SysML v2

THE DEFINITIVE GUIDE TO

MODEL-BASED SYSTEMS ENGINEERING

FROM FIRST PRINCIPLES TO INDUSTRIAL PRACTICE

FROM FIRST PRINCIPLES
Understand every construct from the ground up.

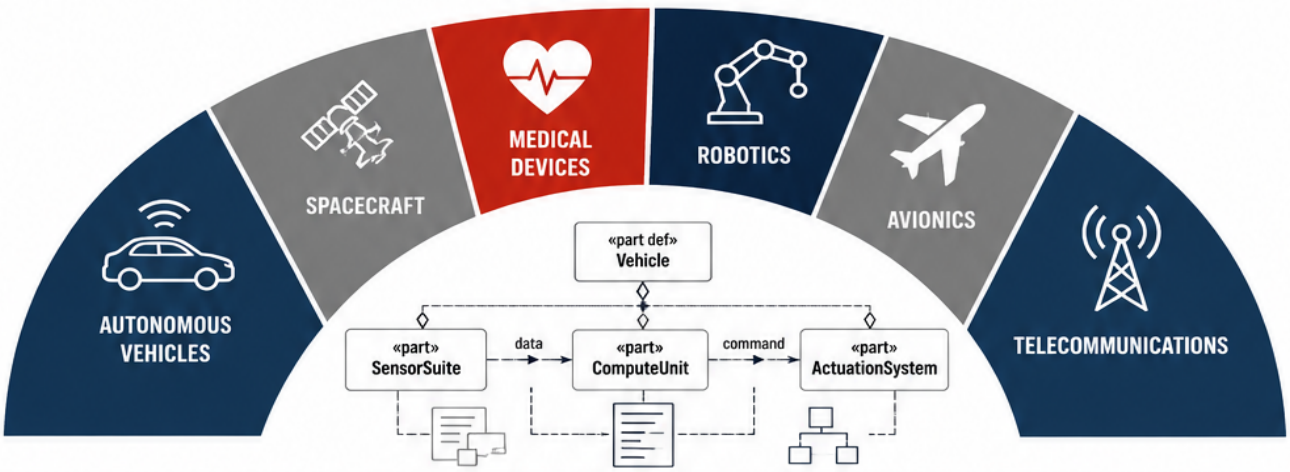
FORMAL SEMANTICS & KerML FOUNDATION
Explore the precise semantics and the KerML basis.

TEXTUAL & GRAPHICAL NOTATIONS
Master both notations with complete syntax coverage.

PRACTICAL ENGINEERING APPLICATIONS
Apply SysML v2 to solve real engineering challenges.

REAL-WORLD CASE STUDIES
Learn through end-to-end examples across major industries.

FOR ENGINEERS, ARCHITECTS & TOOL DEVELOPERS
The authoritative resource for practitioners.



STEVE T.

SysML v2: The Definitive Guide to Model-Based Systems Engineering

From First Principles to Industrial Practice

Steve Publications

This book is available at
https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering

This version was published on 2026-07-31

Contents

From First Principles to Industrial Practice

This book provides complete, technically rigorous coverage of the Systems Modeling Language version 2 (SysML v2) for model-based systems engineering. It explains every construct in the language from first principles, including its formal semantics defined by the OMG specification, the KerML foundation, textual and graphical notations, and practical engineering applications. Through detailed explanations, complete syntax descriptions, real-world examples, and end-to-end case studies spanning autonomous vehicles, spacecraft, medical devices, robotics, avionics, and telecommunications, this book serves as both a comprehensive learning resource for newcomers and an authoritative reference for experienced systems engineers, architects, researchers, and tool developers working with SysML v2 in modern digital engineering environments.

Introduction: Why Models, and Why SysML v2?

In January 2015, the European Space Agency's Philae lander touched down on comet 67P/Churyumov-Gerasimenko, marking humanity's first soft landing on a comet. The landing itself was a triumph of engineering precision across hundreds of organizations and thousands of individual components. Yet within minutes of touchdown, the harpoons designed to anchor the probe failed to fire, and Philae bounced twice before settling in shadowed terrain where its solar panels could not recharge. The mission was not a failure; it returned extraordinary data for months. But the anchoring issue revealed something fundamental about complex systems: no matter how expertly individual components are engineered, failures at integration boundaries can undermine the entire endeavor.

This is the problem that model-based systems engineering seeks to solve. In document-centric engineering, requirements live in word processors, architectures in drawing tools, interfaces in spreadsheets, test plans in separate trackers, and safety analyses in yet another repository. When a requirement changes, engineers manually trace its impact through dozens of artifacts, hoping nothing was missed. The Philae harpoon issue exemplifies this fragmentation: the mechanical design, electrical power budget, software sequencing, and thermal constraints that together determined harpoon behavior were managed in separate silos without a unified model to expose their interactions.

Model-based systems engineering (MBSE) replaces scattered documents with a single coherent model that captures requirements, structure, behavior, interfaces, analysis, and verification in one interconnected representation. When a requirement changes, the model reveals its impact automatically. When two subsystems' constraints conflict, the model exposes the inconsistency before hardware is fabricated. MBSE is not a methodology or a tool; it is an approach to engineering where models become the primary artifacts of design, replacing documents as the authoritative source of truth.

The Systems Modeling Language (SysML) was created to serve as the lingua franca for MBSE. First adopted by the Object Management Group in 2007,

SysML v1 extended UML with constructs tailored for systems engineering: blocks instead of classes, flows instead of objects, requirements and parametric diagrams alongside structural and behavioral views. For nearly two decades, SysML v1 served as the de facto standard for model-based systems engineering across aerospace, defense, automotive, medical devices, and industrial automation. Thousands of engineers learned it; hundreds of organizations built their MBSE practices around it.

Yet SysML v1 carried fundamental limitations that became increasingly apparent as systems grew more complex and digital engineering practices matured. Built as a profile layered on top of UML, it inherited ambiguities from a language designed primarily for software. Its semantics were often underspecified, leaving room for tool vendors to interpret the same diagram differently. It lacked a standardized textual notation, making version control, code review, and automation difficult. Its support for variability modeling, product-line engineering, and formal analysis was bolted on rather than built in. Perhaps most critically, its reliance on proprietary XMI exports meant that models were effectively locked into whichever tool created them.

These limitations motivated the creation of SysML v2. Development began in 2018 through a collaborative effort by the SysML v2 Submission Team (SST), a consortium of industry experts, academics, and tool vendors. The result, formally adopted by OMG on July 21, 2025, is not an incremental update but a fundamental redesign. SysML v2 is built from the ground up on a new semantic foundation called KerML (Kernel Modeling Language), which provides mathematically rigorous semantics independent of UML. It introduces a fully standardized textual notation alongside graphical diagrams. It unifies requirements, structure, behavior, and analysis in a single coherent framework. And it defines a standard API that enables true tool interoperability for the first time.

This book is your guide to SysML v2. It assumes no prior knowledge of SysML or MBSE, yet provides sufficient depth to serve as a professional reference for experienced practitioners. We begin with first principles and build systematically toward mastery. You will learn not only how to use every construct in the language but also why each exists, what engineering problem it solves, and how it fits into the broader architecture of model-based systems engineering.

The journey ahead is structured in three phases. First, we establish foundations: what MBSE is and why it matters, the KerML language that underlies SysML v2, and the core organizational constructs of namespaces, packages,

and imports. Second, we explore the full expressive power of SysML v2: structural modeling with parts, ports, and interfaces; behavioral modeling with state machines, actions, and interactions; requirements engineering with traceability and verification; analysis modeling with constraints and trade studies; and advanced patterns for variability, product lines, and reuse. Third, we address practical concerns: tooling ecosystems, migration strategies from SysML v1.x, integration with digital engineering workflows, and lessons learned from industrial-scale deployments.

Throughout the book, you will encounter complete examples in both textual SysML v2 notation and corresponding diagrammatic representations. The textual notation is not secondary to diagrams; it is equally expressive and often more precise for detailed specifications. We explain every keyword, relationship, constraint, and modeling decision rather than assuming prior knowledge. Real-world case studies progressively build realistic systems from stakeholder needs through requirements, architecture, behavior, analysis, and verification, demonstrating how every SysML v2 element contributes to a coherent whole.

By the end of this book, you will be able to design, implement, and maintain SysML v2 models for complex engineering systems with confidence. You will understand the formal semantics behind every construct you use. You will know when to apply each relationship type and why one choice is preferable to another in a given context. You will have seen how requirements trace through architecture to behavior to verification in a single integrated model. And you will be equipped to evaluate SysML v2 tooling, plan migrations from legacy models, and contribute to the evolving practice of model-based systems engineering in your organization.

