SysML DISTILLED

A Brief Guide to the Systems Modeling Language

LENNY DELLIGATTI

Forewords by RICK STEINER *and* RICHARD SOLEY

OMG SysML™

Praise for *SysML Distilled*

"In keeping with the outstanding tradition of Addison-Wesley's technical publications, Lenny Delligatti's *SysML Distilled* does not disappoint. Lenny has done a masterful job of capturing the spirit of OMG SysML as a practical, standards-based modeling language to help systems engineers address growing system complexity. This book is loaded with matter-of-fact insights, starting with basic MBSE concepts to distinguishing the subtle differences between use cases and scenarios to illumination on namespaces and SysML packages, and even speaks to some of the more esoteric SysML semantics such as token flows."

— *Jeff Estefan, Principal Engineer, NASA's Jet Propulsion Laboratory*

"The power of a modeling language, such as SysML, is that it facilitates communication not only within systems engineering but across disciplines and across the development life cycle. Many languages have the potential to increase communication, but without an effective guide, they can fall short of that objective. In *SysML Distilled,* Lenny Delligatti combines just the right amount of technology with a common-sense approach to utilizing SysML toward achieving that communication. Having worked in systems and software engineering across many domains for the last 30 years, and having taught computer languages, UML, and SysML to many organizations and within the college setting, I find Lenny's book an invaluable resource. He presents the concepts clearly and provides useful and pragmatic examples to get you off the ground quickly and enables you to be an effective modeler."

— *Thomas W. Fargnoli, Lead Member of the Engineering Staff, Lockheed Martin*

"This book provides an excellent introduction to SysML. Lenny Delligatti's explanations are concise and easy to understand; the examples well thought out and interesting."

— *Susanne Sherba, Senior Lecturer, Department of Computer Science, University of Denver*

"Lenny hits the thin line between a reference book for SysML to look up elements and an entertaining book that could be read in its entirety to learn the language. A great book in the tradition of the famous *UML Distilled*."

— *Tim Weilkiens, CEO, oose*

- You cannot define a generalization between an actor and a block.
- An actor cannot have parts; that is, it cannot appear at the composite end of a composite association. (We always regard an actor as a "black box.")

3.9 Value Types

Like a block, a **value type** is an element of definition—one that generally defines a type of quantity. I say "generally" because there are two value types in SysML—*Boolean* and *String*—that arguably are not quantities.

You can use a value type in many places throughout your model. Most often, it appears as the type of a **value property**, which is a kind of structural feature of blocks. (Section 3.4.1.3, "Value Properties," has more details.) But that's not the only place where value types make an appearance; they're actually ubiquitous in system models. They can also appear as the types of the following:

- Atomic flow ports on blocks and actors
- Flow properties in flow specifications
- Constraint parameters in constraint blocks
- Item flows and item properties on connectors
- Return types of operations
- Parameters of operations and receptions
- Object nodes, pins, and activity parameters within activities

There are three kinds of value types—primitive, structured, and enumerated—that you typically define in your system model. A **primitive** value type has no internal structure (it doesn't own any value properties). Its notation is a rectangle with the stereotype «valueType» preceding the name.

SysML defines four primitive value types: *String*, *Boolean*, *Integer*, and *Real*. You can, of course, define your own primitive value types as specializations (subtypes) of these four. For example, Figure 3.25 shows three value types (°, *V*, and ° *C*) that are subtypes of *Real*.

As its name implies, a **structured** value type has an internal structure—generally two or more value properties. As with a primitive value type, the notation for a structured value type is a rectangle with

