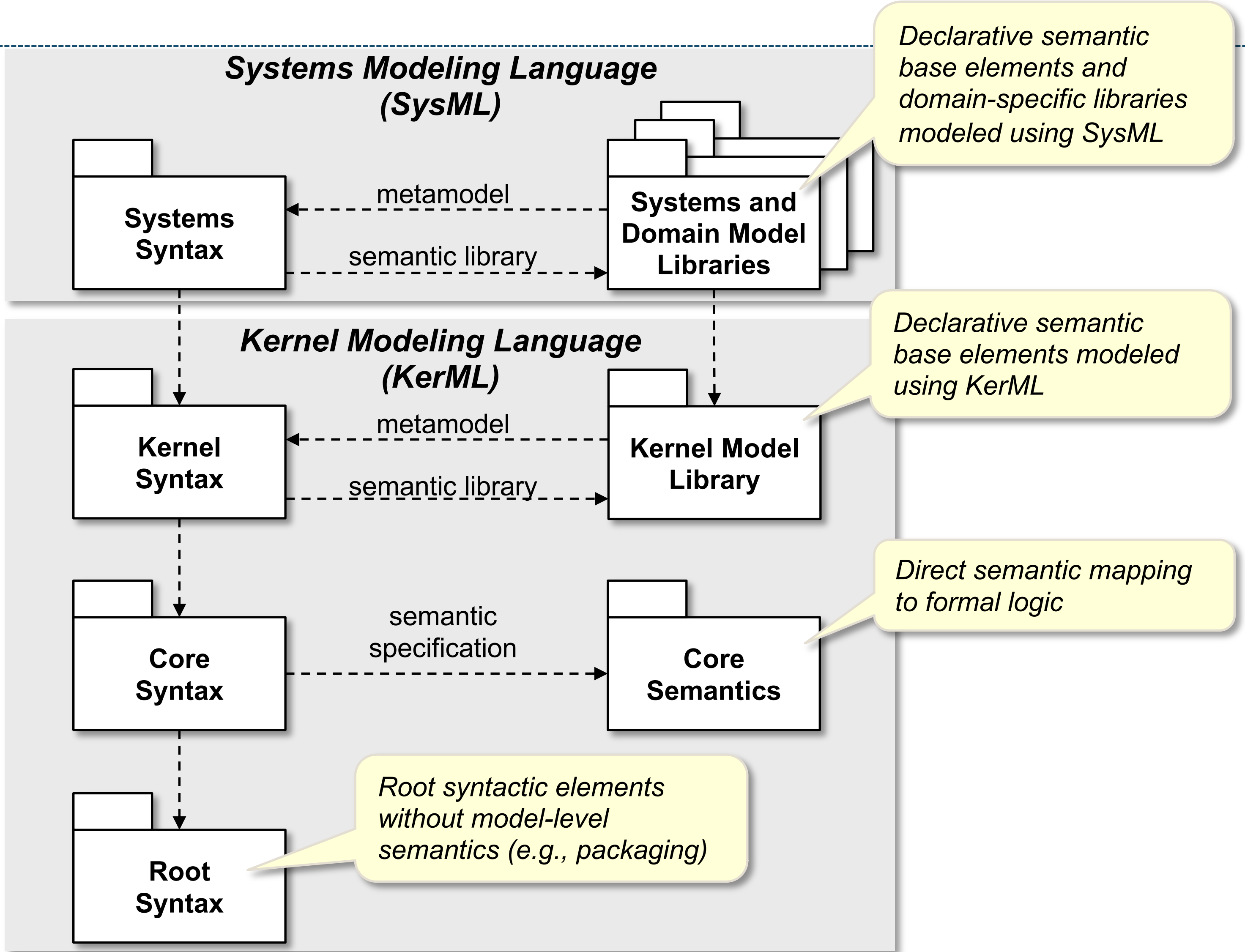
Kernel Modeling Language (KerML)