The era of document-centric systems engineering is ending. Complex modern systems, including autonomous vehicles navigating dynamic environments, spacecraft operating billions of kilometers from Earth with no possibility of physical repair, and medical devices that must guarantee safety under all conceivable conditions, demand a more rigorous approach. SysML v2 provides the language for that approach. Let us begin.

Chapter 1: The Case for SysML v2: MBSE, History, and Architectural Vision

The story of how we got to SysML v2 begins with the challenges of engineering increasingly complex systems in an era of digital transformation. To understand why SysML v2 was designed the way it was, we must first understand what problems it seeks to solve, what came before it, and what architectural decisions distinguish it from its predecessor.

What Is Model-Based Systems Engineering

Model-based systems engineering is an approach where models serve as the primary artifacts of system design, replacing or augmenting documents as the authoritative source of truth about a system's requirements, architecture, behavior, interfaces, and verification criteria. The core idea is simple but profound: instead of describing a system through disconnected text documents, diagrams in different tools, and spreadsheets tracking relationships between them, we capture all that information in a single coherent model with well-defined semantics.

The benefits of this approach are particularly evident for complex systems where many stakeholders contribute to design decisions across multiple disciplines. In document-centric engineering, a requirements engineer writes specifications in a word processor; a mechanical engineer creates CAD models; an electrical engineer designs circuit diagrams; a software architect produces sequence diagrams; a safety analyst documents hazard analyses. Each uses different tools with different formats. When the requirements change, someone must manually update each downstream artifact and hope nothing is missed. Traceability between requirements and design elements exists in spreadsheets or proprietary traceability matrices that are notoriously difficult to keep current.

In MBSE, all these artifacts become views of a single model. A requirement is a first-class model element with typed attributes, constraints, and explicit relationships to the structural components and behavioral elements that satisfy it. When the requirement changes, its relationships reveal exactly which parts of the design are affected. Inconsistencies between requirements and design can be detected automatically through constraint checking rather than manual review.

The INCOSE MBSE Handbook defines MBSE as "the formalized use of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases." The key word is "formalized": models must have precise semantics so that different stakeholders interpret them consistently and tools can reason about them automatically.

MBSE is not a methodology prescribing specific processes or workflows. It is an enabling approach that can support various methodologies, from traditional V-model development to agile and iterative practices. What MBSE provides is a rigorous foundation for capturing design decisions, exploring alternatives, analyzing tradeoffs, and maintaining consistency across the lifecycle of complex systems.

The Failure Modes of Document-Centric Engineering

To appreciate what MBSE with SysML v2 offers, it helps to understand where document-centric engineering fails in practice. These failure modes are not theoretical; they are documented causes of cost overruns, schedule delays, and safety incidents in real-world programs.

The first failure mode is inconsistency. When requirements, architecture, interfaces, and test plans live in separate documents maintained by different teams, they inevitably drift out of alignment. A requirement might specify a performance threshold that the design cannot meet; an interface might be changed without updating dependent subsystems' specifications; a safety constraint might be documented but never translated into verifiable design criteria. Inconsistencies discovered late in development are exponentially more expensive to fix than those caught during initial design.

The second failure mode is incomplete traceability. Regulatory standards across aerospace, medical devices, automotive, and rail require bidirectional

traceability from requirements through design to verification. In document-centric workflows, this traceability is typically maintained in spreadsheets or proprietary tools that link documents by reference IDs. These links are fragile: when a requirement moves to a different section of a document or its ID changes, the traceability chain breaks. Manual maintenance of traceability matrices is tedious and error-prone; studies have found that traceability coverage in large programs often falls well below stated targets because keeping it current requires effort that competes with actual design work.

The third failure mode is limited analysis capability. Engineering decisions require quantitative analysis: can the power system support all subsystems simultaneously? Will the thermal design keep components within operating limits under worst-case conditions? Is the control system stable across the entire flight envelope? In document-centric engineering, these analyses are performed in separate tools (MATLAB/Simulink, SPICE, finite element analysis software) with inputs and outputs manually transferred from documents. This process is slow, error-prone, and makes it difficult to explore design alternatives systematically because each change requires regenerating inputs for all downstream analyses.

The fourth failure mode is knowledge loss. When a program ends or team members leave, institutional knowledge about design decisions and their rationale often disappears with them. Documents capture what was decided but rarely why. In MBSE, the model itself preserves this knowledge: requirements include rationale attributes; design alternatives are captured as variants with documented trade study results; assumptions about the operating environment are explicit model elements rather than implicit understanding held by individual engineers.

The fifth failure mode is tool lock-in. Document-centric workflows often involve a heterogeneous set of tools, each with proprietary formats. Integrating these tools requires custom connectors and data translation scripts that are fragile and difficult to maintain. When one vendor changes their format or discontinues a product, the entire integration ecosystem can break. This lock-in limits organizations' ability to choose best-of-breed tools for specific tasks and creates dependency on single vendors for critical capabilities.

SysML v2 was designed with these failure modes in mind. Its formal semantics reduce inconsistency by ensuring all stakeholders interpret models the same way. Its integrated requirements, architecture, behavior, and verification constructs support native traceability without external spreadsheets.

Its constraint-based analysis modeling enables quantitative evaluation directly within the system model. Its textual notation and standard API reduce tool lock-in by enabling version control, code review, and interoperable data exchange.

SysML v1.x: Achievements and Limitations

SysML version 1 was adopted by OMG in 2007 as an extension of UML tailored for systems engineering needs. It introduced nine diagram types: block definition diagrams for structural decomposition, internal block diagrams for component interconnections, parameter value diagrams for constraints, requirement diagrams for requirements management, use case diagrams for stakeholder interactions, activity diagrams for workflows and algorithms, state machine diagrams for mode-based behavior, sequence diagrams for message exchanges, and timing diagrams for real-time constraints.

SysML v1 achieved significant success as a standard for MBSE practice. It provided a common vocabulary that enabled communication across disciplines. Major aerospace programs, automotive platform developments, medical device designs, and defense systems used SysML v1 models to manage complexity. Tool vendors invested heavily in SysML v1 support, creating mature environments with simulation capabilities, requirements management integration, and code generation features.

However, SysML v1 carried fundamental limitations that became increasingly problematic as systems grew more complex and digital engineering practices evolved. Understanding these limitations is essential for appreciating the design decisions in SysML v2.

The first limitation was its foundation on UML. SysML v1 was defined as a profile of UML, meaning it extended UML's metamodel with stereotypes and constraints rather than defining its own semantic foundation. UML was designed primarily for software engineering, where concepts like classes, objects, inheritance, and messages map naturally to programming language constructs. Systems engineering deals with physical components, continuous quantities, real-time behavior, safety requirements, and integration across hardware, software, firmware, and human operators. Forcing these concepts into a software-centric metamodel created awkward mappings and ambiguities.

For example, SysML v1's block concept attempted to serve multiple roles

simultaneously: as a type definition (like a UML class), as an instance specification (like a UML object), and as a structural element with both data properties and behavioral operations. This conflation made it difficult to distinguish between defining what something is versus specifying how it is used in a particular context. The distinction between "definition" and "usage" existed informally but was not formally specified, leading to inconsistent interpretations across tools.

The second limitation was loose semantics. Many SysML v1 constructs were underspecified in the OMG standard, leaving room for tool vendors to implement them differently. Two tools might both claim SysML v1 compliance yet interpret the same diagram differently. This ambiguity undermined one of MBSE's core promises: a shared model that all stakeholders can rely on. When semantics are unclear, models become illustrations rather than specifications.

The third limitation was the absence of a standardized textual notation. SysML v1 relied exclusively on graphical diagrams created in visual modeling tools. While diagrams excel at communication and high-level visualization, they are poorly suited for version control, code review, diff comparison, automated testing, and text-based search , practices that have become standard in software development and are increasingly expected in systems engineering,. Without a textual notation, SysML v1 models were effectively binary blobs locked inside proprietary tools.

The fourth limitation was weak support for variability and product-line engineering. Many products exist as families of variants sharing common platforms: automotive model lines with different body styles and powertrains, aircraft configurations for military versus civilian missions, medical device variants for different markets. SysML v1 had no native constructs for modeling variability; engineers resorted to stereotypes, tagged values, or separate models for each variant. These approaches were fragile, difficult to maintain, and made it hard to ensure consistency across variants.