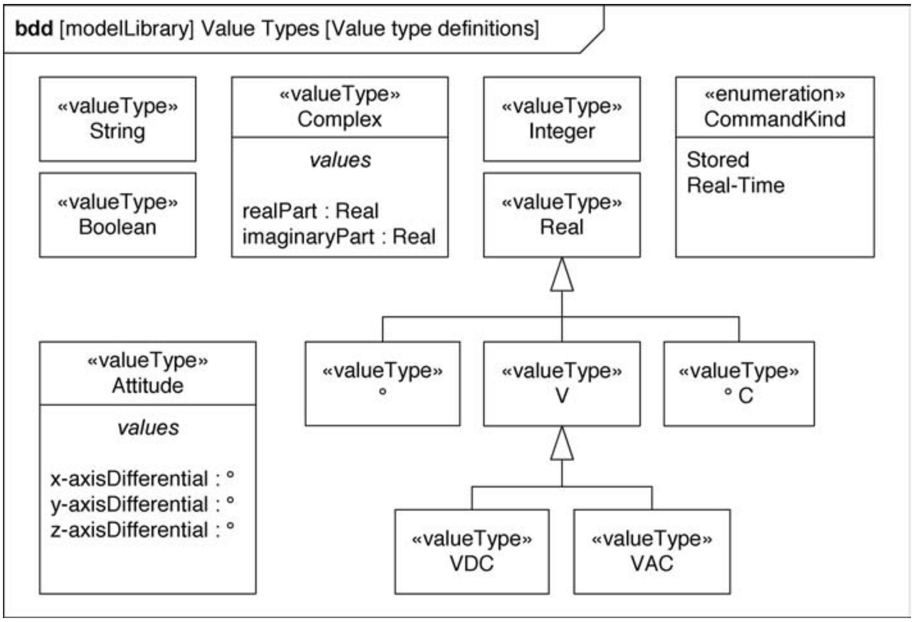

Figure 3.25 *Value types*

the stereotype «valueType» preceding the name. SysML defines one structured value type: *Complex*. Its structure consists of two value properties—*realPart* and *imaginaryPart*—that are both of type *Real*. One structured value type may, in turn, be the type of a value property within another structured value type. In this way, you can create arbitrarily complex systems of value types.

An **enumerated** value type—colloquially called an **enumeration**—simply defines a set of **literals** (legal values). If a parameter of an operation (or some other kind of element shown in the earlier bulleted list) is typed by an enumeration, then the value it holds at any moment must be one of the literals in that enumeration. The BDD in Figure 3.25 shows an enumeration named *CommandKind,* which defines two literals: *Stored* and *Real-Time.* I could use this enumeration, for example, to type an input parameter named *kind* in an operation named *buildCommand*. When a client calls this operation (within a running system), the only legal values it can pass are *Stored* and *Real-Time.*

I mentioned earlier that value types can be related to one another by using generalizations. A value type hierarchy can be arbitrarily deep, and generalizations—as you may recall—are transitive. For example, Figure 3.25 conveys that the value types *VDC* and *VAC* are (indirectly) subtypes of *Real*. The principle of substitutability applies here just as it does in the case of generalizations between blocks: Values of type *VDC* and *VAC* will be accepted wherever their supertypes (*V* and *Real*) are required. These supertypes are abstractions. And the principle of designing to an abstraction—and its consequent extensibility—also applies to this practice of creating a value type hierarchy. This is a widely used and powerful modeling practice.

3.10 Constraint Blocks

Like a block, a **constraint block** is an element of definition—one that defines a Boolean **constraint expression** (an expression that must evaluate to either *true* or *false*). Most often, the constraint expression you define in a constraint block is an equation or an inequality: a mathematical relationship that you use to constrain value properties of blocks. You would do this for two reasons:

- To specify assertions about valid system values in an operational system
- To perform engineering analyses during the design stage of the life cycle

The variables in a constraint expression are called **constraint parameters**. Generally, they represent quantities, and so they're typed most often by value types. For example, Figure 3.26 shows a constraint block named *Transfer Orbit Size,* which defines a constraint expression that contains three constraint parameters: *semimajorAxis, initialOrbitRadius,* and *finalOrbitRadius.* These three constraint parameters are typed by the value type *km*.

Constraint parameters receive their values from the value properties they're bound to—that is, the value properties that are being constrained. At any given moment, those values either satisfy the constraint expression, or they don't; the system is either operating nominally, or it isn't. Note, however, that a BDD by itself can't convey which constraint parameters and value properties are bound to one another. You would express this piece of information on a parametric diagram. (I discuss this in detail in Chapter 9.)

The notation for a constraint block on a BDD is a rectangle with the stereotype «constraint» preceding the name. The constraint expression always appears between curly brackets ({}) in the constraints compartment. The constraint parameters in the constraint expression are listed individually in the parameters compartment.

You sometimes build a more complex constraint block from a set of simpler constraint blocks. You would do this to create a more complex mathematical relationship from simpler equations and inequalities. The more complex constraint block can display its constituent parts as a list of **constraint properties** in the constraints compartment. Recall from Section 3.4.1.4 that a constraint property has a name and a type in the format *name : type*. The type, as mentioned earlier, must be the name of a constraint block.

