Table of Contents

Table of Figures

Table of Tables

0

1 Introduction to SMIF Semantics

The following is a high-level, non-normative, description of some of the fundamental SMIF concepts.

The fundamental concepts will be described in a way that most practitioners can relate it to their familiar experiences. In this chapter we will gradually build a semantic-conceptual architecture (an architecture that is completely independent of any particular technology and in which there is a clear distinction between the world of the things and the world of the representations of those things).

The prime aim of this chapter is to demonstrate the value that a SMIF semantic-conceptual model and transformations can offer to the ever-increasing need of federation of (information) systems in business and government practice. Note that this section amplifies the reference documentation in section 8. Section 8 should be consulted for specific concept definitions.

1.1 T*he SMIF Conceptual Model Foundation*

The SMIF conceptual model serves three potential purposes:

- It defines the SMIF language
- It provides foundation concepts which other models may directly use, including domain models
- As a reference model to which other, independently conceived, models may be mapped (where there are concpets in common).

SMIF has been built with the expectation that by providing *reference models* that define *common shared concepts* we can either *directly reuse* those concepts or *map* them to related concepts in different models or data structures. This is the essence of federation.

Many of the concepts used to define the SMIF language **may** also be used as reference concepts for domain models. Many of the fundamental concepts needed for the SMIF language are also found in many, if not all, domain models. Examples would be entities, identifiers, situations and values. That said, there is no requirement that these concepts be used or referenced by SMIF domain models – the choice of what reference models to use is made by the domain architect, not by SMIF.

SMIF, as a language for modeling, needs to interoperate with and share concepts with other languages such as UML, OWL or XML-Schema. This is really the same problem as an application containing, for example, company information it may need to share information with other applications providing or consuming company information. The basis for sharing information, at any level, is that there are different ways to represent information but they must, ultimately, be sharing meaning (concepts) for useful communication to take place. Communications takes place when you *understand* what another party has said based on some concept you share about the world, system or domain you are communicating about. If there are no shared concepts there can be no understanding, not communications.

To understand what is said you must have some way to *reference* a concept you share. We reference a shared concept by using *terms*, or "*signs*". Those signs can be textual or even gestures, like pointing at something. Natural language uses words or phrases as these signs. But, since words can have many or fuzzy meanings SMIF also references concepts with model based identifiers. These model based identifiers serve as signs to connect a more formalized definition of a concept, in a conceptual reference model, with the various ways that concept may be used or expressed.

The following section identifies common concepts used by and defined within SMIF that may also be used in domain models as well. The way concepts have been partitioned in SMIF to enable its use as a reference model across language concepts may serve models at many levels. This approach to partitioning models *may* be useful in other domains as well. Of course, some of the SMIF concepts are more focused on language design and are less useful for typical domains.

The SMIF model has already been used in this way, it is used and extended by the [ThreatRisk] conceptual model which is used in this section to provide examples.