The fifth limitation was limited interoperability. SysML v1 tools exchanged models using XMI (XML Metadata Interchange), but XMI implementations varied widely in completeness and correctness. Importing a model from one tool into another often resulted in lost information, corrupted relationships, or unreadable content. There was no standard API for programmatic access to models, making integration with analysis tools, PLM systems, and other engineering applications require custom connectors that were expensive to develop and maintain.

The sixth limitation was requirements as second-class citizens. While SysML v1 included requirement diagrams and a requirement stereotype, requirements were not fully integrated with the rest of the model. Linking requirements to design elements relied on dependency relationships that lacked precise semantics. Formal verification of whether a design satisfied its requirements was essentially impossible within the modeling language itself; it required external tools and manual correlation.

These limitations did not invalidate SysML v1 or the MBSE practices built around it. Thousands of engineers successfully used SysML v1 to manage complexity in real-world programs. But as systems grew more complex, safety-critical, and software-intensive, and as digital engineering practices demanded tighter integration across tools and disciplines, these limitations became obstacles that incremental updates could not overcome. A fundamental redesign was needed.

The OMG's Vision for SysML v2

The requirements for SysML v2 were developed through an 18-month effort culminating in the SysML v2 RFP issued by OMG on December 8, 2017. The response came from the SysML v2 Submission Team (SST), a consortium led by key industry experts including representatives from Boeing, Thales, and academic institutions, along with tool vendors and independent consultants.

The SST's vision was ambitious: create a systems modeling language that is precise enough for formal analysis, expressive enough for real-world engineering problems, usable enough for broad adoption, and interoperable enough to support heterogeneous tool ecosystems. The submission team identified several guiding principles that shaped every design decision in SysML v2.

First, precision over convenience. Every construct must have unambiguous semantics defined independently of any particular tool or graphical notation. Models should be interpretable consistently by humans and machines alike. This principle led directly to the creation of KerML as a formal semantic foundation.

Second, systems engineering from the ground up. Rather than extending a software-centric language with systems concepts bolted on, SysML v2 was designed specifically for systems engineering needs. Physical components,

continuous quantities, real-time behavior, safety requirements, and multi-disciplinary integration are first-class concerns, not afterthoughts.

Third, textual parity with graphical notation. The language must support both textual and graphical representations that are equally expressive and semantically equivalent. Textual notation enables version control, code review, automation, and tool independence. Graphical notation supports communication, visualization, and stakeholder engagement. Neither should be primary; they are different views of the same underlying model.

Fourth, unified semantics across all modeling concerns. Requirements, structure, behavior, analysis, and verification should not be separate sub-languages with their own syntax and semantics loosely connected by relationships. They should be integrated aspects of a single coherent language where requirements can reference structural elements, behaviors can be allocated to components, and analysis constraints can evaluate design properties , all within the same semantic framework.

Fifth, interoperability by design. The language must define standard interfaces for model access, data exchange, and tool integration. Organizations should be able to use best-of-breed tools for different tasks while maintaining a consistent model representation across the tool chain. This principle led to the Systems Modeling API specification developed alongside SysML v2.

Sixth, backward compatibility through transformation. Existing investments in SysML v1 models must not be stranded. A formal transformation specification should enable systematic migration from SysML v1 to SysML v2 while preserving modeling intent as much as possible. This principle resulted in the SysML v1-to-v2 Transformation Specification included in the official release.

The development process was collaborative and transparent. The SST maintained public repositories on GitHub where draft specifications, example models, and training materials were available for community review. Regular face-to-face meetings brought together language designers, tool implementers, and domain experts to debate design decisions. Beta versions of the specification were released for community evaluation, with feedback incorporated iteratively.

After nearly seven years of development, OMG approved final adoption of SysML v2.0 on July 21, 2025. The release included four complementary specifications: the SysML v2.0 Language Specification defining syntax, seman-

tics, libraries, and conformance requirements; the KerML v1.0 Specification providing the foundational semantic kernel; the Systems Modeling API and Services v1.0 Specification enabling tool interoperability; and the SysML v1-to-v2 Transformation Specification supporting migration from legacy models.

KerML: A New Foundation for Systems Modeling

The most significant architectural decision in SysML v2 was to build it on a new semantic foundation rather than extending UML. This foundation is KerML, the Kernel Modeling Language, which provides a mathematically rigorous basis for modeling systems while remaining accessible for practical engineering use.

KerML is not itself a systems modeling language; it is a kernel language designed to be extended by domain-specific languages. Think of it as the core operating system upon which SysML v2 is built as an application layer. KerML defines fundamental modeling concepts , including types, features, classifiers, definitions, usages, expressions, and relationships, with precise semantics that do not depend on any particular domain vocabulary.

The KerML specification is organized in layers, each building on the previous one:

The Root layer provides syntactic foundation without model-level semantics. It defines the basic organizational constructs: Element (the abstract base for everything), Relationship (connections between elements), and Namespace (containers that organize elements with scoping rules). These constructs exist to enable organization and navigation but carry no engineering meaning.

The Core layer introduces semantic base concepts. Type is the fundamental concept for classifying things; Classifier specializes Type for types that can have instances; Feature represents a property or relationship of a classifier (attributes, parts, ports, operations); Behavior models dynamic aspects including actions and state machines. The Core layer establishes what it means for something to be a type, how features describe types, and how behaviors model change over time.

The Kernel layer adds declarative semantics through Definition and Usage constructs. This is where KerML's most important innovation lives: the strict separation between definitions (reusable blueprints or types) and usages (contextual applications of those types in specific situations). Definitions specify what something is; usages specify how something is used in a particular

context. This distinction, which existed informally in SysML v1, becomes explicit and enforceable in KerML-based modeling.

KerML's semantics are grounded in first-order logic concepts. Types correspond to sets of things sharing common properties. Features correspond to functions mapping instances to values or related instances. Specialization corresponds to set inclusion (a subtype's instances are a subset of its supertype's instances). These logical foundations enable formal reasoning about models: consistency checking, conformance verification, and automated analysis become possible because the meaning of every construct is precisely defined.

SysML v2 extends KerML by adding systems engineering vocabulary on top of these foundational concepts. A "part" in SysML v2 is a specialized kind of Feature from KerML with semantics appropriate for physical or logical components. A "requirement" is a specialized kind of Definition with attributes and constraints suitable for specifying system obligations. An "action" is a specialized Behavior modeling procedural execution. But all these SysML-specific constructs inherit their fundamental semantics from KerML, ensuring consistency across the entire language.

The separation between KerML and SysML v2 provides important benefits. Tool developers can implement KerML-based languages other than SysML by extending the same kernel with different domain vocabularies. Domain experts can create specialized extensions for safety-critical systems, real-time embedded systems, or cyber-physical systems while relying on KerML's consistent foundation. And most importantly for practitioners, the clear layering means that understanding KerML's core concepts provides a unified mental model for all of SysML v2 rather than memorizing disconnected rules for each diagram type.

Textual Notation: Precision, Tool Independence, and Collaboration

One of SysML v2's most visible innovations is its fully standardized textual notation. Unlike SysML v1, where models existed only as graphical diagrams in visual tools, SysML v2 defines a precise text-based syntax that is semantically equivalent to graphical representations. This decision was motivated by practical needs observed in modern engineering workflows.

Textual notation enables version control using standard tools like Git. Engineers can track changes to models over time with fine-grained diffs showing exactly what was added, modified, or removed. Branching and merging support parallel development by different teams working on different aspects of the system. Code review practices from software engineering , such as pull requests, comments, and approvals, become applicable to systems models. These capabilities were essentially impossible with SysML v1's binary XMI model files.

Textual notation supports tool independence. A model written in SysML v2 textual syntax is a plain text file that can be read by any conformant tool. Engineers are not locked into whichever vendor created the initial model; they can switch tools without losing information or spending months on data migration. This interoperability was explicitly designed into the specification and validated through pilot implementations across multiple vendors.

Textual notation enables automation and integration with engineering workflows. Models can be generated from templates, validated by scripts, tested in continuous integration pipelines, and searched using text-based tools. Engineers familiar with programming language syntax find textual SysML v2 more natural for specifying detailed constraints, mathematical expressions, and complex relationships that are cumbersome to represent graphically.

Textual notation facilitates collaboration across distributed teams. Text files merge cleanly in version control systems; graphical model files often conflict when multiple people edit simultaneously. Reviewing a text-based model change is as straightforward as reviewing code changes: the diff shows exactly what was modified. This lowers barriers to collaboration, especially for engineers who are more comfortable with text than with visual modeling tools.