For example, Figure 3.26 shows that the constraint block *Hohmann Transfer* is composed of two constraint properties—*ttof* and *tos*—which

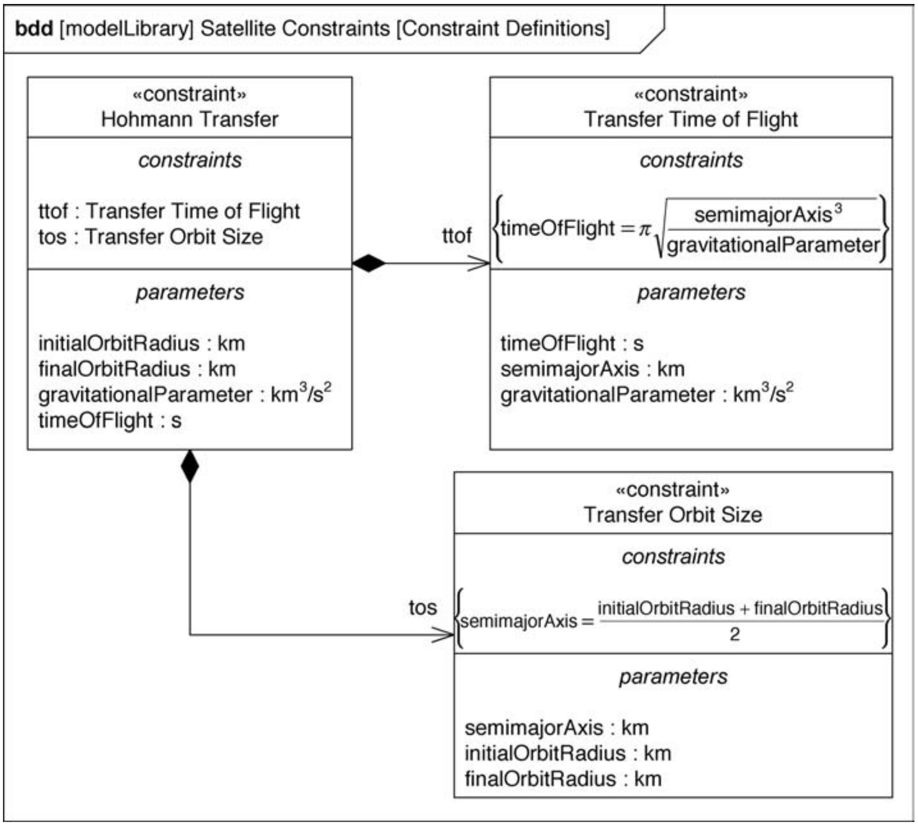
bdd [modelLibrary] Satellite Constraints [Constraint Definitions]
«constraint» Hohmann Transfer
constraints
ttof : Transfer Time of Flight
tos : Transfer Orbit Size
parameters
initialOrbitRadius : km
finalOrbitRadius : km
gravitationalParameter : km³/s²
timeOfFlight : s
ttof
«constraint» Transfer Time of Flight
constraints
{timeOfFlight = π √(semimajorAxis³ / gravitationalParameter)}
parameters
timeOfFlight : s
semimajorAxis : km
gravitationalParameter : km³/s²
tos
«constraint» Transfer Orbit Size
constraints
{semimajorAxis = (initialOrbitRadius + finalOrbitRadius) / 2}
parameters
semimajorAxis : km
initialOrbitRadius : km
finalOrbitRadius : km

Figure 3.26 *Relationships between constraint blocks*

represent usages of the constraint blocks *Transfer Time of Flight* and *Transfer Orbit Size*, respectively. This model conveys that *Hohmann Transfer* defines a constraint expression that is a composite of two simpler constraint expressions—in effect, defining a more complex mathematical relationship.

Note, though, what this BDD doesn't (and can't) convey: *where* those two simpler constraint expressions are specifically connected to each other to create the composite constraint expression. A parametric diagram would convey this additional piece of information (more on this in Chapter 9).

As an alternative to the constraints compartment notation, you can use composite associations to convey that one constraint block is composed of other, simpler ones (as shown in Figure 3.26). Note that the role names shown on the part ends of the two composite associations correspond to the names of the constraint properties in the *Hohmann Transfer* constraint block. These are equivalent notations. You use composite associations when you need to expose the details of the simpler constraint blocks; in contrast, you use the constraints compartment notation to hide those details when they're not the focus of the diagram.

3.11 Comments

SysML has a lot of rules (and they all exist to serve the very useful purpose of giving your design unambiguous meaning from one reader to the next). However, you sometimes need to express information on a diagram in an unconstrained way as a block of text. You can do this with a comment.

