from The Rational Edge: The first in a three-part series, this article introduces the Systems Modeling Language (SysML), a general-purpose, graphical modeling language for product and systems development. Part 1 describes SysML requirements, use-case, and test-case diagrams.

Today's competitive pressures and other market forces drive manufacturing companies to improve the efficiency with which they design and manufacture products and systems. Across the product lifecycle, one area where there has been a notorious lack of efficiency support is the conceptual stage, during which the functional architecture (and sometimes the physical architecture) is decided upon.

The conceptual stage follows the transformation of customer needs into product functions and use cases, and precedes the design of these functions across the engineering disciplines (for example, mechanical, electrical, software, etc.). A lack of support during product conceptualization makes it difficult to efficiently trace the realization of requirements in the product. The lack of a formal representation for concepts also results in an inadequate ability to make decisions at the level of systems in the product, such as during feasibility studies. Moreover, the lack of a clear vision of the product architecture hinders team understanding and communication, which in turn often increases the risk of integration issues. It is these and other challenges confronted during the conceptual phase of product and system development that SysML is designed to mitigate.

In this Part 1 of a three-part article, I explain the basic purpose and value of SysML, relate it to Unified Modeling Language (UML), and describe its Requirements diagram, Use-Case diagram, and test-case representations. In subsequent parts, I'll cover SysML's structural and behavioral
diagrams, as well as its allocation mechanism. Over the course of the entire article, I'll explore all
the diagrams available in SysML, as well as most of the constructs associated with each diagram.
To provide a context for the discussion, I'll utilize a detailed, real-life example of an embedded
system, a Rain Sensing Wiper for an automotive application.

Introducing SysML

The SysML standard gives systems engineers and architects a much-needed way to collaborate
using a common language that is specifically designed to support this process. As a standard
modeling language for systems engineering, SysML will enable improved communications
across development teams, while greatly enhancing our ability to manage ever-growing system
complexity. Further, by enabling an electronic representation of the product design, SysML opens
the door to analytics for faster and more effective decision-making across the entire systems
development lifecycle.

SysML and UML

SysML is based on the Unified Modeling Language (UML), a de facto standard for software
engineering developed within the Object Modeling Group (OMG) consortium. SysML was
developed as a response to the request for proposal (RFP) issued by the OMG in March 2003.
The development team includes representatives from more than ten companies. IBM has played a
leadership role in the definition of the SysML standard by authoring part of the specification.

![Comparison of SysML 1.0 with UML 2.0](image)

Figure 1: Comparison of SysML 1.0 with UML 2.0

The text in Figure 1 summarizes the various diagrams available in SysML. Requirements,
Parametrics, and Allocations are new diagrams not available in UML. Activity and Block diagrams
are reused from UML 2.0 and extended in SysML. Lastly, state machines, interactions, and use
cases are reused from UML 2.0 without modification.

SysML is a modeling language for representing systems and product architectures, as well as their
behavior and functionality. It builds on the experience gained in the software engineering discipline
of building software architectures in UML (think of the classic Class diagram.) The architecture
SysML describes represents the elements realizing the functional aspect of the system. The
physical aspect is sometimes represented, too; for example, when the architecture represents how
the software is deployed on a set of computing resources.
A SysML example: the Rain Sensing Wiper

I explore the capabilities of SysML through an example: a Rain Sensing Wiper (RSW) system. This example is inspired by a real-life product, which experienced some challenges during its initial development. (See Appendix A for background on the RSW project.) This example serves to illustrate the importance of understanding a product at the level of its sub-systems, in order to prevent complex failure modes leading to costly product recalls. In this case, the RSW manufacturer endured a lengthy (hence costly) root-cause analysis that eventually required a design change. In the example used in this article, I present a model that is resilient to the failure the manufacturer experienced.

