In a given context, the term "service" represents either a service specification or a service implementation, or both. A *service specification* is the definition of a set of capabilities that fulfill a defined purpose. In model-driven systems development (MDSD), a service specification can be a Systems Modeling Language (SysML**)[3] interface. SysML is an adopted extension to the OMG** UML** (Unified Modeling Language**) standard that supports systems engineering. It is to hardware products and systems what UML is to software architecture and systems. To represent today's product embedding both hardware and software, both SysML and UML

are used in a common setting. A *service implementation* realizes the behavior described in the service specification and fulfills the service contract. In MDSD, the service implementation is represented by the logical and physical projections (known as "viewpoints") of the model.

A *model* is defined as a collection of all the artifacts that describe the system. Generally, model-driven development (MDD) is a technique for addressing complex development challenges by dealing with complexity through abstraction. Using this technique, complex systems are modeled at different levels of specificity. As the development program proceeds, the model undergoes a series of transformations, with each transformation adding levels of specificity and detail. For the development of complex systems, MDSD begins with the black-box specification of the system and, through a rigorous process of transformation, creates a model of the system; this model is ultimately realized with tested and functioning system components.

The classical notion of a black-box and white-box characterization of a system is well-suited to an MDSD approach. The process of starting with an external view of a system's behavior and deriving specifications for the components and their integration in a semantically rigorous framework is achieved by model transformation. As we will describe next, this overall transformation from external view to component specification and integration is decomposed into several sub-transformations that account for multiple sets of engineering concerns.

Systems development approaches

Traditional requirements-driven systems development methods were developed before the Internet era. The purpose of life-cycle reviews in the traditional development environment was to synchronize a program's cost, schedule, and technical baselines in order to review the program in its entirety. Such reviews necessarily relied upon paper documents because of the inability of early information systems to provide electronic reviews of such programs. Hence a practice of paper-oriented life-cycle reviews was built around available technology, and this practice continues to this day.

Technology has changed. Object-oriented software development has led to the development of systems models to characterize complex behaviors. Further, current database and Web technology may be applied to enable new development methods. Effective exploitation of new technologies requires adoption of new processes. In this spirit, this paper describes an MDSD method that replaces traditional requirements-driven systems development methods.

Requirements-driven systems development methods

Requirements-driven systems development methods define requirements early in the life cycle, after which the techniques of functional decomposition are applied to determine the mapping of requirements to system components. At every level of the hierarchy, functional analysis derives requirements, and engineering methods derive measures of effectiveness. Once the requirements are described in sufficient detail, detailed design activities begin. This process is defined in Reference 4 and elsewhere.

As systems become more complex and integrated, with fewer components delivering more capability, this traditional approach becomes unwieldy due to the large number of possible mappings. It is common for a modern system, such as an automobile, to have thousands of detailed requirements and thousands of components, resulting in millions of possible mappings. Faced with this dilemma, developers limit the level of integration, resulting in systems that may be highly capable but are brittle and difficult to maintain.[5] MDSD methods mitigate this explosion of mappings by providing levels of abstraction.

The development team must understand the mission and concerns of the business and how the proposed system relates to them. To ensure that the system indeed satisfies its intended purpose, the development team must establish that the top-level system requirements and the requirements derived from them are satisfied by the collaboration of the system's components. MDSD must address these system concerns, and, at the same time, provide the development team with an improved understanding of the system, its goals, and its components.

A model-driven systems development approach

MDSD must build upon the techniques of requirements-driven development methods in light of their historic success, but, for reasons described previously, a change in the approach to systems development is required. MDSD starts with *system decomposition*, that is, the division of a system into

elements in order to improve comprehension of the system and the way in which it meets the needs of the user. Because of the limited capability of humans to understand complexity, a "divide and conquer" system decomposition approach is appropriate.[4] In this approach, the system is decomposed into a comprehensible set of elements, each of which has a comprehensible set of requirements. Sometimes, to manage complexity in very large systems, system decomposition must be applied recursively.

Effective application of system decomposition requires the means of modeling the system from a variety of viewpoints and at increasing levels of specificity. In addition, a set of transformations between model levels is required as a basis of the development process. These transformations provide a means of deriving the next level of specificity while maintaining traceability and coherence for the entire model. MDSD consists of creating the model artifacts as a means of specifying the system elements and their integration. An *artifact* is defined as any item that describes the architecture, including a diagram, matrix, text document, or the like. This model provides a common means for facilitating collaboration across the engineering disciplines, coordinating iterative development methods, and assigning technical and managerial responsibilities.[6]

In this paper, we describe an MDSD modeling framework, the Rational Unified Process* for Systems Engineering (RUP* SE), and some of its fundamental transformation methods. This MDSD framework and its associated transformation methods have been refined over eight years of field experience by Rational*, IBM, and our clients.[7–9]

A comprehensive treatment of MDSD is beyond the scope of this paper. We provide a brief introduction to this field and supply references to more extensive treatments of the topics touched upon. A discussion of the underlying framework for modeling systems is presented in the section "The RUP SE architecture framework." This framework provides the setting for the transformations. The section "Modeling Languages and MDSD" is a discussion of the use of SysML to capture some of the key framework artifacts. The generation of the artifacts requires some specific transformation methods, described in the section "Transformation methods."

Finally, we conclude with a precise description of the framework captured in a RUP SE metamodel. This metamodel is implemented as a profile for SysML (and for UML when RUP SE concepts apply to the software part of the system). The metamodel therefore adds specific semantics to enable these languages to handle our framework. Thus, the various models shown in this paper are represented using RUP SE semantics that extend SysML, making them, in that sense, RUP SE models. Although this approach is appropriately supported by a modeling language such as SysML (and its toolset), it is actually independent of any language.

THE RUP SE ARCHITECTURE FRAMEWORK

The RUP SE architecture framework provides support for constructing a sound architecture on the basis of four principles: separation of concerns, integration, system decomposition, and scalability. *Separation of concerns* allows designers to address each set of stakeholder concerns independently; *integration* is achieved by requiring the use of a common set of design elements across multiple sets of concerns. *System decomposition* subdivides the system by structure, rather than by function, enabling the framework to provide levels of structure that enable parallel development; and *scalability* is achieved by using the same framework, whether the system under consideration is an enterprise or a product component or anything in between.

In this section we provide an overview of the framework and the reasoning underlying its design. In addition to the discussion here, a detailed description is provided in the section "RUP SE metamodel" at the end of this paper.

Like many frameworks, the RUP SE framework consists of two kinds of artifact: *static artifacts*, namely, representations of the system in its context and the things that comprise the system; and *dynamic artifacts*, namely, how the static elements collaborate to fulfill their role in the system. The static artifacts enable separation of concerns and scalability and provide the semantics for system decomposition. The dynamic artifacts enable integration of concerns. We discuss both types of artifact in the following sections.

Table 1 illustrates the RUP SE architecture static framework. The framework consists of three types

Table 1 The RUP SE architecture framework. The cells of this table show sample model views.