A **comment** is, in fact, a model element. It consists of a single attribute: a string of text called the **body**. You can convey any information you need to in the body of a comment, and you can optionally attach a comment to other elements on a diagram to provide additional information about them. You can use comments on any of the nine kinds of SysML diagrams.

The notation for a comment is commonly referred to as a **note symbol**: a rectangle whose upper-right corner is bent. You use a dashed line to attach a comment to other elements (as shown at the bottom of the BDD in Figure 3.27). If you need to, you can attach a comment to several model elements simultaneously by using a separate dashed line for each one.

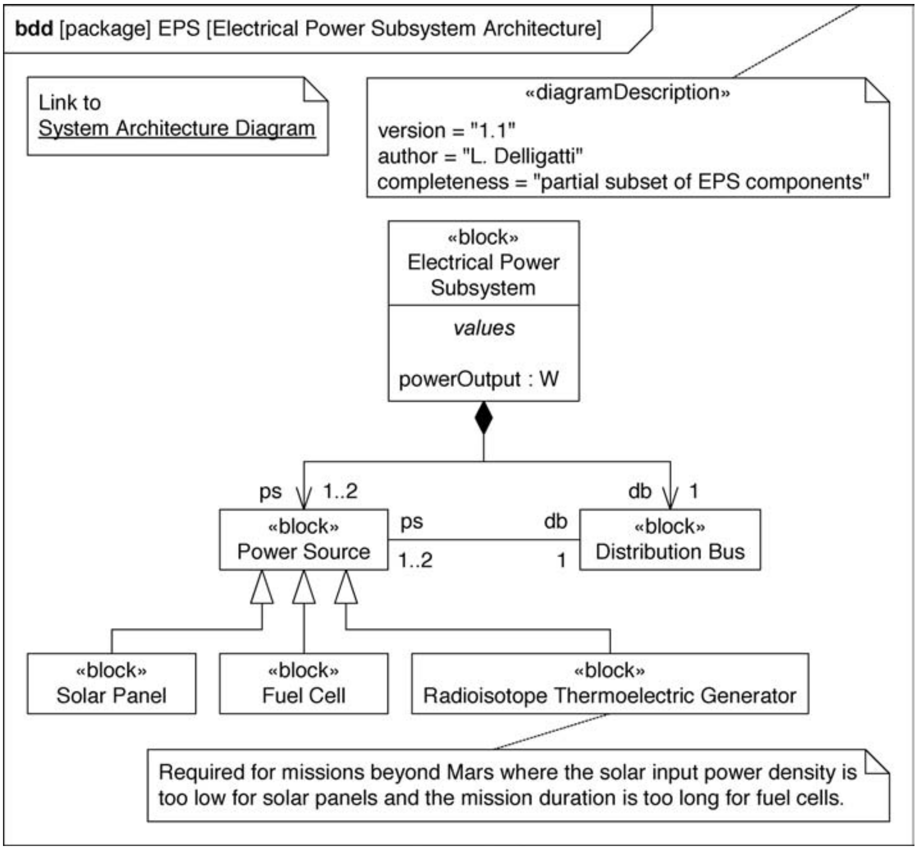
bdd [package] EPS [Electrical Power Subsystem Architecture]
Link to System Architecture Diagram
«diagramDescription»
version = "1.1"
author = "L. Delligatti"
completeness = "partial subset of EPS components"
«block» Electrical Power Subsystem
values
powerOutput : W
ps 1..2
db 1
«block» Power Source
ps
db
1..2
1
«block» Distribution Bus
«block» Solar Panel
«block» Fuel Cell
«block» Radioisotope Thermoelectric Generator
Required for missions beyond Mars where the solar input power density is too low for solar panels and the mission duration is too long for fuel cells.

Figure 3.27 *Comments on a BDD*

Modelers sometimes put freestanding comments with hyperlinks on a diagram to enable readers to quickly navigate to a related diagram in the model (or to an external document). An example of this is shown in the upper-left corner of the BDD in Figure 3.27. To be clear, though, this capability is a function of the modeling tool you use; not all tools do this. And SysML itself says nothing about this capability.

SysML defines some specialized kinds of comments: rationale, problem, and diagram description. These appear as a note symbol with the respective stereotype preceding the body of the comment. Figure 3.27 shows an example of a diagram description comment in the upper-right corner of the BDD. Modelers often use rationale comments in conjunction with requirements relationships and allocations. I discuss these topics in detail in Chapters 11 and 12.