The textual syntax was designed to be readable and writable by humans while remaining unambiguous for machines. Keywords like `package`, `part def`, `attribute`, `port`, `action`, `state`, `requirement`, and `constraint` make the intent of each element clear. Structural hierarchy is indicated by indentation and braces, similar to most modern programming languages. Relationships are expressed with operators: `:` for typing, `:>` for specialization, `subsets` for subsetting, `redefines` for redefinition, `connect` for interconnections.

```
package AutonomousVehicle {
  import standard::ISQ::*;

  part def Vehicle {
    attribute mass : MassValue = 1500 [kg];
    port in controlSignal : ControlMessage;
    port out telemetry : TelemetryData;

    part chassis : Chassis;
    part propulsion : ElectricPropulsion;
    part perception : SensorFusionSystem;
    part planning : MotionPlanner;
    part control : VehicleController;
  }
}
```

This example shows a SysML v2 package defining an autonomous vehicle with typed attributes, directed ports, and internal parts. The same model could be displayed graphically as a block definition diagram or interconnection view, but the textual representation captures all information precisely without ambiguity.

Importantly, textual notation is not intended to replace graphical diagrams. Diagrams remain valuable for communication, visualization, stakeholder engagement, and high-level architectural overview. SysML v2 treats both representations as complementary views of the same underlying model. A well-designed tool should allow engineers to switch seamlessly between text and graphics, editing in whichever representation is most appropriate for the task at hand.

How This Book Is Organized

This book is structured to take you from first principles through advanced mastery of SysML v2. Each chapter builds on previous material, so we recommend reading sequentially rather than jumping around. However, once you have completed the foundational chapters, later chapters can be consulted as reference material for specific topics.

The Introduction (this section) establishes why model-based systems engineering matters, traces the evolution from SysML v1 to v2, and presents the architectural vision that makes v2 fundamentally different.

Chapter 2 explains KerML, the semantic foundation of SysML v2. Understanding KerML is essential for grasping why SysML v2 constructs behave the way they do. We cover definitions versus usages, features, multiplicities, typing rules, and the core type system.

Chapter 3 covers model organization: namespaces, packages, imports, exports, visibility, and strategies for structuring large-scale models.

Chapter 4 provides a deep dive into SysML v2's polymorphism system: specialization, subsetting, and redefinition. These relationships are among the most powerful and commonly misunderstood constructs in the language.

Chapters 5 through 11 cover the core modeling capabilities of SysML v2: structural modeling with parts, ports, interfaces, and connectors (Chapter 5); value semantics including types, units, quantities, and dimensional analysis (Chapter 6); requirements engineering with traceability and verification (Chapter 7); state machines for event-based behavior (Chapter 8); action models for procedural behavior and control flow (Chapter 9); interactions for message sequences and protocol modeling (Chapter 10); and parametric modeling for constraints, analysis, and trade studies (Chapter 11).

Chapters 12 through 14 address integration and advanced patterns: allocations, dependencies, and traceability across model layers (Chapter 12); views, viewpoints, and stakeholder perspectives for managing complexity (Chapter 13); and variability modeling, product-line engineering, libraries, and reuse strategies (Chapter 14).

Chapter 15 covers the practical ecosystem: tooling landscape, model interchange formats, APIs, repository-based modeling, versioning, and integration with PLM and digital thread systems.

Chapter 16 provides a comprehensive migration guide for practitioners coming from SysML v1.x, mapping concepts, explaining breaking changes, and addressing common challenges.

Chapter 17 presents a complete end-to-end case study of an autonomous vehicle system, progressively building the model from stakeholder needs through requirements, architecture, behavior, analysis, and verification. This chapter demonstrates how all the constructs you have learned integrate into a coherent whole.

Chapter 18 concludes with a synthesis of key insights, reflection on where SysML v2 fits in the broader digital engineering landscape, emerging trends,

and advice for practitioners beginning their MBSE journey.

Throughout the book, every concept is explained with complete syntax descriptions, semantic interpretation, modeling rationale, engineering context, best practices, common mistakes, and practical guidance. Examples are fully worked rather than abbreviated; we explain every statement, keyword, relationship, and modeling decision in depth.

Let us begin with KerML, the foundation upon which everything else is built.

Chapter 2: KerML Foundations: The Core Modeling Language

KerML, the Kernel Modeling Language, is the semantic foundation of SysML v2. Every construct in SysML v2, including parts, ports, requirements, actions, state machines, and constraints, ultimately traces back to KerML concepts. Understanding KerML is not optional for serious SysML v2 practitioners; it provides the unified mental model that makes the language coherent rather than a collection of disconnected diagram types with their own rules.

This chapter explains KerML from first principles. We begin with the layered architecture of the specification, then cover each layer's key concepts: namespaces and organization in the Root layer, types and features in the Core layer, definitions and usages in the Kernel layer. By the end, you will understand why SysML v2 distinguishes definitions from usages, how features work, what multiplicities mean formally, and how typing rules ensure model consistency.

What Is KerML and Why It Exists

KerML was created to solve a fundamental problem: previous modeling languages for systems engineering lacked precise, consistent semantics that could support both human understanding and machine reasoning. SysML v1, built as a UML profile, inherited ambiguities from a language designed primarily for software. Different tools interpreted the same diagrams differently. Engineers disagreed about what certain relationships meant. Formal analysis was difficult because the semantics were underspecified.

KerML's designers set out to create a kernel language with three key properties:

First, formal semantics grounded in well-understood mathematical concepts. Types correspond to sets. Features correspond to functions or relations. Specialization corresponds to set inclusion. These mappings to first-order logic enable precise specification of what every construct means and support automated reasoning about models.

Second, layering that separates syntactic concerns from semantic ones. The Root layer handles organization without engineering meaning. The Core layer introduces semantic concepts like types and features. The Kernel layer adds declarative semantics through definitions, usages, and expressions. Each layer builds on the previous one, enabling clean separation of concerns.

Third, extensibility that allows domain-specific languages to be built on top of the kernel. KerML is not intended as a complete systems modeling language; it is designed to be extended. SysML v2 extends KerML with systems engineering vocabulary. Other languages could extend KerML for safety analysis, real-time systems, or other domains while sharing the same semantic foundation.

The KerML specification defines 79 metaclasses compared to SysML's 172, reflecting its role as a kernel rather than a complete domain language. This smaller core makes it easier to understand, implement correctly, and verify for consistency. Every SysML v2 construct maps cleanly to KerML concepts, ensuring that the full language inherits KerML's semantic rigor.

The Definition-Usage Distinction

The most important architectural concept in SysML v2, inherited directly from KerML, is the strict separation between Definitions and Usages. This distinction resolves ambiguities that plagued SysML v1, where a single "block" concept attempted to serve multiple roles simultaneously.

A Definition is a reusable blueprint or type specification. It defines what something is: its structure, its features, its constraints, its behavior. Definitions are not specific things; they are templates for creating things. Examples in SysML v2 include `part def`, `item def`, `port def`, `action def`, `state def`, `requirement def`, and `constraint def`. The keyword `def` explicitly marks an element as a definition.

A Usage is a contextual application of a definition. It specifies how something defined elsewhere is used in a particular situation. Usages are not instances in the object-oriented sense (they do not represent individual runtime objects); they represent roles or positions within a larger structure. Examples include `part` (without `def`), `attribute`, `port`, `action`, and `requirement`. Usages inherit features from their defining definitions and can add local features, constraints, or multiplicity restrictions.

Consider an automotive example. We define what an engine is once as a reusable blueprint:

```
part def Engine {
  attribute displacement : VolumeValue;
  attribute maxPower : PowerValue;
  port in fuelIn : FuelPort;
  port out powerOut : MechanicalPowerPort;
}
```

This definition specifies the common features shared by all engines: they have a displacement, a maximum power rating, a fuel input port, and a mechanical power output port. It does not describe any particular engine; it describes what it means to be an engine in our model.

We then use this definition wherever an engine appears in our system:

```
part def Vehicle {
  part engine : Engine;
  part transmission : Transmission;

  connect engine.powerOut -> transmission.inputShaft;
}
```

The usage `part engine : Engine` says that within a vehicle, there is a position or role filled by something of type Engine. This usage inherits all features from the Engine definition (displacement, maxPower, fuelIn, powerOut) and can add vehicle-specific features if needed.