Model Levels	Model Viewpoints					
	Worker	Logical	Information	Distribution	Process	Geometric
Context	Role definition, activity modeling	Use case diagram specification	Enterprise data view	Domain-dependent views		Domain-dependent views
Analysis	Partitioning of system	Product logical decomposition	Product data conceptual schema	Product locality view	Product process view	Layouts
Design	Operator instructions	Software component design	Product data schema	ECM (electronic control media) design	Timing diagrams	MCAD (mechanical computer-assisted design)
Implementation	Hardware and software configuration					

of element: *model levels, viewpoints*, and *views*, each of which is described in detail in the following subsections. In the representation of Table 1, the rows represent model levels (context, analysis, design, and implementation levels), the columns represent viewpoints (worker viewpoint, logical viewpoint, information viewpoint, etc.), and the cells represent views (e.g., the worker viewpoint at the design level).

While Table 1 describes the framework elements at a high level, the metamodel described in the section "The RUP SE metamodel" provides a more complete description of the elements and their relationships.

Table 2 Model levels in the RUP SE architecture framework

Model Level	Expresses
Context	System black box—the system and its actors (though this is a black-box view for the system, it is a white-box view for the enterprise containing the system)
Analysis	System white box—initial system partitioning in each viewpoint that establishes the conceptual approach
Design	Realization of the analysis level in hardware, software, and people
Implementation	Realization of the design model into specific configurations

Model levels

A *model level* is defined as a subset of the architecture model that represents a certain level of specificity (abstract to concrete); lower levels capture more specific technology choices. Model levels are not levels of abstraction; in fact, a model level may contain multiple levels of abstraction.

Model levels are elements designed to group artifacts with a similar level of detail. The MDSD transformations occur between the model levels. *Table 2* describes the four model levels expressed in the framework.

The *context level* treats the entire system as a single entity, a black box. This level addresses the system's interaction with external entities. At the *analysis level*, the system's internal elements are identified and described at a relatively high level. Which elements are described at this level depends upon the viewpoint. For example, in the logical viewpoint, subsystems are created to represent abstract, high-level elements of functionality. Less abstract elements are represented as sub-subsystems, or classes. In the distribution viewpoint, "localities" are created (as described later in the paper) to represent the places where functionality is distributed.

At the *design level*, the decisions that drive the implementation are captured. In the transition from the analysis level to the design level, subsystems, classes, and localities are transformed into hardware, software, and worker designs. This is *not* a

Table 3 The core RUP SE viewpoints

Viewpoint	Expresses	Concern
Worker	Roles and responsibilities of system workers	Worker activities, human/system interaction, human performance specification
Logical	Logical decomposition of the system as a coherent set of SysML blocks that collaborate to provide the desired behavior	• Adequate system functionality to realize use cases • System extensibility and maintainability • Internal reuse • Good cohesion and connectivity
Distribution	Distribution of the physical elements that can host the logical services	Adequate system physical characteristics to host functionality and meet supplementary requirements
Information	Information stored and processed by the system	Sufficient system capacity to store data; sufficient system throughput to provide timely data access
Geometric	Spatial relationship between physical components	Manufacturability, accessibility
Process	Threads of control that carry out computation elements	Sufficient partitioning of processing to support concurrency and reliability needs

direct mapping from system elements to designs; rather, design decisions are made by deriving the design from the functionality represented in the subsystems and classes. These design decisions are constrained by supplementary requirements and distribution choices represented by the localities and their attributes. The resulting design transformation realizes all of the specifications from the analysis level. In other words, the system architecture is specified at the analysis level, creating requirements that the design level must satisfy.

At the *implementation level*, decisions about technology choices for the implementation are captured. Commercial products may be specified (e.g., a messaging middleware product, a model and part number for a piece of hardware), or items might be specified for internal implementation. As before, moving from the design level to the implementation level is a transformation, but this time the mapping is more direct. For example, at the design level, the functional activities of a worker are mapped to a position specification with a defined set of skills. Then, at this level, the specification can be fulfilled either by hiring someone with the correct skill set (similar to choosing a commercial product with certain capabilities) or by training an individual to acquire the required skills (similar to doing an internal implementation). This transformation of worker activities to specific skills, though necessary for the system to meet its goal, is not well understood, and is seldom, if ever, practiced.

Viewpoints

A *viewpoint* is defined as a subset of the architecture model that addresses a certain set of engineering concerns. The same artifact may appear in more than one viewpoint. Viewpoints allow framework users to separately address different engineering concerns while maintaining an integrated, consistent representation of the underlying design. *Table 3* describes the core RUP SE viewpoints.

The set of viewpoints is fluid and has grown over time. Most development efforts do not require all of the viewpoints shown in Table 3. Further, viewpoints are extensible to address program domain-specific needs, such as security or safety. Generally these extended viewpoints can reuse the semantics of the core set of viewpoints.

A particular viewpoint may not be useful at all model levels. For example, hardware developers are a category of (internal) program stakeholders concerned with the allocation of functionality and distribution of hardware within the system. However, at the analysis model level, decisions about where functionality will be implemented (in hardware, software, or workers) have not yet been made. As a result, there is typically no need for a hardware viewpoint at the analysis model level. However, if the system involves actual hardware development, then one certainly does need a hardware viewpoint at the more specific (lower) model levels.

Although different architectures require different sets of viewpoints, almost all require the logical and distribution viewpoints.

The viewpoints chosen in RUP SE have been influenced by the ISO/ITU standard RM-ODP, the Reference Model for Open Distributed Processing.[10] One other important standard that was influential in determining how the viewpoints and views are documented and how they relate to stakeholders and their concerns is the ANSI/IEEE 1471-2000 standard, the IEEE Recommended Practice for Architectural Description of Software-Intensive Systems.[11]

Views

Views constitute the intersection of viewpoints and model levels. Views contain artifacts (i.e., objects used to document engineering data) that describe how the viewpoint's engineering concern is addressed at a particular model level. Table 1 includes the set of view artifacts. In practice, each program chooses the view artifacts that meet its individual needs. The project's set of view artifacts is what the RUP calls the *development case*, which includes the choice of artifacts and prescriptive guidance on how to document them, along with guidelines, templates, and checklists.

The framework may leave the impression that the views contain unrelated artifacts. In reality, there are many relationships between the artifacts. These relationships are captured in the RUP SE metamodel described in the final section of this paper.

MODELING LANGUAGES AND MDSD

SysML can be used to capture some of the key artifacts of the RUP SE framework. As mentioned previously, RUP SE augments this language to support a richer framework by adding new semantics. In this section, we introduce the syntax for expressing some of the key concepts of the RUP SE framework, along with their rationale. Because the RUP SE metamodel is implemented as a profile of SysML, it borrows the concrete syntax of SysML (with the exception of the RUP SE locality, for which it introduces a new concrete syntax, as we describe later).

Key concepts

Two of the most important concepts related to the use of SysML to capture the framework's artifacts are the distribution viewpoint and the idea of context as used here.

Distribution viewpoint

Because systems include all of the resources (hardware, software, workers, etc.) needed to provide their services, systems analysis must include not only the logical decomposition commonly found in MDD frameworks, but also decomposition which determines how the resources will be partitioned. The initial partitioning of the system provides a basis for reasoning about how the logical functionality will be distributed across the physical resources. Thus the distribution viewpoint is a critical element of the RUP SE framework.