for Building Modeling Languages

Jet Propulsion Laboratory
1 February 2024

Ed Seidewitz
Model Driven Solutions
ed-s@modeldriven.com

SysML v2 Language Architecture



KerML Core
Key Semantic Concepts

- **Type** – classifies a set of instances
  - **Classifier** – a type that classifies a subset of the things in the universe of discourse
  - **Feature** – a type that classifies pairs of things classified by a domain (featuring) type and a co-domain (featured) type
- **Specialization** – relates a subtype that classifies a subset of the instances of a supertype
  - **Subclassification** – specialization between two classifiers
  - **Subsetting** – specialization between two features
    - **Redefinition** – subsetting that redefines a feature in a specialized context
  - **Feature Typing** – specialization between a feature and its featured type (co-domain)
- **Type Featuring** – relates a feature to its featuring type (domain)

KerML Core
Other Relationships

- Applicable to any kind of type
  - **Disjoining** – a relationship between types asserted to classify disjoint sets of instances
  - **Unioning** – a relationship between a union type and one of the types being unioned
  - **Intersecting** – a relationship between an intersection type and one of the types being intersected
  - **Differencing** – a relationship between a difference type and one of the types being differenced
  - **Conjugation** – a relationship between a conjugated type an an original type such that the conjugated type inherits features from the original type with directions reversed
- Applicable only to features
  - **Feature Inverting** – a relationship between two features asserting they are inverses
  - **Feature Chaining** – a relationship between a chained feature and one of the features in the chain

KerML Core
Basic Syntax

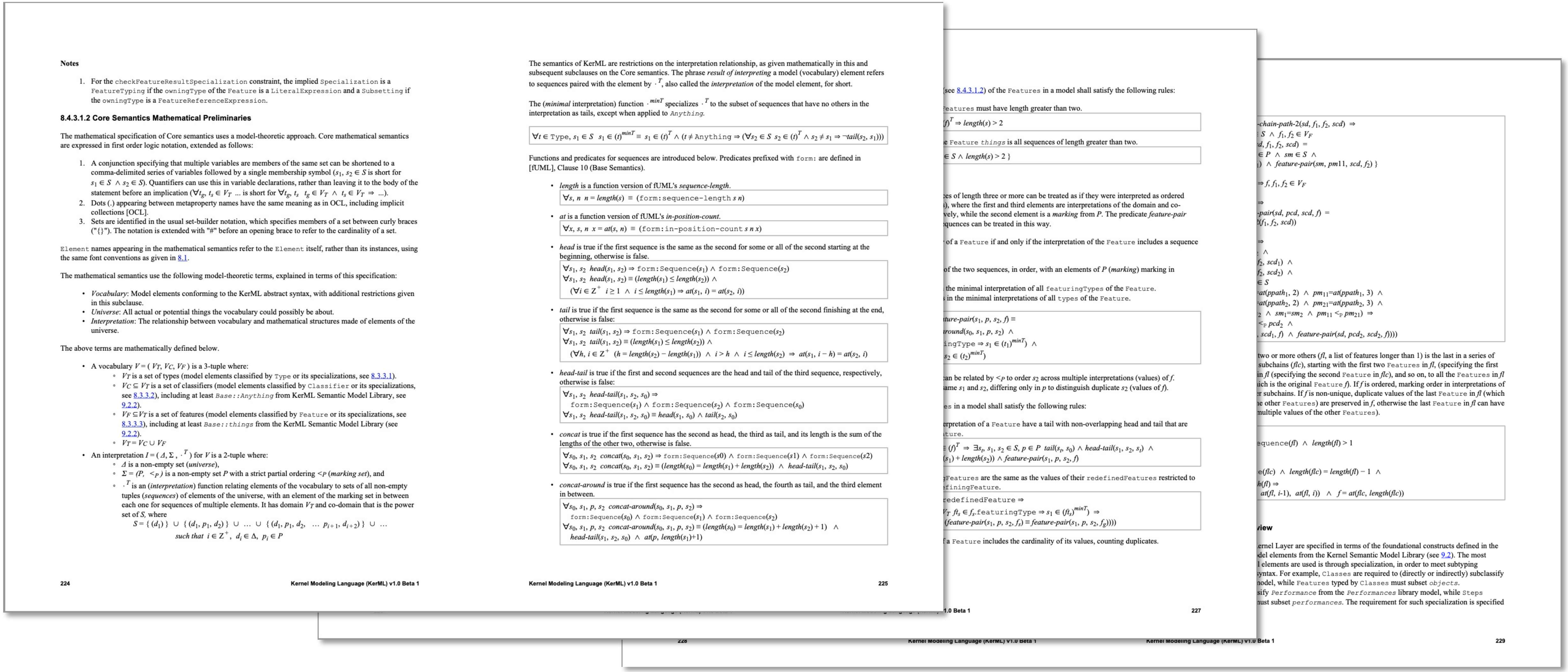
A feature is commonly an owned member of its *featuring type* (in this case Engine).

```
package KerML_Base_Example {
   classifier TorqueValue;

   classifier Person;
   classifier Engine {
      feature engineTorque: TorqueValue[1];
   }
   classifier Wheel;

   classifier Car {
      feature driver: Person[0..1];
      feature engine: Engine[1];
      feature wheels: Wheel[4];
   }
}
```