The goal of the RSW is to wipe the surface of the windshield automatically (i.e., without user intervention) whenever droplets of liquid are detected on the windshield’s exterior surface. In addition, the amount of liquid detected dictates the speed of the wiper. This system has three main components: (1) the software that controls the behavior of the wiper, (2) an electronic control unit that executes the software, and (3) a sensor fixed on the windshield’s interior surface, whose task is to sense droplets through the windshield.

In this article, I present a detailed model that describes many aspects of this system. This example constitutes a realistic product in the area of embedded electronics for the automotive industry, whose design greatly benefits from a SysML representation.

The Context diagram

When modeling a system, an important primary task is to decide what belongs to the system and what does not. The Context diagram is an informal (in the sense that it does not carry precise semantics) way to represent the boundaries of the system. Note that the Context diagram is not represented in Figure 1 because it is actually built using a Block (Definition) diagram. Figure 2 shows a Context diagram for the RSW.
The Context diagram establishes the scope of the system. In this instance, it identifies three actors for the system: Maintenance (for repair purposes), Car Electrical System (to activate the system in the car), and Driver (to manually disable the system, for example). Three external systems are considered: the wiper interface, the windshield, and the car electrical system. Note that a user-defined keyword "external" is used to qualify the external components. Note also that the car electrical system also provides electrical power to the RSW. Hence, it is considered both an actor and external system.

The Requirements diagram

As noted above, the conceptual stage of the development lifecycle secedes to the analysis of the customer needs, which evolve into product requirements. Requirements have traditionally been represented as text, often accompanied by figures and drawings, and stored in files or databases. The requirements describe all the product functions, as well as the constraints under which these functions should be realized.

SysML allows the representation of requirements as model elements. Hence, requirements become an integral part of the product architecture. The language offers a flexible means by which to represent text-based requirements of any nature (e.g., functional or non-functional) as well as the relationships between them. Figure 3 shows a Requirements diagram for the RSW.
Note that the Requirements diagram contains both functional and non-functional requirements. Requirements in SysML are abstract classifiers -- that is, they cannot be instantiated -- with no operations or attributes. It is nevertheless possible to add attributes to a requirement element by adding attributes to the Requirement stereotype. Using the stereotype’s attributes allows values to be assigned to these attributes at design time. Requirements cannot participate in associations or generalizations; however, a set of predefined relationships help to characterize the relationships between the requirements. I will review these relationships below.

Sub-requirements are related to their parent requirements using the crosshair relationship, which denotes namespace embedding. For example, in Figure 3, some of the sub-requirements of the requirement *Automatic Wiping* are connected to it using the crosshair. The parent requirement is a package for the embedded requirements. In that sense, deleting the parent requirement

Figure 3: SysML Requirements diagram for the Rain Sensing Wiper system
Click to enlarge.
will automatically delete all the embedded ones. Another example of a requirement acting as a package for other requirements is the requirement Core Functions, which contains two sub-requirements. For readability in the model, a user-defined keyword "package" is rendered next to the Requirement stereotype.

During requirement analysis (e.g., decomposition and flow-down), new requirements are created by derivation. These new requirements can be connected to the initial ones with the DeriveRqt dependency. For example, in Figure 3, a set of core functions for the product are derived from the set of requirements under Automatic Wiping. The name DeriveRqt was chosen in order to avoid any confusion with the standard Derive dependency in UML 2.0. Other examples of derived requirements are the technical choices for each function (see the box Technical Choices in Figure 3). Note that the designer captures a Rationale comment to explain his choice for using a sensor fixed on the windshield. A last example of a derived requirement is the quality requirement System Calibration, which states that the system should be calibrated. This is the requirement added to the product after the infamous RSW failure was identified. The satisfaction of this requirement ensures that the product will be resilient to changes in the sensor and windshield characteristics.

Another relationship between requirements is Refine. An example of requirement refinement is shown in Figure 3. The requirement on speed actuation is refined by the possible selection for speed: slow, medium, or fast. Lastly, a generic Trace dependency can be used to emphasize that a pair of requirements are related in some way or another. In Figure 3, the requirement for manual deactivation is traced to the one about automatic deactivation.