The distribution viewpoint describes how the functionality of the system is distributed across physical resources. At the analysis level, we need to describe a generalized view of resources, capturing the attributes needed to support the transformation from analysis to design. Cantor introduced the concept of *locality*[12] to represent a generalized resource. Localities are linked to each other with connections. A locality is defined as a member of a system partition representing a generalized or abstract view of the physical resources. Localities can perform operations and have attributes appropriate for specifying physical designs. Localities are represented in SysML as stereotyped blocks with tagged values that contain characteristics derived from supplementary requirements, assigned technical risk, and cost. A *stereotype* is defined as a variation of an existing SysML modeling element that enables the use of platform or domain-specific terminology or notation. The use of stereotypes is the mechanism for extending SysML in a consistent manner, and this is the mechanism used to implement the RUP SE metamodel.

Connections are defined as generalized physical linkages in RUP SE. Connections are characterized by what they carry or transmit and the necessary performance and quality attributes in order to specify their physical realization at the design level. They are linked to the concept of a "Flow Port" in SysML, which allows the designer to specify what can flow through an association and its ports (data, power, fuel, etc.). A RUP SE distribution diagram showing two localities and a connection between them is shown in ***Figure 1***.

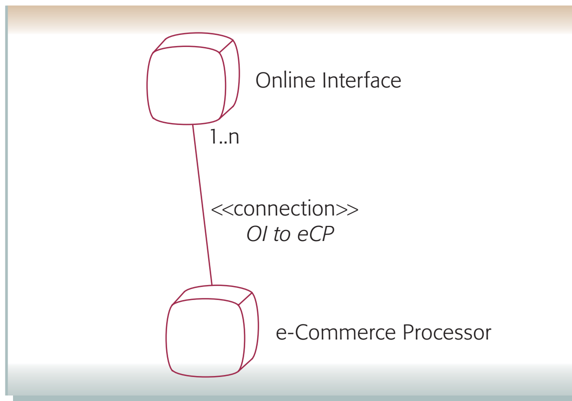


Figure 1
Two localities and a connection

Context

In UML-based software development, context modeling consists of specifying software use cases. Some authors, such as Conallen,[13] added diagram elements to enterprise use case diagrams to specify which enterprise use cases are handled by the software. In systems engineering, the concept of context and context diagramming has a different meaning, which, in fact, is more consistent with the principles of model-driven development. In systems engineering, *context* includes the set of things (people, other systems, etc.) with which the system interacts and how those interactions proceed so that the system can fulfill its role in the enterprise.

In working with the systems community, who typically interact with large teams requiring precise communications, we found that the common informal definition of a use case (namely, a description of a service that the software provides which provides value to the actor) is inadequate for a variety of reasons. A *service* (defined more precisely in the following) is a behavior of a system. The actual semantics of use cases more closely resemble collaboration than behavior. *Value* is far too subjective a term to be included in the definition of a framework element. In any case, the entity receiving benefit from the system behavior may not include the actors in the collaborations. In addition, the software definition of a use case does not provide for scalability. In general, use case decomposition, like all forms of behavior decomposition, does not provide a robust framework for systems architecture. Hence we found it useful to revisit the notion of context in defining the RUP SE framework. In the following subsections, we extend the definitions of "actor" and "use case" from their software definition[14] to meet the needs of systems development. We introduce system context diagrams for systems modeling that go beyond software use case diagrams.

Use cases and actors

In RUP SE, an *actor* is anything that interacts with the system. Examples of actors include users, other systems, and the environment, including time and weather. There is often confusion between 'users' and 'workers.' In systems engineering, users are external to the system, and thus are considered actors. Workers are a part of the system, and thus are not actors. The specification of what is expected of the workers in a system is captured in the worker viewpoint.

In RUP SE, a *use case* is a sequence of events that describes the collaboration between the system and external actors to accomplish the goals of the system. In other words, the use case is a way to specify the behavior required of the system and external entities in response to a given sequence of stimuli.

Context diagrams

(Many of the ideas in this section stem from conversations with Sanford Friedenthal of the Lockheed-Martin Corporation.)

As defined previously, a system is a set of collaborating resources that deliver services. In particular, a system encapsulates the needed resources to deliver those services. To fulfill its goal, a system may rely on the services of others. Hence, context in systems modeling is well-captured by static diagrams capturing the actors with which the system interacts and how the system and the actors are related. Context varies according to viewpoint. In the logical viewpoint, context relates to the static relationships between the system and its actors. The system in this viewpoint is represented as a classifier with attributes and operations, organized into interfaces. An *interface* is defined as a named set of operations that characterize the behavior of an element. An interface declares a set of public features and obligations that constitute a coherent

service offered by a block (in SysML) or a classifier (from UML 2.0). In the distribution viewpoint, context relates to the physical relationships between the system and the actors. In this viewpoint, the system is a kind of locality. In the data viewpoint, context relates to the relationships between the data maintained by the system and the data maintained by its actors.

Figure 2 includes an example of a RUP SE logical context diagram. This diagram is identical to what Lykins et al. call an elaborated context diagram in Reference 15. Following the work of Friedenthal and his colleagues, the diagram includes (in the logical view context diagram) the stereotyped block "I/O entity," which has attributes and no operations. I/O entities capture what passes between the system and the actors. I/O entities may typically contain data, but may also include physical things like power. Figure 2 also includes a context diagram of a distribution view. Although the system is both a logical entity and a distribution entity, it is shown as a class in the distribution view as a matter of style.

TRANSFORMATION METHODS

The RUP SE framework includes novel, related artifacts for transformation methods between model levels. The generation of these artifacts and their relationships requires new techniques. These techniques are described next.

System of systems decomposition

In this subsection, we describe a method of object-oriented logical decomposition to describe a hierarchical system of systems. Additionally, we discuss a number of principles, found in traditional systems development, that underpin the MDSD framework discussed in this paper.

A system encapsulates the resources it requires to deliver its services. Systems may be decomposed into systems, each of which also encapsulates all of their resources. Because systems control their resources and may encapsulate other systems, a "system of systems" is a recursive pattern. A process may therefore be applied to recursively decompose a system into other systems, which are themselves decomposed further. During such recursive decomposition it is important to understand at which "level" in the hierarchy we stand during a discussion. Although terms such as "superordinate system" and "subordinate system" are relevant when discussing the pattern, it is sometimes more useful to discuss "system levels" because more than two levels may be considered.