TorqueValue is the *featured type*.

Multiplicity constrains the allowable cardinality of featured values for each featuring value. (E.g., that 0 or 1 drivers are allowed for each Car.)

KerML Core Mathematical Semantics



Notes

1. For the `checkFeatureResultSpecialization` constraint, the implied `Specialization` is a `FeatureTyping` if the `owningType` of the `Feature` is a `LiteralExpression` and a `Subsetting` if the `owningType` is a `FeatureReferenceExpression`.

8.4.3.1.2 Core Semantics Mathematical Preliminaries

The mathematical specification of Core semantics uses a model-theoretic approach. Core mathematical semantics are expressed in first order logic notation, extended as follows:

1. A conjunction specifying that multiple variables are members of the same set can be shortened to a comma-delimited series of variables followed by a single membership symbol ($s_1, s_2 \in S$ is short for $s_1 \in S \wedge s_2 \in S$). Quantifiers can use this in variable declarations, rather than leaving it to the body of the statement before an implication ($\forall t_g, t_s \in V_T$ ... is short for $\forall t_g, t_s \;\; t_g \in V_T \wedge t_s \in V_T \Rightarrow$ ...).
2. Dots (.) appearing between metaproperty names have the same meaning as in OCL, including implicit collections [OCL].
3. Sets are identified in the usual set-builder notation, which specifies members of a set between curly braces ("{}"). The notation is extended with "#" before an opening brace to refer to the cardinality of a set.

`Element` names appearing in the mathematical semantics refer to the `Element` itself, rather than its instances, using the same font conventions as given in 8.1.

The mathematical semantics use the following model-theoretic terms, explained in terms of this specification:

- *Vocabulary*: Model elements conforming to the KerML abstract syntax, with additional restrictions given in this subclause.
- *Universe*: All actual or potential things the vocabulary could possibly be about.
- *Interpretation*: The relationship between vocabulary and mathematical structures made of elements of the universe.

The above terms are mathematically defined below.

- A vocabulary $V = (V_T, V_C, V_F)$ is a 3-tuple where:
  - V_T is a set of types (model elements classified by `Type` or its specializations, see 8.3.3.1).
  - $V_C \subseteq V_T$ is a set of classifiers (model elements classified by `Classifier` or its specializations, see 8.3.3.2), including at least `Base::Anything` from KerML Semantic Model Library, see 9.2.2).
  - $V_F \subseteq V_T$ is a set of features (model elements classified by `Feature` or its specializations, see 8.3.3.3), including at least `Base::things` from the KerML Semantic Model Library (see 9.2.2).
  - $V_T = V_C \cup V_F$
- An interpretation $I = (\Delta, \Sigma, \cdot^T)$ for V is a 2-tuple where:
  - Δ is a non-empty set (*universe*),
  - $\Sigma = (P, <_P)$ is a non-empty set P with a strict partial ordering $<_P$ (*marking set*), and
  - $\cdot^T$ is an (*interpretation*) function relating elements of the vocabulary to sets of all non-empty tuples (*sequences*) of elements of the universe, with an element of the marking set in between each one for sequences of multiple elements. It has domain V_T and co-domain that is the power set of S, where