The distinction matters for several reasons. First, it enables clean reuse: we define the engine once and reference it in many places without duplicating its specification. Second, it supports specialization: we can define a DieselEngine as a specialized kind of Engine, and any usage typed by Engine automatically accepts either an Engine or a DieselEngine. Third, it clarifies semantics: when we modify the Engine definition, all usages of that engine inherit the change, making it clear what is affected.

A common source of confusion is whether usages are types or instances. The answer is subtle: usages are types in the sense that they classify things within a specific context, but they are not standalone definitions. A usage typed by a definition inherits the definition's type and adds contextual information

(multiplicity, constraints, local features). Think of a usage as a "slot" in a structure that expects something of a particular type.

The KerML specification formalizes this distinction through the metamodel: Definition and Usage both specialize Classifier, but they have different semantics. A Definition classifies individual things; a Usage classifies tuples or relationships between things. This technical distinction underlies the practical difference: definitions describe what exists, usages describe how things relate.

Features: Attributes, Parts, References, and Parameters

Features are the fundamental mechanism in KerML for describing properties and relationships of types. Every meaningful characteristic of a type, whether it is a data value, a structural component, an interface port, or a behavioral capability, is represented as a feature.

In KerML's metamodel, Feature is a direct specialization of Type. This means features themselves are types with their own characteristics: they have names, multiplicities, directions, and can be typed by other features. This recursive structure enables powerful composition patterns.

SysML v2 defines several kinds of features specialized from KerML's base Feature concept:

Attribute features represent data values or properties. An attribute has a type that specifies what kind of values it can hold. Attributes are the primary mechanism for specifying quantitative characteristics:

```
part def Sensor {
  attribute accuracy : Real;
  attribute samplingRate : FrequencyValue;
  attribute calibratedAt : TimeInstant;
}
```

Part features represent structural components or subassemblies. A part feature indicates that the owning type is composed of parts of some other type. Parts are the primary mechanism for structural decomposition:

```
part def Aircraft {
  part fuselage : Fuselage;
  part wings : Wing[2];
  part engines : JetEngine[2..4];
}
```

The multiplicity `[2]` on wings means exactly two wing parts; `[2..4]` on engines means between two and four engines. Parts can be composite (the part cannot exist independently of its whole) or referential (the part can exist independently and is merely referenced).

Port features represent interaction points through which the owning type communicates with its environment or other types. Ports have directed features indicating the direction of exchange:

```
part def Battery {
  port out power : ElectricalPowerPort;
  port in charge : ChargingPort;
}
```

Reference features represent relationships to other elements that are not structural composition. A reference indicates that the owning type depends on or is associated with something of a particular type without containing it:

```
part def SoftwareModule {
  ref part hardwarePlatform : ComputePlatform;
  ref part dependencies : SoftwareModule[0..*];
}
```

The `ref` keyword distinguishes references from composite parts. The hardwarePlatform reference says this software module runs on some compute platform but does not own or contain that platform.

Parameter features are used in actions, calculations, and constraints to represent inputs and outputs of behaviors. Parameters have directions: `in` for inputs, `out` for outputs, `inout` for bidirectional:

```
action def CalculateThrust {
  in massFlowRate : MassFlowRateValue;
  in exhaustVelocity : SpeedValue;
  out thrust : ForceValue;
}
```

All features share common characteristics defined by KerML. They have a name (used to identify them within their owner), a type (specifying what kind of values they can hold), a multiplicity (specifying how many values), and a direction (for ports and parameters). Features can be typed by other features, enabling feature inheritance and specialization.

Feature chaining is an important KerML capability: dot notation navigates nested features to create derived features. If part A has a part feature `engine` of type Engine, and Engine has an attribute feature `power`, then the chained feature `A.engine.power` is a derived attribute representing the power of A's engine. This capability enables concise expression of complex relationships:

```
part def Vehicle {
  // Direct access to nested features via chaining
  attribute maxSpeed subsets propulsion.engine.maxOutputSpeed;
}
```

Multiplicities and Cardinality Semantics

Multiplicities specify how many values a feature can hold. They are fundamental to expressing quantitative constraints in models: exactly one engine, zero or more sensors, between two and four wheels.

KerML defines multiplicities as intervals over natural numbers. The general form is `[lower..upper]` where lower and upper are non-negative integers or * (representing unbounded). Common forms include:

- `[1]` or omitted: exactly one value (SysML v2 default for parts, attributes, ports)
- `[0..1]`: zero or one value (optional)
- `[0..*]`: any number of values, including zero (KerML default)
- `[1..*]`: at least one value
- `[n]`: exactly n values

- `[m..n]`: between m and n values inclusive

SysML v2 modifies KerML's defaults for practical engineering use. While KerML defaults to `[0..*]` for all features, SysML v2 defaults to `[1..1]` for parts, attributes, and ports, reflecting the common engineering expectation that a named feature represents exactly one thing unless explicitly stated otherwise.

```
part def Robot {
  part chassis : Chassis;           // [1] by default
  part arm : RoboticArm[2];        // exactly 2 arms
  part sensor : Sensor[0..*];      // any number of sensors
  part gripper : Gripper[0..1];    // optional gripper
}
```

Multiplicities interact with feature typing and specialization. When a subtype specializes a supertype, the subtype's multiplicities must be compatible with the supertype's multiplicities. Generally, this means the subtype can narrow but not widen the multiplicity interval: if a supertype specifies `[2..4]` engines, a subtype can specify `[2..3]` or `[3..3]` but not `[1..5]`.

Collection semantics add additional constraints beyond simple cardinality. KerML supports two collection modifiers:

- `ordered`: the values form a sequence with positional significance; elements are indexed
- `nonunique`: duplicate values are allowed

By default, collections are unordered and unique (no duplicates). These modifiers affect how features can be accessed and manipulated:

```
part def TaskScheduler {
  ordered part taskQueue : Task[0..*];    // FIFO queue, order matters
  nonunique part logEntries : LogEntry[0..*];  // duplicates allowed
}
```

An ordered collection supports indexing: `taskQueue#(1)` accesses the first element, `taskQueue->size()` returns the count. These operations are defined by KerML's expression language and enable precise specification of data access patterns.

Multiplicities are not merely documentation; they are enforceable constraints. A conformant SysML v2 tool should validate that model elements satisfy their declared multiplicities. If a part definition specifies `part wheels : Wheel[4]`, any usage of that part must have exactly four wheel parts. Tools can report multiplicity violations during model validation, catching design errors early.

Typing Rules and Type Conformance

Typing is the mechanism by which KerML and SysML v2 ensure that features hold values of appropriate kinds. Every feature has a type, either explicitly declared or implicitly derived. The type constrains what values the feature can take and ensures consistency across the model.

The basic typing rule is simple: if a feature `f` is typed by type `T`, then every value of `f` must be an instance of `T` or a subtype of `T`. This rule supports polymorphism: a feature typed by Vehicle accepts values that are Vehicles or any specialized kind of vehicle (Car, Truck, Motorcycle).

```
part def Garage {
  part vehicles : Vehicle[0..*];  // accepts any Vehicle or subtype
}

part def Car :> Vehicle;
part def Truck :> Vehicle;

// Both valid:
part myGarage : Garage {
  part vehicles#(1) : Car;
  part vehicles#(2) : Truck;
}
```

Type conformance is the formal relationship that determines whether one type is acceptable where another is expected. KerML defines precise rules for type conformance:

1. A type conforms to itself.
2. A subtype conforms to its supertype (substitutability).
3. For features, the conforming feature's type must conform to the original feature's type, and multiplicities must be compatible.

4. For behaviors, parameters and return types must conform with appropriate directionality.

These rules enable powerful reuse patterns while preventing invalid assignments. If a function expects a parameter of type LengthValue, you can pass any length value (meters, feet, kilometers) but not a MassValue or SpeedValue. The type system catches category errors that would otherwise require runtime checking.

Feature typing uses the colon syntax: `featureName : FeatureType`. This declares both the feature and its type simultaneously:

```
part def Thermometer {
  attribute temperature : TemperatureValue;
  part sensor : TemperatureSensor;
}
```

Here, `temperature` is an attribute typed by `TemperatureValue`, and `sensor` is a part typed by `TemperatureSensor`. The types constrain what values these features can hold.

KerML also supports feature typing between features themselves (not just definitions). This enables patterns where one usage constrains another:

```
part def Vehicle {
  attribute length : LengthValue;
}

part def CompactCar :> Vehicle {
  attribute length subsets Vehicle::length;  // typed by parent's length
}
```

The `subsets` relationship here implies that CompactCar's length is typed by the same type as Vehicle's length (LengthValue), ensuring consistency.