Requirements have a number of derived attributes to store the status of the relationships reviewed in the above paragraphs. As I explain in Part 2 of this article, these attributes become particularly handy when requirement relationships are represented outside requirements diagrams.

Often, requirements are elicited during the entire product lifecycle, and additional Requirements diagrams are used to represent them. Hence, the product requirements are typically laid out on a set of requirement diagrams.

SysML provides a generic model for requirements that allows the modeling of both functional and non-functional requirements. A non-normative set of requirement types are proposed in the appendix of the OMG SysML specification. Specific types of requirements (for example, related to timing or scheduling, etc.) are handled by language extensions. SysML supports a profile mechanism to extend the language. The OMG has released a series of modeling standards that address specific needs for the modeling of non-functional requirements related to performance and quality, and for the modeling of test cases. These profiles can be used in SysML without restriction.

The Use-Case diagram and test cases

SysML provides a Use-Case diagram that is inherited from UML 2.0 without modifications. In Figure 4, we show the interaction of the external actors with some of the main use cases (represented by ellipses) owned by the RSW.
Figure 4: SysML Use-Case diagram for the Rain Sensing Wiper system

This Use-Case diagram represents the three actors and connects them to their respective use cases. A central use case called Automatic Wiping is composed of a series of sub use cases. The hierarchical relationship is modeled using the Include dependency.

SysML also has the capability to represent test cases and to attach them to their related requirements or use cases. A test case can be an operation or a behavioral model (Interaction, State Machine, or Activity).

Figures 5 and 6 show a test case for the RSW. The test case verifies the requirement System Calibration (see Figure 3). This is accomplished as follows:

- First, retrieving the characteristics of the different components (sensor, windshield, and software configuration file).
- Second, using these characteristics to compute an operating range for both the sensor and windshield in order to assess their compatibility. If the sensor and windshield are compatible, then the test case is successful. Otherwise, an alert is triggered. The actions in the activity diagram contributing to each step are enclosed in illustrative boxes (see Figure 5) for the sake of clarity.
Figure 5: SysML Test-Case diagram for the quality requirement System Calibration. The test case is realized with an activity diagram. 
Click to enlarge.

The first step is realized in this example using a set of Web services to access the repositories containing the artifacts related to the different components (see the leftmost enclosing box in Figure 5). More precisely, the Bill of Materials (as stored in a product data management system, for instance) is queried for the characteristics of the sensor and the windshield. Then the configuration file is retrieved, for example, from a software configuration management system.

The second step (see the central box in Figure 5) is realized by defining constraints on the attributes of the sensor and the windshield in SysML Parametric diagrams. I'll explain how these diagrams are constructed in Part 2 of this article.

In Figure 6, a test context is created to host prototypes for the Web services and other functions necessary to execute the test case. This context is traced to the requirement. The Activity diagram (described in Part 2 of the article) uses the functions of the test context for its execution.
Figure 6: SysML Test-Case diagram for the quality requirement System Calibration: requirement and test-case traceability
Click to enlarge.

Next month

In this month’s installment, Part 1, I’ve introduced SysML and presented an overview of its capabilities for modeling requirements, use cases, and test cases. In upcoming installments, I'll show how SysML is used to create a product structure that satisfies some of these requirements and how to use its allocation mechanism.

Notes

1 Assigning class values to the classifier attributes would provide an alternative, but classifier attributes are also meant to hold values at runtime, which contradicts the semantic of a requirement in SysML.


6 Read all three parts of this series:

1. An overview of the Systems Modeling Language for product and systems development -- Part 1: Requirements, use-case, and test-case modeling
2. An overview of the Systems Modeling Language for product and systems development -- Part 2: Structuring the Rain Sensing Wiper system
3. An Overview of the Systems Modeling Language for product and systems development -- Part 3: Modeling system behavior

© Copyright IBM Corporation 2006
Trademarks