$$S = \{(d_1)\} \cup \{(d_1, p_1, d_2)\} \cup \ldots \cup \{(d_1, p_1, d_2, \ldots p_{i+1}, d_{i+2})\} \cup \ldots$$
$$\textit{such that } i \in Z^+,\; d_i \in \Delta,\; p_i \in P$$

224 Kernel Modeling Language (KerML) v1.0 Beta 1

The semantics of KerML are restrictions on the interpretation relationship, as given mathematically in this and subsequent subclauses on the Core semantics. The phrase *result of interpreting* a model (vocabulary) element refers to sequences paired with the element by $\cdot^T$, also called the *interpretation* of the model element, for short.

The (*minimal* interpretation) function $\cdot^{minT}$ specializes $\cdot^T$ to the subset of sequences that have no others in the interpretation as tails, except when applied to `Anything`.

$$\forall t \in \texttt{Type}, s_1 \in S \;\; s_1 \in (t)^{minT} \equiv s_1 \in (t)^T \wedge (t \neq \texttt{Anything} \Rightarrow (\forall s_2 \in S \;\; s_2 \in (t)^T \wedge s_2 \neq s_1 \Rightarrow \neg tail(s_2, s_1)))$$

Functions and predicates for sequences are introduced below. Predicates prefixed with `form:` are defined in [fUML], Clause 10 (Base Semantics).

- *length* is a function version of fUML's *sequence-length*.

$$\forall s, n \;\; n = length(s) \equiv (\texttt{form:sequence-length}\; s\; n)$$

- *at* is a function version of fUML's *in-position-count*.

$$\forall x, s, n \;\; x = at(s, n) \equiv (\texttt{form:in-position-count}\; s\; n\; x)$$

- *head* is true if the first sequence is the same as the second for some or all of the second starting at the beginning, otherwise is false.

$$\forall s_1, s_2 \;\; head(s_1, s_2) \Rightarrow \texttt{form:Sequence}(s_1) \wedge \texttt{form:Sequence}(s_2)$$
$$\forall s_1, s_2 \;\; head(s_1, s_2) \equiv (length(s_1) \leq length(s_2)) \wedge (\forall i \in Z^+ \;\; i \geq 1 \wedge i \leq length(s_1) \Rightarrow at(s_1, i) = at(s_2, i))$$

- *tail* is true if the first sequence is the same as the second for some or all of the second finishing at the end, otherwise is false:

$$\forall s_1, s_2 \;\; tail(s_1, s_2) \Rightarrow \texttt{form:Sequence}(s_1) \wedge \texttt{form:Sequence}(s_2)$$
$$\forall s_1, s_2 \;\; tail(s_1, s_2) \equiv (length(s_1) \leq length(s_2)) \wedge (\forall h, i \in Z^+ \;\; (h = length(s_2) - length(s_1)) \wedge i > h \wedge i \leq length(s_2) \Rightarrow at(s_1, i-h) = at(s_2, i))$$

- *head-tail* is true if the first and second sequences are the head and tail of the third sequence, respectively, otherwise is false:

$$\forall s_1, s_2 \;\; head\text{-}tail(s_1, s_2, s_0) \Rightarrow \texttt{form:Sequence}(s_1) \wedge \texttt{form:Sequence}(s_2) \wedge \texttt{form:Sequence}(s_0)$$
$$\forall s_1, s_2 \;\; head\text{-}tail(s_1, s_2, s_0) \equiv head(s_1, s_0) \wedge tail(s_2, s_0)$$

- *concat* is true if the first sequence has the second as head, the third as tail, and its length is the sum of the lengths of the other two, otherwise is false.