Type errors are among the most common modeling mistakes. KerML's formal typing rules enable tools to detect these errors automatically: assigning a MassValue where a LengthValue is expected, using a part without declaring its type, or violating multiplicity constraints. These checks should run continuously during model development, providing immediate feedback.

Memberships and Containment Hierarchies

Memberships define the organizational structure of models by specifying which elements contain which other elements. Every element in a KerML-based model (except the root namespace) is contained within some enclosing namespace through a membership relationship.

The basic membership kind is FeatureMembership, which adds a feature to its owner's set of features. When you write `attribute mass : MassValue` inside a part definition, you are creating a FeatureMembership that links the attribute to the part.

```
part def Engine {
  // Each line creates a FeatureMembership:
  attribute displacement : VolumeValue;
  port fuelIn : FuelPort;
}
```

Other membership kinds serve specialized purposes:

- EndFeatureMembership: used for association ends (relational features between types)
- Generalization: expresses specialization relationships (subtype to supertype)
- Subsetting: expresses subsetting relationships between features
- Redefinition: expresses redefinition relationships between features
- Import: brings elements from one namespace into another's scope
- Export: controls visibility of imported elements

Memberships establish containment hierarchies that determine scoping, ownership, and navigation. An element's qualified name includes its enclosing namespaces: `Vehicle::Engine::displacement`. This hierarchical naming prevents conflicts and enables precise identification.

Ownership semantics are important: when an owner is deleted, its owned members are typically deleted as well (unless they are shared or imported). This cascading deletion simplifies model management but requires care when restructuring large models.

The containment hierarchy also determines visibility. By default, members are visible within their enclosing namespace and sub-namespaces. Visibility modifiers (`public`, `protected`, `private`) control access:

- `public`: visible everywhere (default in SysML v2)
- `protected`: visible within the defining namespace and its specializations
- `private`: visible only within the defining namespace

```
package InternalDesign {
  part def ProprietaryComponent {
    public attribute interfaceSpec : InterfaceDefinition;
    protected attribute designMargin : Real;
    private attribute costEstimate : CurrencyValue;
  }
}
```

This visibility control enables information hiding: implementation details can be concealed while exposing necessary interfaces, supporting modular design and encapsulation.

Understanding memberships is essential for navigating and querying models programmatically. The SysML v2 API uses membership relationships to traverse the model structure: finding all features of a definition, locating an element by qualified name, or identifying which package owns a particular element.

KerML Semantics: Declarative Foundations and Executability

Understanding KerML's semantics is essential for grasping how SysML v2 models can be analyzed, validated, and executed. KerML's semantic design represents a deliberate choice that distinguishes it from previous modeling languages.

Declarative versus operational semantics. KerML uses declarative (also called denotational or ontological) semantics rather than operational semantics. This means KerML specifies what a model means, not how to compute it step by step. A KerML model defines relationships between things in the universe of discourse: types classify sets of instances, features define mappings from instances to values, constraints specify conditions that must hold. The semantics are grounded in first-order logic and set theory.

This contrasts with operational semantics, which defines a precise execution procedure: given a model and an initial state, what is the next state, and

the next after that? Languages like fUML (Foundational UML) use operational semantics to enable direct execution of models as state machines. KerML deliberately chose not to embed operational semantics in the core specification.

Why declarative semantics matter. The declarative approach provides several advantages:

First, it separates meaning from implementation. A KerML model's semantics are defined independently of any particular tool or execution engine. Different tools can implement different operational strategies while preserving the same underlying meaning. This supports tool interoperability and prevents vendor lock-in at the semantic level.

Second, it enables formal reasoning. Because KerML semantics map to well-understood mathematical concepts, properties of models can be proven using theorem provers, model checkers, and other formal methods tools. Consistency checking, conformance verification, and equivalence analysis become mathematically grounded rather than heuristic.

Third, it supports multiple views of the same model. A single KerML model can be interpreted as a structural description, a constraint satisfaction problem, a simulation specification, or a documentation source without changing the underlying semantics. Different stakeholders extract different value from the same declarative representation.

Generating execution traces. While KerML itself is declarative, it is often necessary to operationally execute models: to simulate behavior, explore design alternatives, or verify dynamic properties. KerML supports this through the concept of execution traces.

An execution trace is a time-ordered sequence of model states and events that is consistent with the model's declarative semantics. The trace does not define the semantics; rather, it must conform to them. Multiple valid execution traces may exist for the same model, reflecting nondeterminism or alternative behaviors permitted by the specification.

To generate execution traces from a KerML-based model, an execution engine interprets the declarative constraints and produces operational behavior that satisfies them. For example:

A state machine's transitions define which state changes are permitted (declarative). An execution engine selects among enabled transitions according to scheduling rules and produces a trace of state changes over time

(operational).

A constraint network specifies relationships between variables (declarative). A solver finds values that satisfy all constraints simultaneously, producing a solution trace (operational).

An action model defines ordering and data dependencies between actions (declarative). An execution engine schedules actions respecting these dependencies and produces an execution trace (operational).

This separation means that KerML models can be executed by different engines with different scheduling policies, concurrency models, or optimization strategies while remaining semantically faithful to the original specification.

Discrete event simulation foundations. SysML v2's integration of state structure modeling with declarative process modeling provides a basis for discrete event simulation (DES). In DES, system behavior is modeled as a sequence of events occurring at discrete points in time, with the system state changing only at event instants.

SysML v2 supports DES through:

State machines that define the system's mode structure and event-driven transitions.

Actions and successions that specify procedural behaviors triggered by events.

Constraints that enforce physical or logical relationships between quantities.

Timed events and duration constraints that govern when events occur.

A SysML v2 model suitable for DES execution typically includes:

An event queue mechanism (explicitly modeled or provided by the execution engine) that orders pending events by time.

State machines governing each component's mode-dependent behavior.

Actions representing processing, computation, or physical operations with specified durations.

Constraints ensuring consistency between state variables and physical quantities.

The execution proceeds by: selecting the next event from the queue, updating relevant states and attributes, evaluating constraints, generating

new events based on transitions and actions, and repeating until termination conditions are met.

This approach enables engineers to simulate system behavior quantitatively: exploring how a manufacturing line responds to demand variation, how a communication protocol handles congestion, or how a spacecraft's power system manages load during eclipse periods. The simulation is grounded in the same declarative model used for design, ensuring consistency between specification and analysis.

Continuous dynamics and hybrid systems. Many engineering systems exhibit continuous behavior: fluid levels changing over time, temperatures evolving according to heat transfer equations, vehicles moving along trajectories governed by differential equations. KerML's core semantics are discrete; continuous dynamics require extension.

SysML v2 supports continuous and hybrid (mixed discrete-continuous) modeling through:

Mathematical expressions in constraints that encode differential or algebraic equations.

Integration with external simulation tools via FMI/FMU (Functional Mock-up Interface/Unit) for detailed continuous-domain analysis.

Co-simulation frameworks where discrete SysML v2 events interact with continuous solvers.

For example, a thermal management model might use SysML v2 state machines for fan control modes (discrete: on/off/auto) while encoding heat transfer equations as constraints solved by an external numerical solver (continuous). The two domains interact: the discrete mode changes affect the continuous heat flow equations, and continuous temperature readings trigger discrete mode transitions.

This hybrid approach enables modeling of cyber-physical systems where digital controllers manage physical processes governed by continuous dynamics.

Executable Semantics: How KerML and SysML v2 Models Run

SysML v2 is not merely a descriptive notation; it defines executable semantics that specify how models behave when run. Understanding these semantics is essential for engineers who want to simulate their models, generate code from them, or use them as the basis for formal verification. This section provides a technically rigorous explanation of KerML's execution model.

The Occurrence Model: 4D Spatiotemporal Semantics

At the foundation of KerML's executable semantics is the concept of occurrence. An occurrence is an entity that exists in both space and time: it has spatial extent (it is something, with structure) and temporal extent (it persists over a duration). This "four-dimensionalist" view unifies structure and behavior under a single semantic framework.

In KerML's semantics, every part usage or action performance is an occurrence. A part occurrence represents a physical or logical entity that exists during some time interval. An action occurrence represents a behavioral execution that occurs during some time interval. Both have identity, both have temporal extent, and both can be related to each other through features and relationships.

The 4D semantics distinguishes between:

Individuals: entities with persistent identity over time (a specific engine in a specific vehicle). Individuals are modeled as occurrences with continuous temporal extent.