1.1.1 Thing

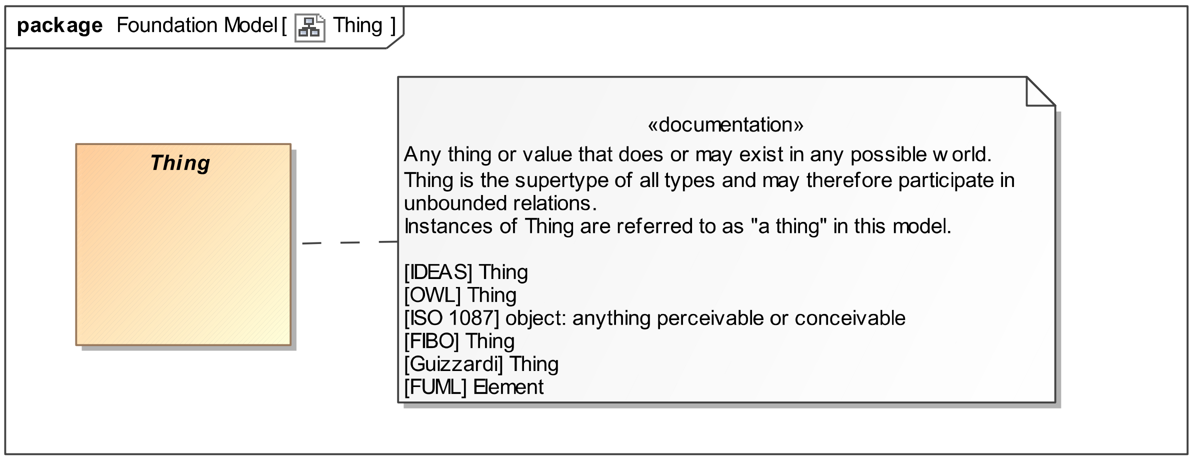

package Foundation Model [Thing]
Thing
«documentation»
Any thing or value that does or may exist in any possible w orld.
Thing is the supertype of all types and may therefore participate in unbounded relations.
Instances of Thing are referred to as "a thing" in this model.
[IDEAS] Thing
[OWL] Thing
[ISO 1087] object: anything perceivable or conceivable
[FIBO] Thing
[Guizzardi] Thing
[FUML] Element

Figure 1.1: Thing

In many models it is convenient to have a sign for anything that could possible be in any world view, any data repository or any model – the most general concept possible and therefor a "super-type" of everything else. We (and many others) call this concept "Thing". As a concept for anything, "thing" may be considered somewhat meaningless – but it is a convenient concept, and one that is very common in models and data structures. More interesting concepts will all be sub-types of "thing".

Examples of things are "George Washington", "The song – Rock of ages", "Unicorns" and the number "5". Other examples includes a DBMS record about George Washington or a recording of the "Rock of ages". Note that things include "real worlds" things as well as made-up things and data about things we find in computers or filing cabinets (mellennials may have to look up the concept "filing cabinet").

Semantics

Everything that is in any world, domain, model or data structure is, directly or indirectly, an instance of “Thing”.

For all X, Thing(X)

1.1.2 Type

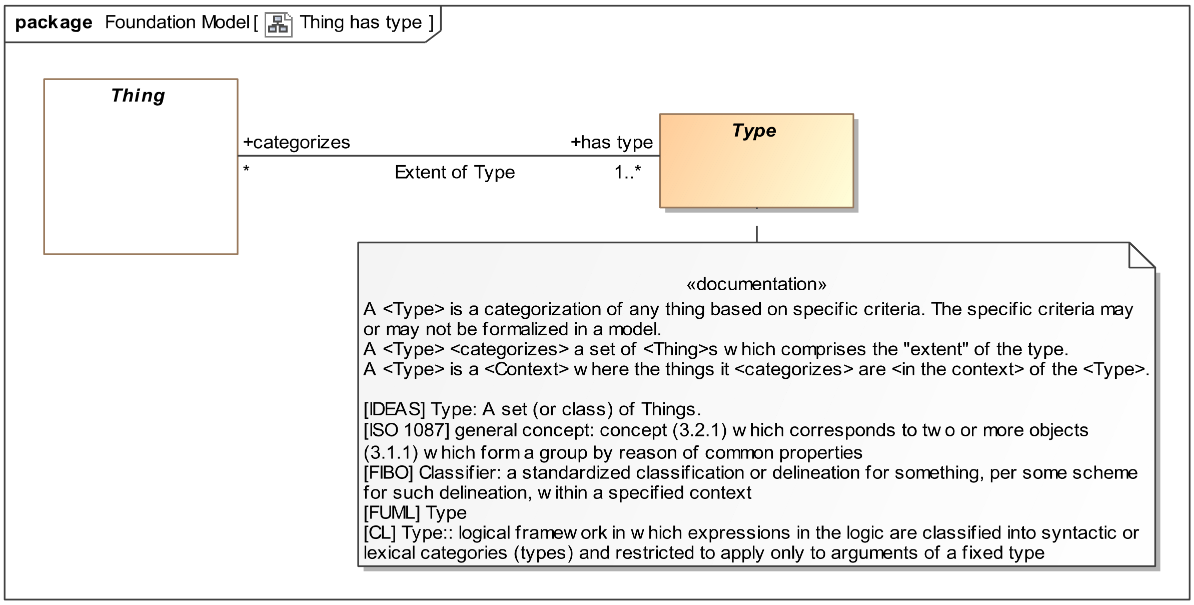

package Foundation Model [Thing has type]
Thing
+categorizes
*
Extent of Type
+has type
1..*
Type
«documentation»
A <Type> is a categorization of any thing based on specific criteria. The specific criteria may or may not be formalized in a model.
A <Type> <categorizes> a set of <Thing>s w hich comprises the "extent" of the type.
A <Type> is a <Context> w here the things it <categorizes> are <in the context> of the <Type>.
[IDEAS] Type: A set (or class) of Things.
[ISO 1087] general concept: concept (3.2.1) w hich corresponds to tw o or more objects (3.1.1) w hich form a group by reason of common properties
[FIBO] Classifier: a standardized classification or delineation for something, per some scheme for such delineation, w ithin a specified context
[FUML] Type
[CL] Type:: logical framew ork in w hich expressions in the logic are classified into syntactic or lexical categories (types) and restricted to apply only to arguments of a fixed type