$$\forall s_0, s_1, s_2 \;\; concat(s_0, s_1, s_2) \Rightarrow \texttt{form:Sequence}(s0) \wedge \texttt{form:Sequence}(s1) \wedge \texttt{form:Sequence}(s2)$$
$$\forall s_0, s_1, s_2 \;\; concat(s_0, s_1, s_2) \equiv (length(s_0) = length(s_1) + length(s_2)) \wedge head\text{-}tail(s_1, s_2, s_0)$$

- *concat-around* is true if the first sequence has the second as head, the fourth as tail, and the third element in between.

$$\forall s_0, s_1, p, s_2 \;\; concat\text{-}around(s_0, s_1, p, s_2) \Rightarrow \texttt{form:Sequence}(s_0) \wedge \texttt{form:Sequence}(s_1) \wedge \texttt{form:Sequence}(s_2)$$
$$\forall s_0, s_1, p, s_2 \;\; concat\text{-}around(s_0, s_1, p, s_2) \equiv (length(s_0) = length(s_1) + length(s_2) + 1) \wedge head\text{-}tail(s_1, s_2, s_0) \wedge at(p, length(s_1)+1)$$

Kernel Modeling Language (KerML) v1.0 Beta 1 225

KerML Kernel Layer
Key Semantic Concepts

- **Anything** – all things in the universe of discourse
  - **Data Value** – a thing without individual identity or existence in space or time
  - **Occurrence** – a thing with individual identity, that exist over time, and (possibly) extend across space
    - **Object** – an occurrence that is a structural objects
    - **Performances** – an occurrences that is a performance of behavior
  - **Link** – Relationships between *participant* things
    - **Binary Link** – a link between exactly two participants
    - **Link Object** – a link that is also an object that exists over time (and possibly space)
      - Binary Link Object – a link object between exactly two participants

KerML Kernel Semantic Concept Model (Notional)

This is a simplified model of an "ontology" of the basic kinds of things.

```
classifier Anything;

classifier DataValue specializes Anything;

classifier Occurrence specializes Anything {
   feature suboccurrences : Occurrence[0..*];
}

classifier Object specializes Occurrence {
   feature subobjects : Object subsets suboccurrences;
   feature ownedPerformances : Performance subsets suboccurrences;
   feature enactedPerformance : Performance;
}

classifier Performance specializes Occurrence {
   feature subperformances : Performance subsets suboccurrences;
}

classifier Link specializes Anything {
   feature participant: Anything[0..*] nonunique ordered;
}
classifier BinaryLink specializes Link {
   feature redefines participant: Anything[2] nonunique ordered;
   end feature source: Anything[0..*] nonunique subsets participant;
   end feature target: Anything[0..*] nonunique subsets participant;
}
classifier BinaryLinkObject specializes BinaryLink, Object;
```

A *suboccurrence* is an occurrence that will be destroyed if its featuring occurrence is destroyed.

A *subobject* is a suboccurrence of an object that is also an object.

An *owned performance* is a suboccurrence of an object that is a performance.

An *enacted performance* is a performance that is caused or performed by an object.

A *subperformance* is a suboccurrence of a performance that is also a performance.

End features have special semantics for multiplicity.

KerML Kernel
Applying Semantic Concepts

```
package KerML_Core_Example {
   import Kernel_Library::*;

   classifier TorqueValue specializes DataValue;

   classifier Person specializes Object;
   classifier Engine specializes Object {
      feature engineTorque: TorqueValue[1];
   }
   classifier Wheel specializes Object;

   classifier DriveTrain specializes BinaryLinkObject {
      end drivingEngine: Engine[0..1] redefines source;
      end drivenWheel: Wheel[0..*] redefines target;
   }

   classifier Car specializes Object {
      feature driver: Person[0..1];
      feature engine: Engine[1] subsets suboccurrences;
      feature wheels: Wheel[4] subsets suboccurrences;
      feature drive: DriveTrain subsets suboccurrences {
         end redefines drivingEngine references engine[1];
         end redefines drivenWheel references wheels[2];
      }
   }
}
```

Classifiers in the user model *specialize* concepts from the semantic model library, identify what kind of thing they subclassify.