Snapshots: the state of an individual at a particular instant. A snapshot captures the values of all features at a point in time without implying persistence.

Performances: behavioral occurrences that happen during a time interval (an action execution, a state machine transition). Performances are occurrences whose primary characteristic is their temporal extent.

This distinction resolves ambiguities present in earlier modeling languages where the same notation was used for types, instances, and behaviors. In KerML, everything that exists or happens is an occurrence with well-defined spatiotemporal properties.

Discrete-Event Execution Model

For discrete behavior (state machines, actions, interactions), KerML defines a discrete-event execution model. The core concepts are:

Time model. Time is modeled as a linearly ordered set of instants. Events occur at specific time instants. Durations are intervals between instants. The time model supports both absolute time (specific instants on a timeline) and relative time (durations measured from reference points).

Event queue. An event queue holds pending events ordered by their scheduled occurrence time. When the simulation clock advances to an event's time, the event is dequeued and processed. Events include:

- Signal events: explicit messages or triggers sent between components
- Value change events: notifications that a monitored attribute has changed value
- Time events: triggers that fire after a specified duration or at an absolute time

Event processing follows a deterministic order:

1. Advance the simulation clock to the next event's time.
2. Dequeue all events scheduled for that time instant.
3. Process events in a defined priority order (typically: higher-priority events first, then by insertion order).
4. Each event may trigger state transitions, action executions, or new event scheduling.
5. Repeat until the queue is empty or a termination condition is met.

State machine execution. When a state machine receives an event, it evaluates all enabled transitions (transitions whose source state is active and whose guard conditions are true). If exactly one transition is enabled, it fires: exit actions of the source state execute, the transition effect executes, and entry actions of the target state execute. If multiple transitions are enabled, conflict resolution rules apply (more specific transitions take precedence; inner-state transitions take precedence over outer-state transitions).

If no transition is enabled for an event, the event is ignored (the state machine remains in its current state). This semantics ensures that state machines are robust to unexpected events.

Orthogonal region execution. In states with orthogonal regions, each region maintains its own active substate independently. An event may trigger transitions in one region without affecting others, unless explicitly designed to do so. The combined state of a composite state with orthogonal regions is the tuple of active substates across all regions.

Action network execution. Actions execute according to their control flow structure (successions, decisions, forks, joins). The execution model ensures:

- An action starts only when all its input data is available (all incoming object flows have delivered values).
- Control flow successions are respected: a successor action starts only after its predecessor completes.
- Fork nodes spawn concurrent executions of all outgoing branches.
- Join nodes wait for all incoming branches to complete before enabling the successor.
- Decision nodes evaluate guards and enable exactly one outgoing branch (or none if no guard is true).

Object flow semantics. Object flows transfer values between actions. When a source action produces an output value, that value becomes available on the connected object flow. A target action that consumes from that flow can read the value when it executes. For collection flows, individual elements may be processed independently or as a batch, depending on the flow's multiplicity and the consumer's expectations.

Continuous Dynamics Handling

For continuous behavior (physical processes governed by differential equations, gradual changes over time), KerML and SysML v2 support modeling through constraint networks and integration with continuous simulation tools. The language itself does not define a native continuous solver; instead, it provides constructs for specifying continuous relationships that can be evaluated by external solvers.

Continuous variables. Attributes typed by quantity value types (LengthValue, TemperatureValue, etc.) can represent continuously varying quantities. Their values change smoothly over time rather than jumping discretely at event instants.

Differential constraints. Constraints can express relationships between continuous variables and their derivatives:

```
constraint def ThermalDynamics {
  parameter temperature : TemperatureValue;
  parameter heatInput : PowerValue;
  parameter heatLoss : PowerValue;
  parameter thermalCapacity : EnergyPerTemperatureValue;

  // dT/dt = (heatIn - heatOut) / C
  assert derivative(temperature) = (heatInput - heatLoss) / thermalCapacity;
}
```

This constraint specifies how temperature changes over time based on heat flow. A continuous solver evaluates this differential equation to compute temperature as a function of time.

Hybrid systems. Real-world systems often combine discrete and continuous behavior: a thermostat (discrete state machine) controls heating (continuous thermal dynamics). SysML v2 models such hybrid systems by integrating discrete-event execution with continuous constraint evaluation:

1. The continuous solver advances the simulation in small time steps, evaluating differential constraints.
2. At each step, the system checks for events (value changes crossing thresholds, timer expirations).
3. When an event occurs, the discrete-event engine processes it (state transitions, action executions).
4. Discrete behavior may modify parameters that affect continuous dynamics (turning a heater on or off).
5. The continuous solver resumes with updated parameters.

This hybrid execution model enables accurate simulation of cyber-physical systems where digital controllers interact with physical processes.

Scheduling and Concurrency Models

SysML v2's execution model supports multiple concurrency paradigms:

Interleaved concurrency. Concurrent actions or state machines are simulated by interleaving their executions: at each step, one concurrent element

advances while others wait. This model is simple and deterministic but may not reflect real parallelism accurately.

True parallel execution. In tools that support multi-threaded simulation, concurrent elements execute in parallel on separate threads. This model better reflects real-time systems where components truly operate simultaneously.

Real-time scheduling. For models intended to run on actual hardware, SysML v2 supports specifying scheduling policies:

- Rate-monotonic scheduling: periodic tasks are prioritized by frequency (higher frequency = higher priority)
- Earliest-deadline-first scheduling: tasks are prioritized by deadline urgency
- Fixed-priority preemptive scheduling: tasks have static priorities; higher-priority tasks can preempt lower-priority ones

These scheduling policies enable analysis of whether a system can meet its timing requirements under worst-case conditions.

Operational Semantics Summary

The operational semantics of KerML and SysML v2 can be summarized as follows:

1. Models are interpreted as sets of occurrences with spatiotemporal properties.
2. Discrete behavior executes via discrete-event simulation with deterministic event processing.
3. Continuous behavior is specified through constraints evaluated by external solvers.
4. Hybrid systems integrate discrete and continuous execution through event-driven coupling.
5. Concurrency is supported through interleaved or parallel execution models.
6. Scheduling policies determine the order of concurrent executions in real-time contexts.

These semantics are declarative: they specify what behavior is correct without prescribing how to implement it. Different tools may use different execution strategies (direct interpretation, code generation, co-simulation) while producing behavior consistent with the model's semantics.

Executable Semantics in Practice

Tool vendors implement executable semantics for SysML v2 models through various strategies:

Direct execution engines interpret the model's behavioral elements (state machines, actions) and execute them according to defined scheduling policies. These engines may be built into modeling tools or available as standalone simulators. For example, the Syside Automator tool can extract state machine definitions from a SysML v2 model and simulate them using the Sismic Python-based simulation engine, validating transition logic before implementation.

Code generation transforms SysML v2 models into executable code in languages like C++, Python, or Modelica. The generated code embodies the model's semantics and can be compiled and run independently. Sinelabore's code generator, for instance, produces C++ code from SysML v2 textual models, generating classes for parts, structs for actions, and state machine implementations. This approach is common for embedded systems where the model serves as both specification and implementation template.

Co-simulation integration connects SysML v2 models with external simulation tools via standard interfaces (FMI/FMU, APIs). The SysML v2 model handles discrete-event logic and coordination while specialized tools handle domain-specific continuous dynamics. For example, a vehicle model might use SysML v2 for control logic and state management while delegating powertrain physics to a Modelica solver and aerodynamics to a CFD simulation.

Constraint solving evaluates constraint networks using mathematical programming solvers (linear programming, mixed-integer programming, constraint satisfaction) to find valid configurations or verify that requirements are satisfiable. Tools like Maple or specialized constraint solvers can analyze SysML v2 parametric models to check feasibility, optimize designs, or perform sensitivity analysis.

The choice of execution strategy depends on the modeling purpose: simulation for behavioral exploration, code generation for implementation, co-simulation for multi-domain analysis, or constraint solving for design optimization. SysML v2's declarative semantics ensure that all these strategies produce behavior consistent with the model's intended meaning.

KerML Textual Syntax Fundamentals

KerML's textual syntax provides the foundation for SysML v2's text-based modeling. While KerML itself is rarely used directly (engineers typically work with SysML v2), understanding its syntax patterns helps explain why SysML v2 is structured the way it is.