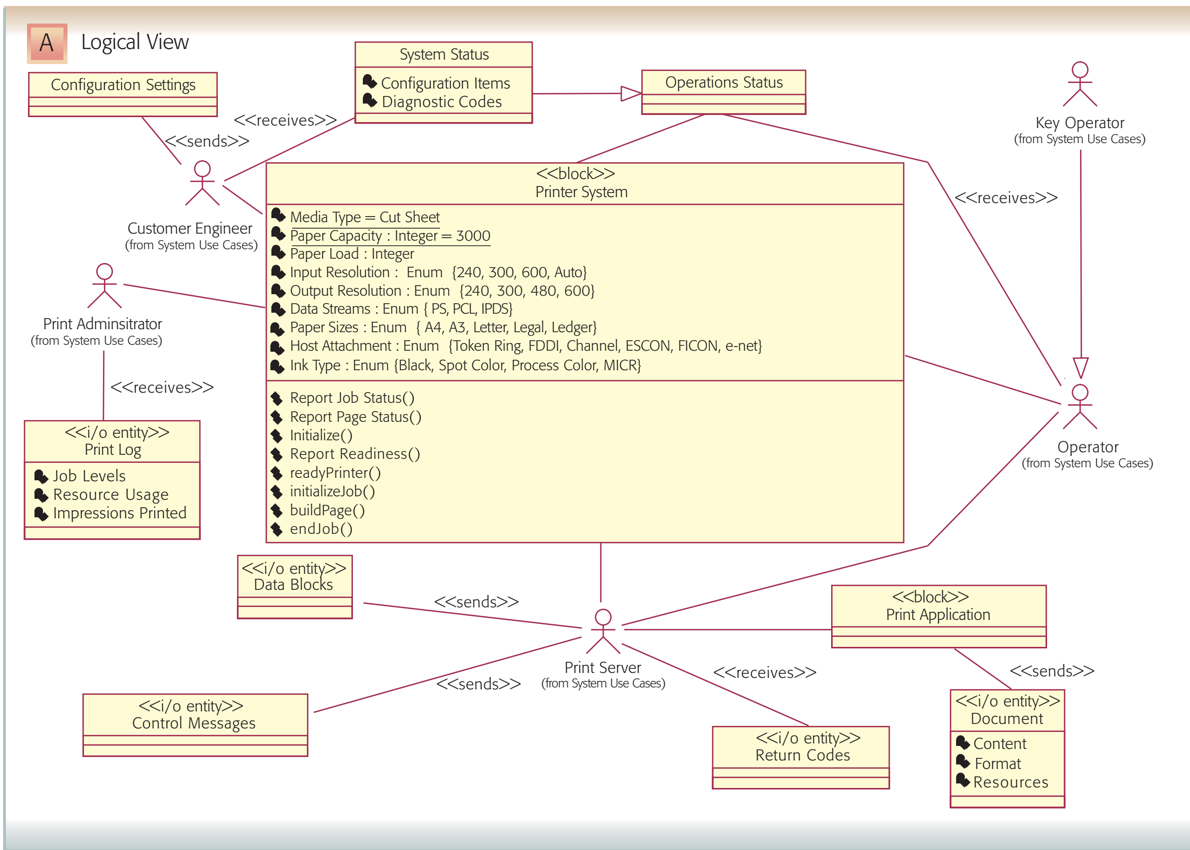
The term "system level" indicates the relative position in the overall hierarchy; "system level 1" represents the "root" system (by definition, there is always exactly one "system level 1" system). An overview of the key artifacts in two system levels is shown in *Figure 3*. This figure shows the pattern that allows the framework to support recursive system decomposition. The dotted lines between the systems indicate UML dependencies.

Operations analysis

Classical use case analysis is a form of requirements decomposition; therefore, for reasons previously described, it is inadequate to meet the needs of systems development. In MDSD, the techniques of use case analysis are extended to *operations analysis*. Operations analysis consists of the following recursive pattern: (1) decompose the system to create a context for the system elements; (2) treat the system operations as use case scenarios for the elements; (3) describe the scenarios in which the elements, as black boxes, interact to realize the system operations; and (4) derive the operations of the elements from the scenarios.

This pattern can be applied starting at the enterprise, which contains the system of interest (hence the context level for the MDSD framework). In this application of the pattern, the enterprise is treated as a system and the system to be developed as a component.

The system decomposition creates the context for the elements; thus, context is maintained at every level of the system hierarchy. The operations analysis provides a method for creating traceability between the use cases, which define the business or mission needs, and the system components that satisfy those needs. The maintenance of this context at each level of the hierarchy was a key insight during our development of the MDSD. The use cases at the top level of the system hierarchy define the interactions of the system with external entities in order to fulfill its mission. These interactions are analyzed to identify the operations that the system provides in order to fulfill its role in the use cases. Operations analysis forms the basis of the use case realization. The operations are combined into interfaces or services.



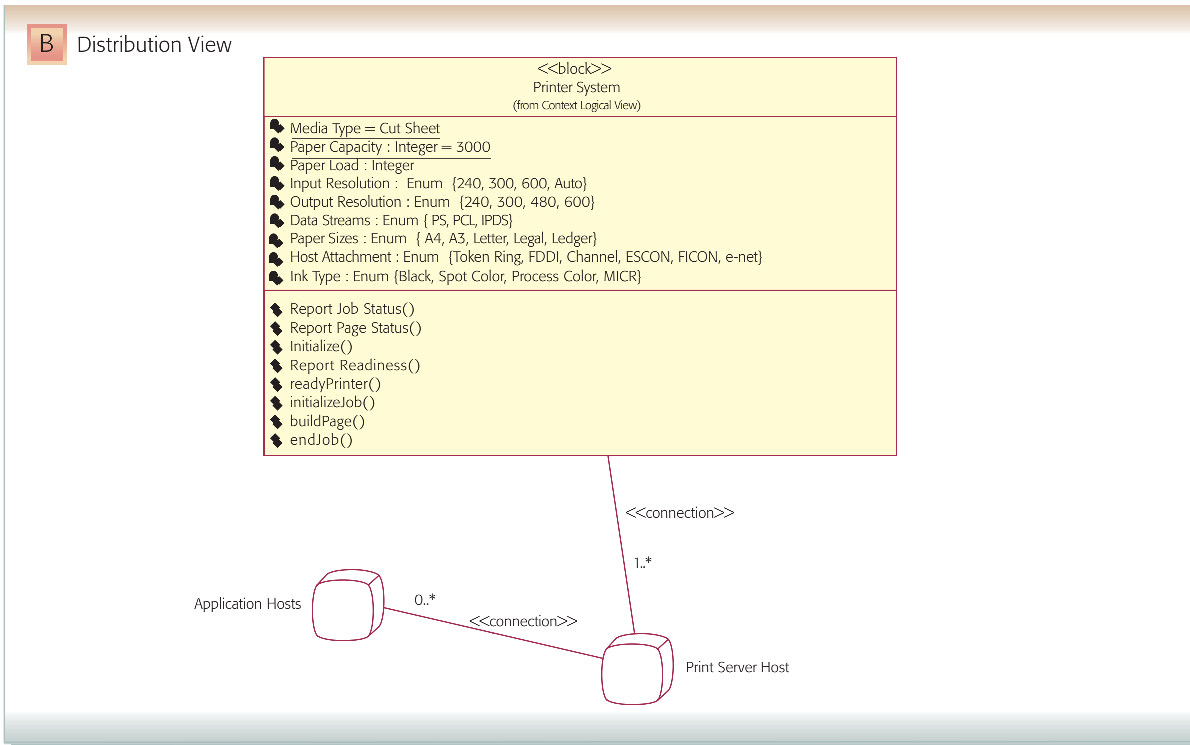


Figure 2
Logical and distribution context for printer example: (A) logical view; (B) distribution view