Redefinition allows otherwise inherited features to instead be further constrained in a specialized type.

Features are types, too, and inherit from their featured types ("typing as specialization").

Referencing is a special kind of subsetting, useful for constraining the links in a "connection".

KerML Kernel
Syntax for Semantic Concepts

A *data type* implies subclassification of the base type DataValue.

A *structure* implies subclassification of the base type Object.

A binary *association structure* implies subclassification of the base type BinaryLinkObject.

A *composite* feature of a structure implies subsetting of the subobject feature of its featuring type.

```
package KerML_Kernel_Example {

   datatype TorqueValue;
   struct Person;
   struct Engine {
      feature engineTorque: TorqueValue[1];
   }
   struct Wheel;

   assoc struct DriveTrain {
      end drivingEngine: Engine[0..1];
      end drivenWheel: Wheel[0..*];
   }

   struct Car {
      feature driver: Person[0..1];
      composite feature engine: Engine[1];
      composite feature wheels: Wheel[4];
      composite connector drive: DriveTrain
         from engine[1] to wheels[2];
   }
}
```

Syntactic keywords act as "markers" for implied specializations of base types from the semantic library, which can be added automatically by tooling as necessary.

End features implicitly redefine the source and target ends from BinaryLinkObject.

A *connector* is a feature that must be typed by an association. The from and to parts are shorthands giving the referenced elements for the *connector ends*.

KerML Kernel
Semantic Library Models

```
standard library package Base {
   abstract classifier Anything {
      feature self: Anything[1] subsets things chains things.that;
   }

   abst
      fe
   }

   abst
      fe
   }

   abst

   abst
…
}
```

```
standard library package Links {
   …
   abstract assoc Link specializes Anything {
      readonly feature participant: Anything[2..*] nonunique ordered;
   }

   assoc
      fea

      rea
      rea
   }

   assoc
      end

      end

   }
   …
}
```

```
standard library package Occurrences {
   …
   abstract class Occurrence specializes Anything disjoint from DataValue {
      feature localClock : Clock[1] default universalClock;

      composite feature suboccurrences: Occurrence[0..*] subsets occurrences {
         feature redefines localClock default (that as Occurrence).localClock;
         …
      }

      feature withoutOccurrences: Occurrence[0..*]
         unions successors, predecessors, outsideOfOccurrences
         inverse of withoutOccurrences;

      feature predecessors: Occurrence[0..*] subsets withoutOccurrences;
      feature successors: Occurrence[0..*] subsets withoutOccurrences
         inverse of predecessors {…}
      …
}
```

Systems Modeling (Some) Key Semantic Concepts

- *Attribute Value* – a value of attributive data on item
- *Item* – an object that is part of, exists in, or flows through a system
  - Part – an item that represents part or all of a system and may perform actions for or within the system
- *Port* – an object that represents a connection point for a part
- *Connection* – a link object between any kind of things
  - *Binary Connection* – a connection between exactly two things
  - *Interface* – a connection between ports
    - Binary Interface – an interface between two ports
- *Action* – a behavior that can be performed by a part

Systems Modeling (in KerML)
Systems Semantic Concept Model (Notional)

Implicitly subclassifies DataValue.

Implicitly subclassifies Object.

A *step* is a feature that is typed by a behavior.

Implicitly subclassifies Performance.

Implicitly subclassifies BinaryLinkObject.

Implicitly subsets subobjects.

Implicitly redefines source and target from BinaryLinkObject.

```
datatype AttributeValue;

struct Item {
   composite feature subitems: Item[0..*];
   composite feature subparts: Part[0..*] subsets subitems;
}

struct Part specializes Item {
   feature ownedPorts: Port[0..*];
   composite step ownedActions: Action subsets ownedPerformances;
   step performedActions: Action subsets enactedPerformances;
}

struct Port {
   feature subports: Port[0..*];
}

behavior Action {
   composite step subactions: Action subsets subperformances;
}

assoc struct BinaryConnection {
   end source;
   end target;
}

assoc struct BinaryInterface specializes BinaryConnection {
   end source : Port;
   end target : Port;
}
```

This is a simplified ontology of basic systems modeling concepts.

