

From: Hans Peter de Koning
To: Bock, Conrad E. (Fed)
Subject: Meaning of abstract / isAbstract - my take
Date: Monday, October 14, 2024 5:30:29 PM

In order to explain what is meant by instance and instantiation, we must first recall and fully understand the semantics of **Type**, **Classifier**, **Feature**, **Class** and **DataType** in KerML, and subsequently **Definition**, **Usage** and **Individual** as well as **OccurrenceDefinition**, **OccurrenceUsage**, **AttributeDefinition** and **AttributeUsage** in SysML v2.

You'll have to excuse me for the long exposé, that contains quite some trivial stuff for OMG experts, but in my opinion is useful to get us all on the same page.

Type is specified and explained in KerML clause 7.3.2.1:

- Types classify things in a modeled system. The set of things classified by a type is the extent of the type, each member of which is an instance of the type. Everything being modeled is an instance of the type Anything from the Base library model (see 9.2.2).

So, a **Type** establishes a named set of possible instances (aka values) that may be bound or assigned to a model element typed by that **Type**.

KerML then specifies two disjoint specializations of **Type**, namely **Classifier** and **Feature**:

- KerML clause 7.3.3.1: Classifiers are types that classify things in the modeled system, as distinct from features, which model the relations between them (see 7.3.4). Subclassification is a kind of specialization that specifically relates classifiers.
- KerML clause 7.3.4.1: Features are types that classify how things in a modeled system are related, including by chains of relations. Relations between things can also be treated as things, allowing relations between relations, recurring as many times as needed. A feature relates instances in the intersection of the extents of its featuring types (the domain) with instances in the intersection of the extents of its featured types (the co-domain). Instances in the domain of a feature are said to "have values" that are instances of the co-domain. The domain of features with no explicit featuring types is the type Anything from the Base library model (see 9.2.2).

So, classifiers classify extents of single possible instances, and features classify extents of possible relationships between things, from a thing that is in the domain of the feature to a thing that is in the co-domain of a feature. Note: the co-domain is also the type (FeatureTyping) of the feature. A feature may relate another feature as source and/or target, so it is also possible to specify relationships between relationships or specify a feature on a feature.

The final important split in the KerML type taxonomy is the disjoint specialization of **Classifier** into **Class** and **DataType**:

- KerML clause 7.4.3: Classes are classifiers that classify occurrences, which exist in time and space (see 9.2.4). Relations between an occurrence and other things can change over time and space, while the occurrence still maintains an independent identity.
- KerML clause 7.4.2: Data types are classifiers that classify data values (see 9.2.2.2.2). Certain primitive data types have specified extents of values, such as the numerical and other types from the ScalarValues library model (see 9.3.2). Other data types have features whose values can distinguish one instance of the data type from another. But, otherwise, different data values are not distinguishable.

So, a **Class** classifies an extent of things that (can) have a life in time and space (and thus change), where each such thing is identified by a (unique) identifier. The identifier allows to (uniquely) reference a thing typed by a **Class**. A **DataType** classifies an extent of instances (aka values) that are identified by the values themselves, that do not have a life, but “are just there”, at least conceptually. E.g., number 3 (denoted by lexical symbol “3”) used in one model is the same value as number 3 used in any other location in the same or any other model. The same goes for e.g. mass : MassValue “25 [kilogram]” or name : String “My example”. Conceptually, for **DataType** values it is important to think of their “essential value”, independent of the selected concrete syntax / lexical symbol through which it is expressed.

The SysML metamodel builds on top of these fundamental KerML concepts. **Definition** specializes **Classifier**, and **Usage** specializes **Feature**, and by implication they both specialize **Type**.

SysML clause 7.6.1 explains:

- In general, a definition element classifies a certain kind of element (e.g., a classification of attributes, parts, actions, etc.). A usage element is a usage of a definition element in a certain context. A usage must always be defined by at least one definition element that corresponds to its usage kind. For example, a part usage is defined by a part definition, and an action usage is defined by an action definition. If no definition is specified explicitly, then the usage is defined implicitly by the most general definition of the appropriate kind from the Systems Library (see 9.2). For example, a part usage is implicitly defined by the most general part definition Part from the model library package Parts.

Note: the “certain context” above is the owning Namespace, typically the **Definition** or **Usage** or **Package** that owns a usage element.