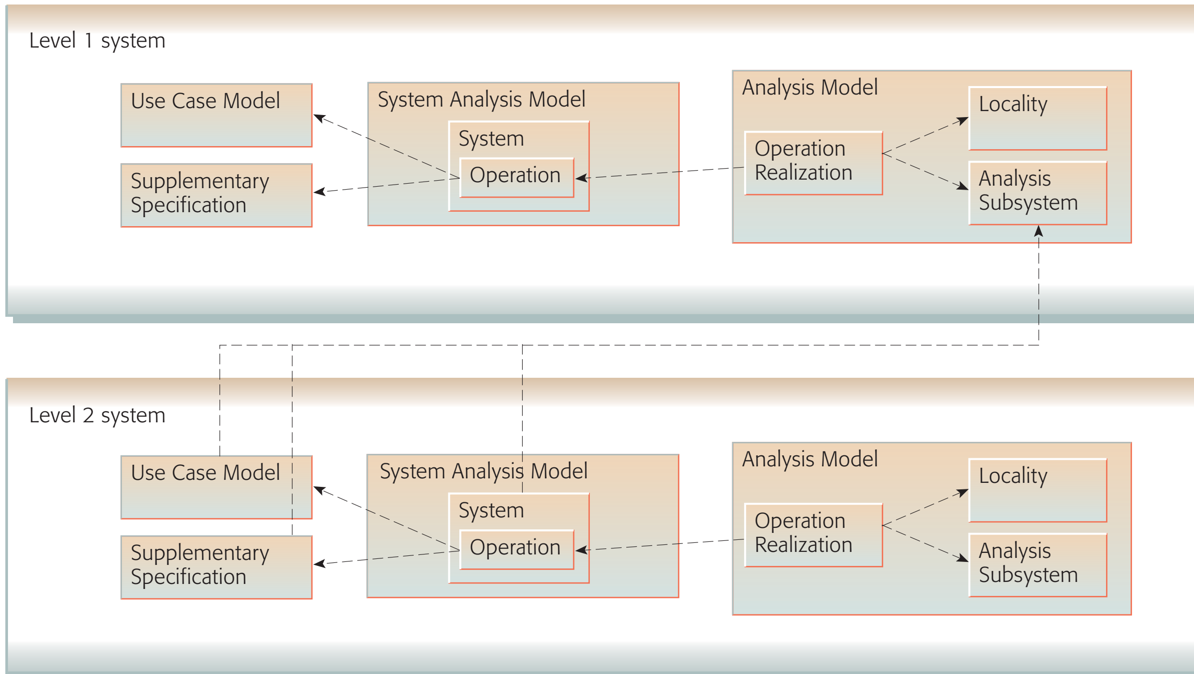


Figure 3
Levels of system decomposition

Figure 4 shows a sequence diagram illustrating the realization of a use case initiated by a client requesting the operation "step 1" from analysis subsystem 1. This sequence diagram describes the internal workings of the system and is therefore a white-box view of the system that encapsulates the analysis subsystems 1, 2, and 3. The diagram is also a black-box view of the interactions between these analysis subsystems. In the context of the overall system this is a white-box view, but within this white-box view are black-box views of the collaborations of the various analysis subsystems. For analysis subsystem 2, the operations "step 2" and "step 4" are derived requirements in the context of a black-box view of this subsystem.

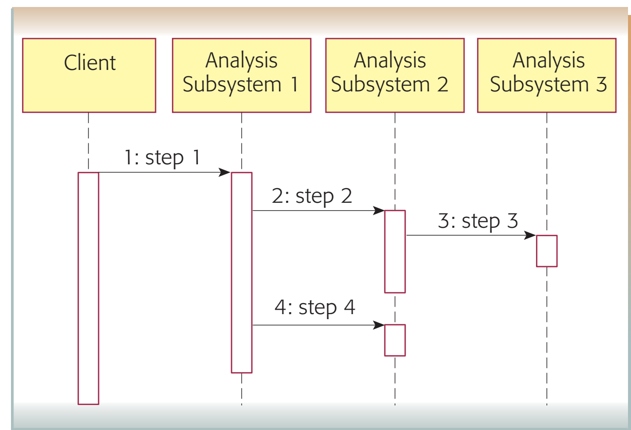


Figure 4
Sequence diagram for operations analysis

Operations analysis uses sequence diagrams to recursively derive system-component black-box requirements at every level of the hierarchy. An operation realization is created for each operation, and the realization is performed concurrently across the system components identified in the architectural analysis activity.

Joint realization

In developing the system model, use cases are written, system components are defined, and the interactions between the components are described. This is standard practice for modeling a system. For large-scale developments, we must also decompose the system, divide and suballocate the requirements, and develop links for traceability purposes. The new mechanism for connecting all of these items is a Joint Realization Table (JRT). The joint realization method is how the JRT is completed and is therefore

Table 4 Partial JRT for printing a page

White-Box Step	Action Performed	White-Box Budgeted Rqts	Distribution Reference (Locality)	Process Reference
1	**LRF1: I/O Services** *WBS1*: receives the block and stores in an available data buffer in memory	*SUP1*: 10 ms	DRF1: Printer Control Unit	PRF1: Data_rec
2	**LRF2: I/O Services** *WBS2*: updates the Input Data Buffer Queue with the address of the received block and sends the awaiting process input data buffer queue address list to the Raster Image Processing subsystem.	*SUP2*: 2 ms	DRF2: Printer Control Unit	PRF2: Input_data_buff_mgt
3	**LRF3: Raster Image Processing** *WBS3*: reads the buffer queue address list and begins reading the data blocks. As the blocks are processed, one or more page bitmaps are rendered to memory and stored in available page bitmap buffers.		DRF3: Printer Control Unit	PRF3: Page_RIP
4	**LRF4: Raster Image Processing** *WBS4*: indicates the input data block is available for reuse after the block is read and processed.		DRF4: Printer Control Unit	PRF4: Input_data_buff_mgt