Systems Modeling (in KerML)
Applying Systems Semantic Concepts

An out feature is one whose value is produced "inside" its featuring instance but used "outside" it. (And the opposite for an in feature.)

Conjugation is a relationship between types in which in features of the conjugated type become out features of the conjugating type, and vice versa.

Feature chains ("dot notation") allow navigation across of chain of features in which the featured type of each feature is compatible with the featuring type of the next.

```
package KerML_Systems_Example {
   import Systems_Library::*;

   datatype TorqueValue specializes AttributeValue;
   struct DrivePort specializes Port {
      out feature torque: TorqueValue;
   }
   struct DrivenPort conjugates DrivePort;
   struct Person specializes Item;
   struct Engine specializes Part {
      feature enginePort: DrivePort subsets ownedPorts;
   }
   struct Wheel specializes Part {
      feature wheelPort: DrivenPort subsets ownedPorts;
   }
   assoc struct DriveTrain specializes Interface {
      end drivePort: DrivePort[0..1];
      end drivenPort: DrivenPort[0..*];
   }
   struct Car specializes Part {
      feature driver: Person[0..1];
      composite feature engine: Engine[1];
      composite feature wheels: Wheel[4];
      composite connector drive: DriveTrain
      from engine.enginePort[1] to wheels.wheelPort[2];
   }
}
```

SysML
Syntax for Semantic Concepts

An *attribute definition* implies subclassification of the base type AttributeValue.

An *item definition* implies subclassification of the base type Item.

A *part definition* implies subclassification of the base type Part.

A *port usage* must be typed by a port definition. A port usage declared within a part definition subsets ownedPorts.

An *interface definition* implies subclassification of the base type Interface.

An *item usage* must be typed by an item definition. A *referential* usage is one that is *not* composite.

```
package SysML_Systems_Example {

   attribute def TorqueValue;

   item def Person;
   port def DrivePort {
      out attribute torque: TorqueValue;
   }
   part def Engine {
      port enginePort: DrivePort;
   }
   part def Wheel {
      port wheelPort: ~DrivePort;
   }
   interface def DriveTrain {
      end drivePort: DrivePort[0..1];
      end drivenPort: ~DrivePort[0..*];
   }
   part def Car {
      ref item driver: Person[0..1];
      part engine: Engine[1];
      part wheels: Wheel[4];
      interface drive: DriveTrain
          connect engine.enginePort[1] to wheels.wheelPort[2];
   }
}
```

A *port definition* implies subclassification of the base type Port.

An *attribute usage* must be typed by an attribute definition.

Every port definition includes a nested declaration of its conjugate, with "~" prepended to its name

A *part usage* must be typed by a part definition. A part usage declared within an item or part definition subsets subparts.

An *interface usage* must be typed by an interface definition, and it must connect port usages.

SysML
Systems Model Library

```
standard library package Attributes {
   …
   alias AttributeValue for DataValue;
}
```

```
standard library package Items {
   …
   abstract item def Item :> Object {
     …
}
```

```
standard library package Parts {
   …
   abstract part def Part :> Item {
      …
      abstract port ownedPorts: Port[0..*] :> ports, timeEnclosedOccurrences;
      abstract ref action performedActions: Action[0..*] :> actions, enactedPerformances;
      abstract action ownedActions: Action[0..*] :> actions, ownedPerformances {
         ref part :>> this : Part = that as Part;
   }

   a
}
```

```
standard library package Ports {
   private import Objects::Object;
   private import Objects::objects;
   abstract port def Port :> Object {
      ref self: Port :>> Object::self;
      port subports: Port :> timeEnclosedOccurrences;
   }

   abstract port ports : Port[0..*] nonunique :> objects;
}
```

SysML
Vehicle Library Model Example

A usage (feature) not nested within another type is considered to implicitly have Anything as its featuring type.

An abstract usage is one for which any instance must also be an instance of some declared subset. (An interface usage without connected features must be abstract.)