In addition, the KerML split between **Class** and **DataType** in the SysML metamodel is mapped as follows:

- **OccurrenceDefinition** specializes **Definition**, **Class**

- **OccurrenceUsage** specializes **Usage** (and so in turn specializes **Feature**)
 - Note: an **OccurrenceUsage** must be defined by (typed by) an **OccurrenceDefinition**.
- **AttributeDefinition** specializes **Definition**, **DataType**
- **AttributeUsage** specializes **Usage** (and so in turn specializes **Feature**)
 - Note: an **AttributeUsage** must be defined by (typed by) an **AttributeDefinition**.

The above remarks on KerML **Class** and **DataType** apply equally to SysML **OccurrenceDefinition** / **OccurrenceUsage** and **AttributeDefinition** / **AttributeUsage** respectively.

To precisely define the meaning of instantiation in the SysML context, it is useful to recall the established OMG MOF layered architecture (see e.g. https://en.wikipedia.org/wiki/Meta-Object_Facility). MOF distinguishes the following four layers:

- **M3** meta-meta model – the basic concepts to specify a language
- **M2** meta model – a language specification (e.g. UML or SysML) conforming to **M3**
- **M1** user-authored model – conforming to an **M2** meta model (e.g. a model written in SysML)
- **M0** real-world instances model – conforming to an **M1** model

A SysML model always resides in **M1**, apart from metadata elements that are considered extensions of **M2**. The model in **M1** contains mostly types (definitions and usages), but it will typically also contain literal values (or a model evaluable expressions that yield literal values) assigned or bound to attributes. Such values could be considered instances because their “essential value” would be the same in **M1** and **M0** (disregarding lexical notation differences). However, to avoid confusion, attribute values in **M1** are probably better referred to as just “values”.

What SysML adds over KerML is the possibility to qualify an occurrence as being an **individual**. SysML clause 7.9.4 explains what this entails:

- An occurrence definition (of any kind) can be declared as an individual definition using the keyword **individual**, placed immediately before the kind keyword of the declaration. Alternatively, **individual** may be used in place of the kind keyword, in which case the declaration is equivalent to **individual occurrence** (that is, an occurrence usage not of a more specialized kind, but representing an individual).
- An occurrence usage (of any kind) is considered to be an individual usage if it has a definition that is an individual definition. An occurrence usage must not have more than one definition that is an individual definition. An occurrence usage may also be explicitly declared to be an individual usage using the keyword **individual**, placed after any of the other usage property keywords described in 7.6.3, but before a **timeslice** or **snapshot** keyword (if any). In this case, the occurrence usage must have exactly one definition that is an individual definition. If the declaration of an occurrence usage includes the the keyword **individual** (and, possibly, **timeslice** or **snapshot**), but no kind keyword, then this is

equivalent to having included the occurrence keyword (that is, an occurrence usage not of a more specialized kind, but representing an individual, and, possibly, a time slice or snapshot).

This is perhaps a little complicated to read, but it is important to understand that an individual occurrence definition or usage is still a **Type** (in **M1**) that specifies that the type's extent is comprised of one thing, i.e., a class or set with one member. An **OccurrenceUsage** defined by an individual **OccurrenceDefinition** can be used in the model to provide a kind of named variable (human readable identification) to reference that individual. Furthermore, an individual **OccurrenceDefinition** or **OccurrenceUsage** may bind values to all its usages, so that it is possible to tie down the specification at **M1** (similar to "instance specification" in UML and SysML v1) so that it fully prescribes any conforming **M0** instance.

This finally allows to explain what abstract (definitions) and instantiation mean.

Fundamentally a SysML model is always a specification in **M1** for any number of **M0** instantiations. An actual conforming **M0** instantiation is e.g. a real-world realization of a serial numbered product or an executing / executed analysis or simulation model. The SMC Execution WG is working hard to complete KerML Annex A "Model Execution" to specify what it means to instantiate an executable simulation or analysis model from a KerML or SysML user model.

An **abstract Usage** is (1) a Usage defined by (exclusively) one or more abstract Definitions, or (2) a Usage that is directly declared abstract.

Note: To make an abstract usage concrete, it must be redefined by a concrete subclassification of (at least one of) its abstract Definitions, or it must be redefined by a concrete subsetting of the original usage.

Coming back to the original answer, as KerML clause 7.3.2.2 says: A type is specified as abstract by placing the keyword abstract before the keyword type. A type that is not abstract is called a concrete type. Declaring a type to be abstract means that all instances of the type must also be instances of at least one concrete type that directly or indirectly specializes the abstract type.

KerML or SysML tools with fully implemented diagnostics would be expected to raise a warning or error in two situations:

1. When a literal value (or a model evaluable expression that yields a literal value) is assigned or bound to an **abstract Usage** in an **M1** model.
2. When an **abstract Usage** is instantiated in an **M0** model.