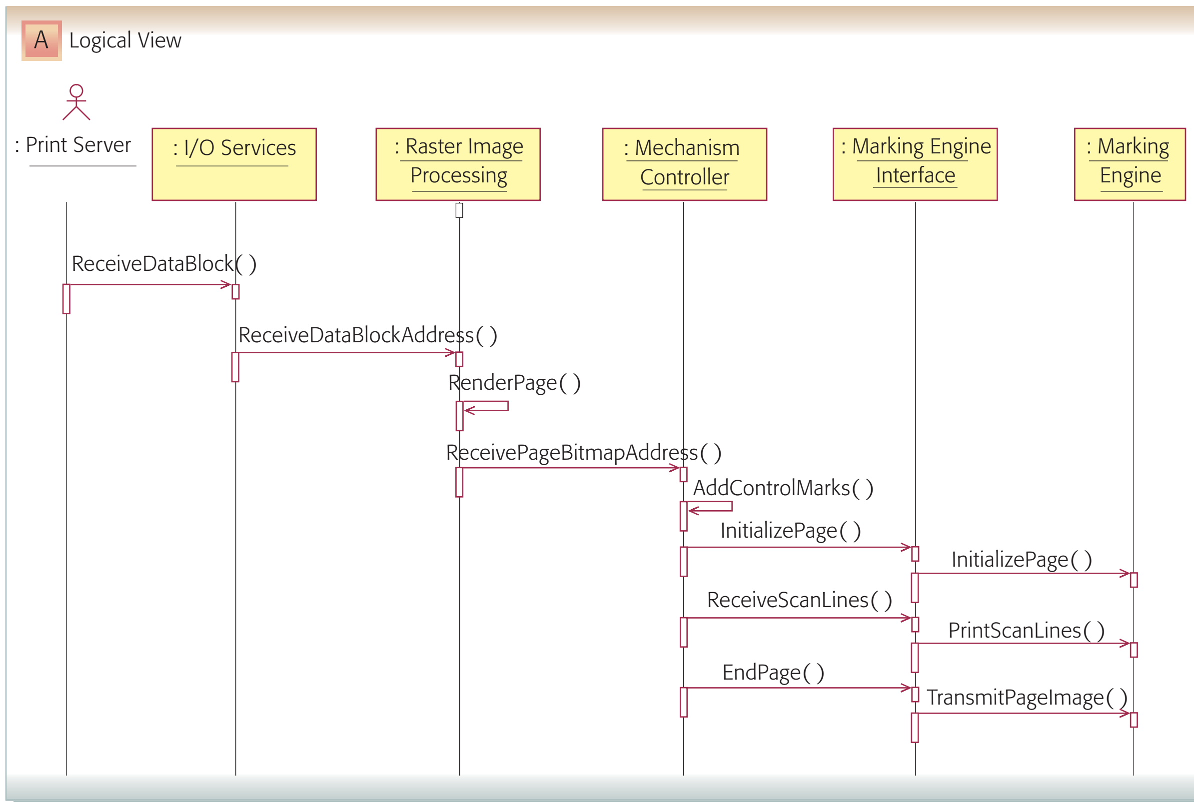
the process by which decomposition is accomplished within MDSD.

In MDSD, we distinguish between functional requirements and nonfunctional requirements (NFRs). Functional requirements describe the system behavior as well as the collaboration among system components to accomplish the system behavior. NFRs pertain to how a system performs its functions and include concerns such as quality, quantity, and timeliness.

JRTs decompose the system in the context of the logical and physical architectures and, at the same time, assign nonfunctional requirements to these system components. In a real sense, this is the missing link—the item that was needed to connect object-oriented development models to the needs of the engineering community developing large-scale systems.

A JRT example that decomposes the task of printing a page is found in ***Table 4***. The header material for the Build Page operation provides context for elaborating the JRT. This JRT decomposition allocates the functionality of the single black-box operation to white-box printer entities. The Action Performed column captures both the logical entity performing the action and the logical step performed. In this example, two logical entities, I/O Services and Raster Image Processing, collaborate to print a page. NFRs are allocated to the logical white-box steps in the White Box Budgeted Requirements column—for example, 10 milliseconds are allocated to the I/O Services to receive and store an available data block in memory. The last two columns provide the distribution and process references. In this example the printer-control-unit locality and `Data_rec` process must perform the task of receiving a block and putting it into memory within 10 milliseconds.

The JRT maintains context, captures the logical and distribution decomposition, and provides for the allocation of nonfunctional requirements. With the JRT in place, it is useful to represent the content in SysML as a coupled set of sequence diagrams showing the same flow in the different viewpoints. ***Figure 5*** shows the sequence diagrams for the print page service. The insights gained by modeling the various elements (analysis subsystems, localities, etc.) may lead to their refactoring and refinement until the needed set of interactions are identified and



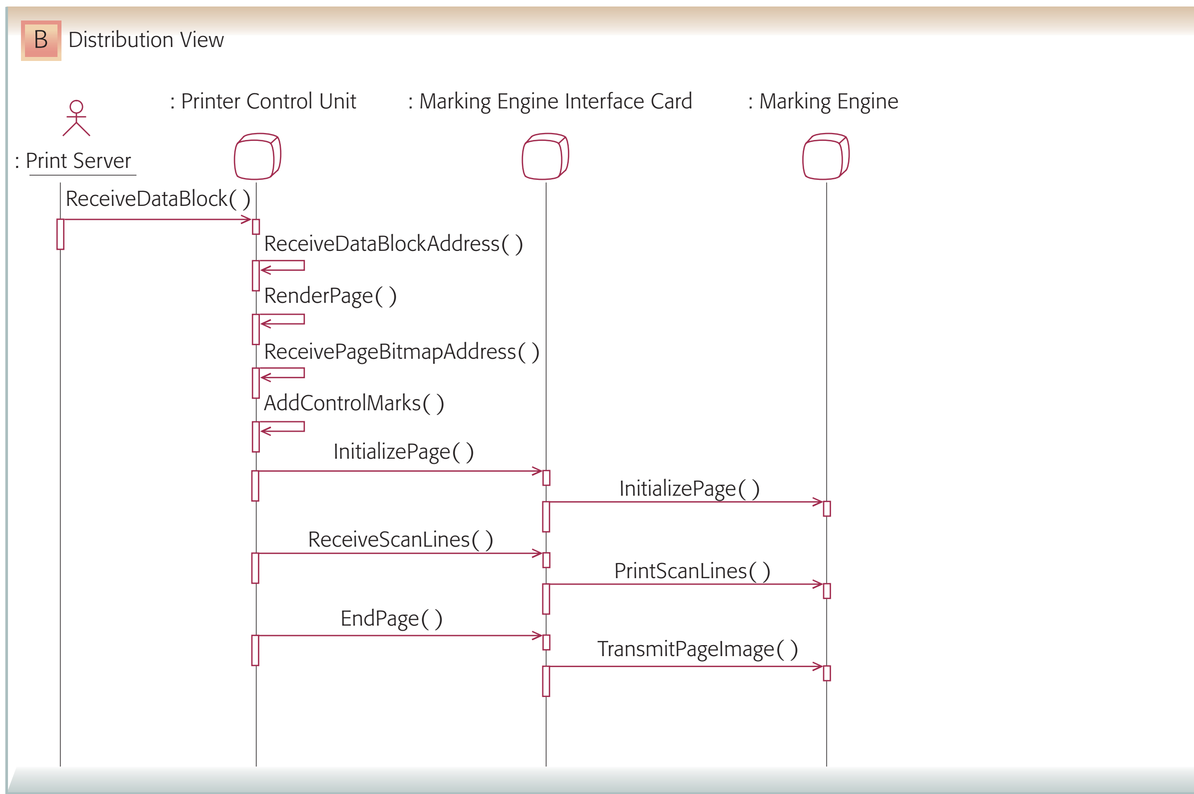


Figure 5
Logical and distribution sequence diagrams for print page flow: (A) logical view; (B) distribution view