The basic structure of a KerML/SysML v2 text file is a sequence of package members enclosed in braces:

```
package MyModel {
  // Package members: definitions, usages, imports, etc.
}
```

Namespaces (including packages) can be nested, creating hierarchical organization:

```
package TopLevel {
  package SubLevel {
    part def Component {
      attribute value : Real;
    }
  }
}
```

Qualified names use double colons to navigate the hierarchy: `TopLevel::SubLevel::Component`.

Comments are supported in two forms: line comments starting with `//` and block comments enclosed in `/* */`:

```
// This is a line comment
part def Sensor {
  /* Block comment
     spanning multiple lines */
  attribute accuracy : Real;
}
```

Names follow standard identifier rules: they begin with a letter or underscore and contain letters, digits, and underscores. Keywords are reserved and cannot be used as names. Important KerML/SysML v2 keywords include:

`package`, `part`, `def`, `attribute`, `port`, `ref`, `action`, `state`, `transition`, `requirement`, `constraint`, `import`, `export`, `specializes`, `subsets`, `redefines`, `connect`, `in`, `out`, `inout`, `ordered`, `nonunique`, `public`, `protected`, `private`, `variation`, `variant`.

The textual syntax is designed for readability and precision. Indentation (typically two or four spaces) indicates nesting but is not semantically significant; braces define structure. Line breaks are generally optional except within certain constructs. This flexibility enables different formatting styles while maintaining semantic consistency.

KerML's expression language supports mathematical operations, logical operators, and feature navigation within attribute values and constraints:

```
part def Rectangle {
  attribute width : LengthValue;
  attribute height : LengthValue;
  attribute area = width * height;  // calculated attribute
}
```

The expression `width * height` multiplies the two length attributes to compute the area. KerML's dimensional analysis ensures that multiplying two lengths produces an area value, not some other quantity kind.

Mastering KerML fundamentals provides the foundation for everything in SysML v2. The next chapters apply these concepts to specific modeling tasks: organizing models with packages and imports, applying specialization and subsetting, building structural models with parts and ports, specifying requirements, modeling behavior, and performing analysis. But all of these capabilities trace back to the core KerML concepts covered in this chapter.

Chapter 3: Organization: Namespaces, Packages, Imports, and Visibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Namespaces as the Organizing Primitive

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Packages: Structure, Hierarchy, and Purpose

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Imports: Selective, Global, and Hidden

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Exports and Controlled Visibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Membership Kinds and Ownership

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Large-Scale Model Organization Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Common Pitfalls in Package Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Chapter 4: Specialization, Subsetting, and Redefinition: The Polymorphism System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Specialization: Generalization and Inheritance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Subsetting: Constraining Without Redefining

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Redefinition: Overriding Feature Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Comparison Table: When to Use Each Relationship

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Conformance Rules and Consistency Constraints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Polymorphism in Structural and Behavioral Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Anti-Patterns: Misusing Specialization, Subsetting, Redefinition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Chapter 5: Structural Modeling: Parts, Ports, Interfaces, and Connectors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Part Definitions and Part Usages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Graphical Notation for Parts and Structural Elements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Port Definitions: Provisioning and Requiring Capabilities

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Graphical Notation for Ports and Interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Interface Definitions and Their Contracts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Connectors: Binding Ports and Establishing Relationships

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Graphical Notation for Connectors and Connections

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Flow Ports and Item Flows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Structural Decomposition Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Case Study Fragment: Autonomous Vehicle Sensor Suite Architecture

Chapter 6: Value Semantics: Types, Units, Quantities, and Dimensional Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Value Types: Primitives and Composites

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Quantity Kinds and Physical Dimensions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Unit Definitions and Conversions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Dimensional Analysis and Consistency Checking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Typed Attributes and Engineering Parameters

Case Study Fragment: Spacecraft Propulsion System Sizing

Extending the Spacecraft Case: Requirements, Behavior, and Verification

Chapter 7: Requirements Engineering: From Needs to Verifiable Specifications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Requirement Definitions vs. Requirements in Practice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Stakeholders, Concerns, and Objectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Assumptions and Constraints on the Environment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Verification Cases and Test Specifications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Traceability: Satisfy, Derive, Refine Relationships

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Bidirectional Traceability to Architecture and Behavior

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Case Study Fragment: Medical Device Safety Requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Chapter 8: State Machines: Event-Based Behavior and Mode Modeling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

State Definitions and State Usages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Events: Triggers, Guards, and Effects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Transitions and Transition Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Hierarchical States and Nesting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Orthogonal Regions and Concurrent Behavior

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

History States and Deep History

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Case Study Fragment: Avionics Flight Control Modes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Graphical Notation for State Machines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Chapter 9: Action Models: Control Flow, Object Flow, and Calculations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Action Definitions and Action Usages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Control Flow and Execution Ordering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Object Flow and Data Passing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Decisions, Merges, Forks, and Joins

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Calculations and Mathematical Expressions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Iteration and Looping Constructs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Execution Semantics of Action Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Graphical Notation for Action Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Case Study Fragment: Industrial Automation Control Logic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Chapter 10: Interactions: Sequences, Messages, and Protocol Modeling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Interaction Definitions and Message Exchanges

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Lifelines and Participant Roles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Sequential Messages and Asynchronous Communication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Combined Fragments: Alt, Opt, Loop, Par

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Timing Constraints and Durations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Protocol Verification via Interactions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Case Study Fragment: Telecommunications Protocol Specification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Graphical Notation for Interactions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Chapter 11: Parametric Modeling: Constraints, Analysis, and Trade Studies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Constraint Definitions and Mathematical Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Probes: Extracting Values from the Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Building Constraint Networks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Engineering Equations and Physical Laws

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Trade Studies and Design Space Exploration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Integration with Analysis Tools and Simulation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Case Study Fragment: Robotics Payload Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Chapter 12: Allocations, Dependencies, and Traceability: Connecting the Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Allocation Relationships and Their Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Allocating Requirements to Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Logical-to-Physical Allocation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Dependency Types: Usage, Trace, Realization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Model-Wide Traceability Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Consistency Checking Across Allocations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Case Study Fragment: End-to-End Avionics System Traceability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Chapter 13: Views, Viewpoints, and Stakeholder Concerns: Managing Model Complexity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Viewpoints: Defining Modeling Perspectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Views: Materializing Stakeholder Perspectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Concerns and Cross-Cutting Properties

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Tailoring Models for Different Audiences

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Managing Model Complexity at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Case Study Fragment: Multi-Stakeholder Consumer Electronics Project

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Chapter 14: Advanced Patterns: Variability, Product Lines, Libraries, and Reuse

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Variability Modeling with Optional Features

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Product-Line Engineering Concepts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Model Libraries and Reusable Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Metadata Annotations and Extensibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Configuration Management in SysML v2 Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Industrial-Scale Reuse Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Case Study Fragment: Automotive Platform Family

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Chapter 15: Tooling, Ecosystem, and Digital Engineering Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

The SysML v2 Tool Landscape

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Model Interchange: XMI, JSON, and Textual Formats

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Repository-Based Modeling and Collaboration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Versioning and Configuration Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Integration with PLM, ALM, and Digital Thread Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

API Concepts for Tool Development

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Model Transformations: From SysML v2 Models to Code, Tests, and Other Artifacts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

The Future of SysML v2 Adoption

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Chapter 16: Migration from SysML v1.x to SysML v2: A Practical Guide

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Architectural Differences Between SysML v1.x and v2

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Mapping Block Definition Diagrams to KerML-Based Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Internal Block Diagrams to Structural Decomposition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Activity Diagrams to Action Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

State Machine Evolution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Sequence Diagrams to Interactions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Parametric Diagrams to Constraint Networks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Requirements Diagram Transformation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Migration Strategies and Tool Support

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Chapter 17: Complete Case Study: Autonomous Vehicle System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Stakeholder Needs and Problem Framing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Requirements Model: Safety, Performance, Regulations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Logical Architecture: Functional Decomposition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Physical Architecture: Hardware Components and Sensors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Interface Specifications Between Subsystems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Behavioral Models: State Machines for Driving Modes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Analysis Models: Sensor Fusion Accuracy Constraints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Verification Cases and Traceability Matrix

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Lessons Learned and Modeling Decisions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Chapter 18: Conclusion: The Future of Model-Based Systems Engineering with SysML v2

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

What Makes SysML v2 Different: A Retrospective

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

The Role of Models in Modern Engineering Organizations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Emerging Trends: AI, Simulation, Digital Twins

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Open Challenges and Research Directions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Advice for Practitioners Starting Their MBSE Journey

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

Final Thoughts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sysmlv2thedefinitiveguidetomodel-basedsystemsengineering.