An abstract definition is one for which any instance must also be an instance of some declared specialization.

The subset of engines whose featuring type is a kind of Vehicle. (The redefinition also adds engines to the namespace for Vehicle.)

```
library package SysML_Vehicle_Library {
   attribute def TorqueValue;

   item def Person;
   item persons : Person[0..*] nonunique;

   port def DrivePort {
      out attribute torque: TorqueValue;
   }

   interface def DriveTrain {
      end drivePort: DrivePort[0..1];
      end drivenPort: ~DrivePort[0..*];
   }
   abstract interface driveTrains: DriveTrain[0..*];

   part def Engine {
      port enginePort: DrivePort;
   }
   abstract part engines: Engine[0..*] nonunique;

   part def Wheel {
      port wheelPort: ~DrivePort;
   }
   abstract part wheels: Wheel[0..*] nonunique;

   abstract part def Vehicle {
      abstract part redefines engines;
      abstract part redefines wheels;
      abstract part redefines driveTrains;
   }
}
```

This is a simple ontology of vehicle modeling concepts.

SysML
Applying Vehicle Model Concepts

This subsets the abstract Vehicle::engines, which is inherited from Vehicle.

```
package SysML_Vehicle_Example {
   import SysML_Vehicle_Library::*;

   part def Car specializes Vehicle {
      ref item driver: Person[0..1];

      part carEngine[1] subsets engines;
      part carWheels[4] subsets wheels;

      interface drive: DriveTrain subsets driveTrains
         connect carEngine.enginePort[1]
         to carWheels.wheelPort[2];
   }
}
```

SysML
Vehicle Metadata

Metadata are user-definable, model-level annotations of an element. *Semantic metadata* is metadata used to annotated a type in order to link it to a base type in a semantic library.

The *base type* of a semantic metadata annotation is bound to the "metacast" of a type from the semantic library.

The allowed *annotated element* for this metadata can be restricted to a specific abstract syntax metaclass.

```
package SysML_Vehicle_Metadata {
   import SysML_Vehicle_Library::*;
   private import Metaobjects::SemanticMetadata;

   metadata def person specializes SemanticMetadata {
      redefines baseType = persons meta SysML::Usage;
      subsets annotatedElement : SysML::Usage;
   }

   metadata def drive specializes SemanticMetadata {
      redefines baseType = driveTrains meta SysML::Usage;
      subsets annotatedElement : SysML::InterfaceUsage;
   }

   metadata def engine specializes SemanticMetadata {
      redefines baseType = engines meta SysML::Usage;
      subsets annotatedElement : SysML::PartUsage;
   }

   metadata def wheel specializes SemanticMetadata {
      redefines baseType = wheels meta SysML::Usage;
      subsets annotatedElement : SysML::PartUsage;
   }

   metadata def vehicle specializes SemanticMetadata {
      redefines baseType = Vehicle meta SysML::Definition;
      subsets annotatedElement : SysML::Definition;
   }
}
```

SysML
Vehicle DSML Example

A *user defined keyword* starting with # is a shorthand for annotating an element with the named metadata. The metaclass of the annotated element must be consistent with what is allowed for the metadata.

```
package Vehicle_DSML_Example {
   import SysML_Vehicle_Metadata::*;

   #vehicle def Car {
      ref #person driver[0..1];

      #engine part carEngine[1];
      #wheel part carWheels[4];

      #drive interface
         connect carEngine.enginePort[1]
         to carWheels.wheelPort[2];
   }
}
```

Specializations are now all implied based on the metadata annotations.

Specifications

- Adopted "Beta 1" Specifications
  - KerML – https://www.omg.org/spec/KerML/
  - SysML v2 – https://www.omg.org/spec/SysML/
- Latest FTF Revised Specifications
  - https://github.com/Systems-Modeling/SysML-v2-Release/tree/master/doc
- Finalization Schedule
  - March 2024 – "Beta 2" specifications available
  - September 2024 – Finalized ("Beta 3") specifications available
  - Mid 2025 – Formal specifications available