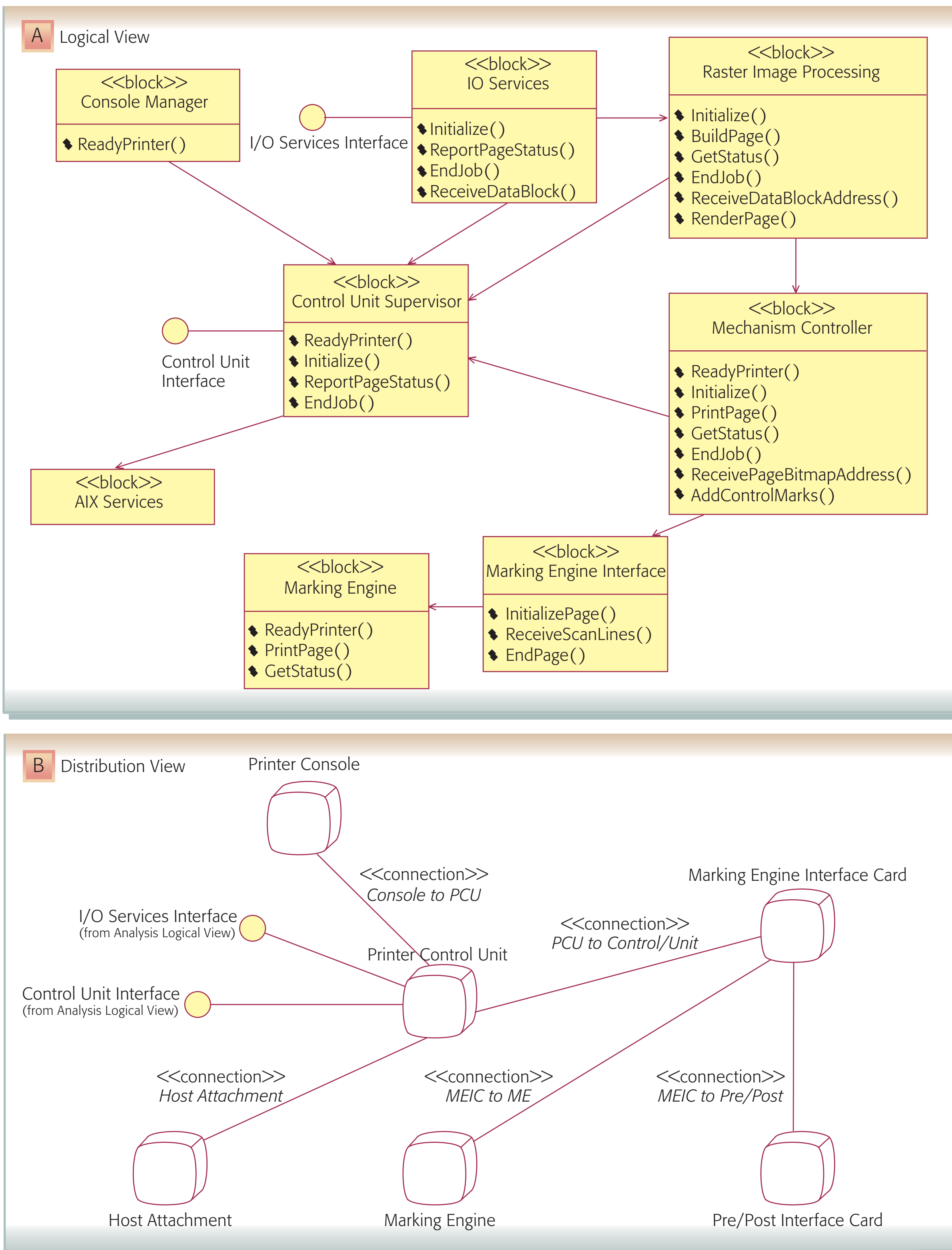


Figure 6
Association of logical entities, localities, and interfaces: (A) logical view; (B) distribution view

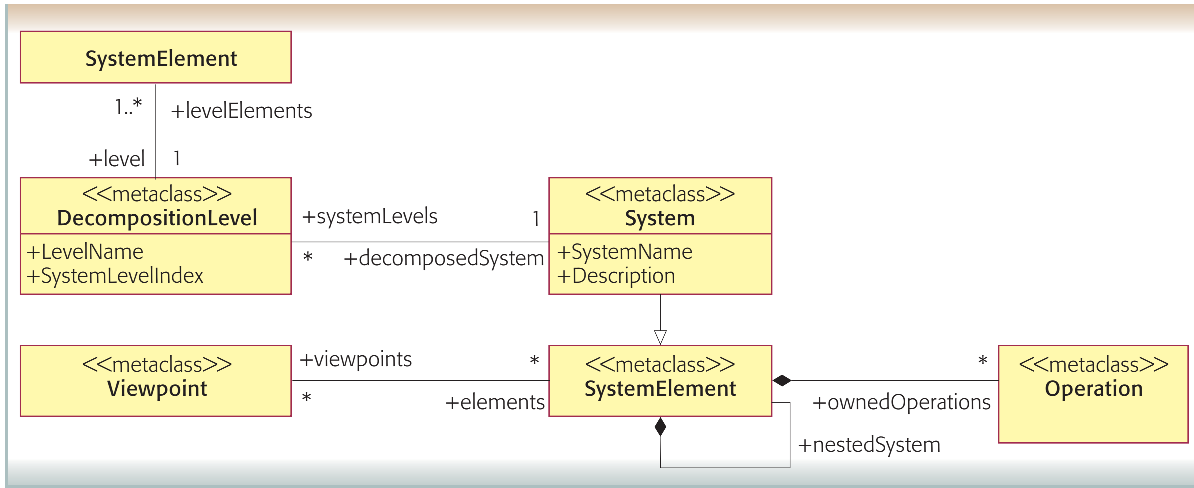


Figure 7
Metamodel representation of a system (partial)

assigned to them. The candidate operations may also be refactored and refined as a result of the insights gained from the model.

Next, we must link the information in the JRT to a model of the system. To do so, it is necessary to identify the subsets of operations that are performed by a particular locality. Examples from the JRT are the "receive" and "store block" operations, which are performed by both the I/O-services subsystem and the printer-control-unit locality. An initial set of interfaces can be derived by considering the mapping of operations to localities. In addition, cohesion principles should be applied to specify interfaces and then the mapping of operations to localities should be used as a check to ensure that the minimum requirement (the "split" of operations across localities for a given analysis subsystem) is satisfied. The resulting analysis-level logical and distribution views are shown in *Figure 6*.

The NFRs associated with operations are handled using the same approach through all levels of the architecture in successively more detail until sufficient detail is developed to implement the operations. The NFRs not associated with operations are budgeted directly to localities by using standard techniques from systems engineering.

Requirement derivation

With current requirements-driven development methods, the system's NFRs are often found in a software requirements specification or similar document. The engineers decompose the functional requirements and then document them in a specification tree. The objective is to continue to suballocate functionality into ever-finer levels of granularity until the details are sufficiently documented for development to proceed.

MDSD differs from this approach by decomposing the system into components, in contrast to traditional methods that decompose the requirements into a specification tree. MDSD is able to recursively define the component architecture at each level of the hierarchy; after this, the NFRs are suballocated to the components. The JRT is used in this approach to link the system behavior, logical components, physical components, and NFRs into a coherent model that maintains context and traceability throughout the system analysis. With this method, MDSD provides a robust means for system decomposition and modeling.

THE RUP SE METAMODEL

The RUP SE metamodel[16] is implemented by using UML and SysML. An author of this paper (Balmelli) is one of the lead authors of the submission for the SysML language accepted by the Object Management Group, Inc., an industry consortium generating the SysML specification as a response to its RFP (request for proposal).

The semantics in these languages are sufficient for most viewpoints. However, the distribution view-

point actually introduces (through the RUP SE metamodel) novel concepts whose implementation requires some care.