Figure 1.2: Thing has type

A primary way we understand things is by categorizing them as types of things. The concept of “Type” is common across most human and modeling languages. The concept of the type of a thing is also common in domain models, such as product types, malware types. Kinds of financial instruments or kinds of fish. A type <categorizes> a set of things of that type, all of these things <has type> of one or more types. The relationship between things and types is called the “Extent of Type”.

Things and how they are categorized as types is one of the primary conceptual mechanisms used in SMIF and most other languages – it is part of how we as humans understand the world. Also note that we expect things may have any number of types, and those types could even change over time or be different in various context – such a “multiple classification” assumption fits with the way our world works and is understood. The multiple classification assumption is different than most “object oriented” programming languages that restrict objects to a single type that can’t change.

Remember that we said everything is a “Thing”, well, types are things as well – we will see how this works later when we see the full hierarchy where Type is defined.

There is a somewhat theoretical discussion about types being defined by “intention” (what we think they mean) or “extension” (enumerating the set of things that are the extent of that type). Type, at this level, may be defined either way. Our norm is to define types by intention based on our observation of and understanding of the world we live in.

As an aside, a notation convention we use: that the primary things we are discussing are shaded where as other related things are not. Also note there are reverences, e.g. [FUML] to other standards with like concepts.

Example 1

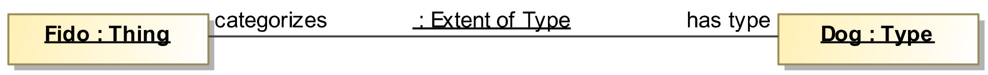


Figure 1.3: Fido is a dog example

In this example we are saying “Fido” is a dog. In terms of the model, there is an “Extent of Type” relationship between “Fido” and the type “Dog” where “Dog” <categorizes> “Fido” and “Fido” <has type> dog. This relationship is one “fact” in our model that can be read either way, from dog to Fido or Fido to dog.

We are also introducing the use of UML “Instance Diagrams” to illustrate our examples.

We would probably never just use “Thing” to categorize “Fido”, we would categorize Fido as something more specific - “down the hierarchy” of types – here we see that Fido is a Dog and that Dog is a kind of animal.. As a shortcut well will usually not show the “Extent of Type” relationship in examples, we will just show the types of something after the name – as is provided for in UML instance diagrams. So the UML shorthand for writing out all the explicit relationships is:

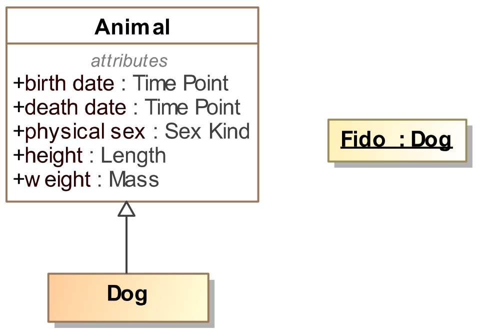


Figure 1.4: UML Type and Instance Example

We should understand that whenever we see an “instance with a name followed by “:” and a type, it means that this instance <has type> of a type with that name as part of an Extent of Type relationship. There could be multiple types listed, separated by commas (i.e. Fido could also be a Pet).

Semantics

For all things X, where X <has type> T, X shall be conform with the propositions that hold within T.

The set Extent of Type(T) = all things X, where X <has type> T

In logic, type may also be considered a function, which also implies:

For all things X, where X <has type> T, T(x)

Note: The constructs for determining the propositions that hold within T as well as the semantics of relationships are described below.