In modeling terms, we define a system as shown in ***Figure 7***. In this figure, we use the Meta-Object Facility (MOF**)[17] as a means to represent the metaclasses that form a system. The metaclasses in this model formally support the MDSD framework. Together they form the metamodel, which is implemented by applications supporting the RUP SE framework. The metamodel describes an abstract syntax and rationalizes our method. The applications that support this metamodel make use of a concrete syntax (or notation) to represent these concepts (e.g., graphically). The complete metamodel supporting RUP SE is available in a specification[16] that describes the formal aspects of our framework.

As explained previously, RUP SE includes the novel concept of localities. We implement this concept with SysML blocks. Connections, that is, communication links between localities, are implemented by using Flow Parts, Item Flows, and Flow Specifications. The attributes are defined according to the application. To keep our framework flexible, localities (as well as other types of system elements) are modeled using a profile extension to the RUP SE metamodel. This is also true for viewpoints and model levels.

CONCLUSION

In this paper, we have summarized the key elements of model-driven systems development, illustrating the method with an example involving the use of a mainframe printer model.

Because MDSD is built from system first principles, it applies to a wide variety of systems, including intelligence information systems, electronic products, and embedded devices. MDSD scales from small to large projects and has been used successfully on a wide range of development programs. MDSD treatment of functional requirements, including analysis of the system context, leads to robust solutions that provide a solid basis for evolving the system as stakeholder needs change. Because MDSD is based on system decomposition rather than requirements decomposition, the resulting systems are better suited for internal reuse. The MDSD joint realization method focuses attention on system specification at each level of specificity, encourages cross-domain collaboration, and facilitates traceability of model elements from system requirements.

MDSD continues to evolve. With the growing complexity of products and Web-based integration, there is a greater interest in "system thinking" in many development organizations. As this trend continues, we expect that MDSD principles will evolve further and that industry-specific usage patterns will emerge.

*Trademark, service mark, or registered trademark of International Business Machines Corporation.

**Trademark, service mark, or registered trademark of Object Management Group, Inc. in the United States, other countries, or both.

CITED REFERENCES

1. G. E. Mogyorodi, "What Is Requirements-Based Testing?," *Crosstalk, The Journal of Defense Software Engineering* **16**, No. 3 (March 2003).
2. D. Alberts, J. Garstka, and F. Stein, *Network Centric Warfare*, 2nd Edition, CCRP Press, Washington, D.C. (1999).
3. *OMG Systems Modeling Language (OMG SysML)*, Object Management Group, Inc., http://syseng.omg.org/SysML.htm.
4. B. Blanchard and W. Fabrycky, *Systems Engineering and Analysis*, 3rd Edition, Prentice Hall, Upper Saddle River, NJ (1997).
5. G. Booch, J. Rumbaugh, and I. Jacobsen, *The Unified Modeling Language Users Guide*, 2nd Edition, Pearson Education, Inc., Upper Saddle River, NJ (2005).
6. M. Cantor, "Organizing RUP SE Projects," *The Rational Edge* (2003).
7. M. Cantor, "Rational Unified Process for Systems Engineering: Part 1—Introducing RUP SE Version 2.0," *The Rational Edge* (August 2003).
8. M. Cantor, "Rational Unified Process for Systems Engineering: Part 2—System Architecture," *The Rational Edge* (September 2003).
9. M. Cantor, "Rational Unified Process for Systems Engineering: Part 3—Requirements Analysis and Design," *The Rational Edge* (October 2003).
10. J. Putman, *Architecting with RM-ODP*, Prentice Hall, Upper Saddle River, NJ (2001).
11. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE Std 1471-2000*, Institute for Electrical and Electronic Engineers (2004), http://standards.ieee.org/reading/ieee/std_public/description/se/1471-2000_desc.html.
12. M. Cantor, "The Rational Unified Process for Systems Engineering 1.0," *The Rational Edge* (June 2001).
13. J. Connallen, *Building Web Applications with UML* (2nd Edition), Addison-Wesley, Reading, MA (2002).

14. K. Bittner and I. Spence, *Use Case Modeling*, Addison-Wesley, Reading, MA (2003).
15. H. Lykins, S. Friedenthal, and A. Meilich, "Adapting UML for an Object-Oriented Systems Engineering Method (OOSEM)," *Proceedings of the Tenth Annual INCOSE Symposium*, International Council on Systems Engineering (July 2000), http://www.omg.org/docs/syseng/02-06-11.pdf.
16. L. Balmelli, J. Densmore, D. L. Brown, M. Cantor, B. Brown, and T. Bohn, *Specification for the Rational Unified Process for Systems Engineering—Semantics and Metamodel*, Technical Report RC23966, IBM Thomas J. Watson Research Center, Hawthorne, NY 10532 (May 2006).
17. OMG's MetaObject Facility, Object Management Group, Inc., http://www.omg.org/mof.

Accepted for publication February 28, 2006
Published online July 25, 2006.

Laurent Balmelli
IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, New York, 10532 (balmelli@us.ibm.com). Dr. Balmelli is a research staff member at the Watson Research Center and a member of several leadership councils in IBM. Since 2003, he has represented IBM within the SysML submission team and is one of the lead authors of the SysML language specification. His areas of expertise are metamodeling, integrated product development solutions, and systems engineering methodologies. He holds more than 15 United States and international patents. In addition, he has coauthored more than 15 conference and journal papers and is the recipient of two Best Technical Paper awards.

David Brown
IBM Rational Software, 1385 Bison Ridge Drive, Colorado Springs, CO 80919 (dave.brown@us.ibm.com). Mr. Brown is a Senior Certified IT Specialist and leader of the Rational Solution Architecture community of practice. His areas of expertise include software and systems architecture and process. He has made significant contributions to the creation of the Rational Unified Process for Systems Engineering (RUP SE) and its adoption by multiple clients.

Murray Cantor
IBM Rational Software, 340 W. 72nd Street, New York, New York 10023 (mcantor@us.ibm.com). Dr. Cantor is an IBM Distinguished Engineer and a member of the Rational CTO team. His areas of expertise include software and systems engineering processes and systems-development management and leadership. Dr. Cantor is the lead architect of the Rational Unified Process for Systems Engineering (RUP SE), the extension of the Rational Unified Process for system and enterprise engineering. In addition, he served as Rational's technical liaison to the Object Management Group Systems Engineering Domain Special Interest Group and was a founding member of SysML partners, an industry consortium devoted to responding to the OMG request for proposal for a systems profile. He is the author of the books *Object-Oriented Project Management with UML*, published by John Wiley & Sons in 1998, and *Software Leadership*, published by Addison-Wesley in October 2001.

Michael Mott
IBM Federal Systems, Strategy and Technology, 195 Mountain City Highway, #16, Elko, Nevada 89801 (mottmi@us.ibm.com). Mr. Mott is an IBM Distinguished Engineer. His areas of expertise include large-scale development, systems architecture, and high-performance technical computing. Mr. Mott recently retired from Boeing and his last position was the Chief Architect for a team developing a very large-scale, high-performance ground station. His current interests are the application of service-oriented architecture principles to ground stations and improving development methods for large-scale systems. ■