1.1.3

1.1.4 Identifiable Entities and Values

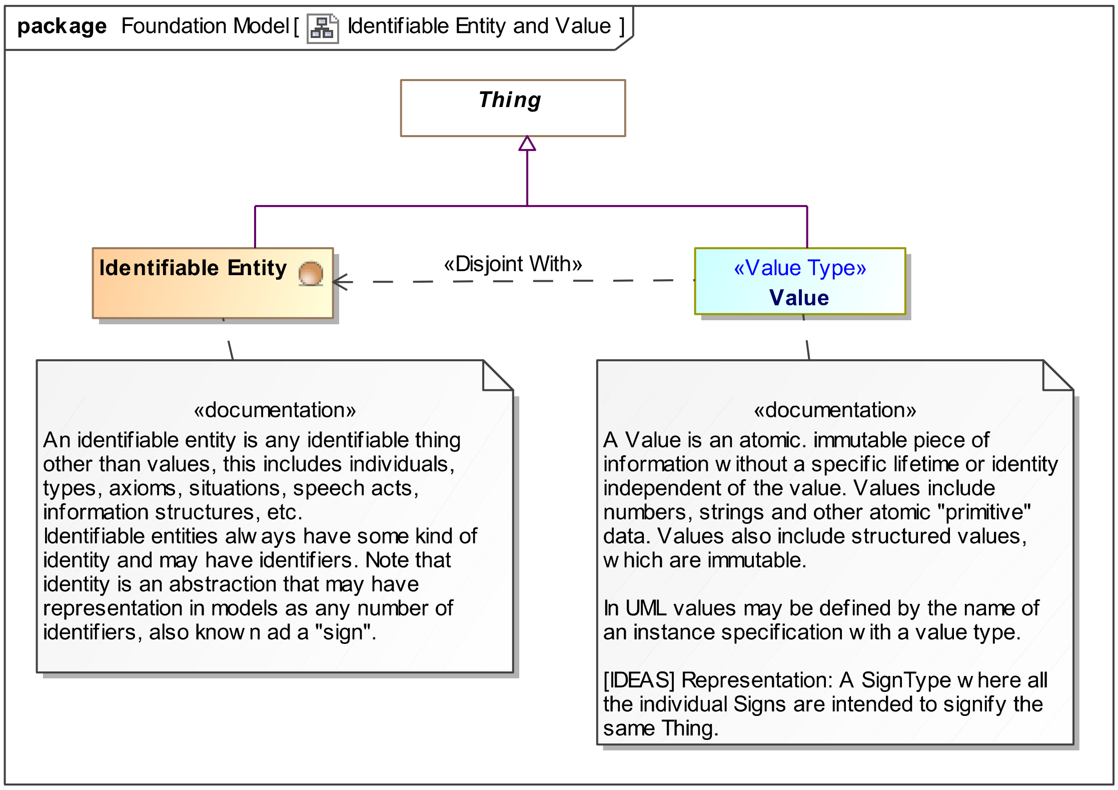

package Foundation Model [Identifiable Entity and Value]
Thing
Identifiable Entity
«Disjoint With»
«Value Type»
Value
«documentation»
An identifiable entity is any identifiable thing other than values, this includes individuals, types, axioms, situations, speech acts, information structures, etc.
Identifiable entities alw ays have some kind of identity and may have identifiers. Note that identity is an abstraction that may have representation in models as any number of identifiers, also know n ad a "sign".
«documentation»
A Value is an atomic. immutable piece of information w ithout a specific lifetime or identity independent of the value. Values include numbers, strings and other atomic "primitive" data. Values also include structured values, w hich are immutable.
In UML values may be defined by the name of an instance specification w ith a value type.
[IDEAS] Representation: A SignType w here all the individual Signs are intended to signify the same Thing.

Figure 1.5: Identifiable Entities and Values

We are presenting the concepts “Identifiable Entity” and “Value” together as they are best understood as complementing each other. Identifiable entities and values are, of course, both kinds of things – but of a very different nature. Identifiable Entities are what we mostly talk about – things we give names to, things that have some kind of independent “identity” - everything we can see & touch are identifiable like people, rocks and dogs. Intangibles can also be identifiable, such as purchases, threats or processes. Many, but not all, identifiable things have some kind of “lifetime” where that may change over that lifetime yet retain their individuality.

Values, on the other hand, “just are”. One way this is explained is that values have no identity or lifetime other than the value it’s self – which can never change and are the same everywhere. All numbers are values as are quantities like “5 Meters” or “pure data” like the text string “abc”. The number “5” is the same number five everywhere (even if it has different representations) – it makes no sense to “delete” 5! The text string “abc” is indistinguishable from the text string “abc” in any other document or database. Values are typically used to describe characteristics of things, such as the weight of rock “R555” is 5 kilograms. Note that values may have representations in our models and data, but they all represent the same underlying value.

In the SMIF foundation model we partition things as being values or identifiable entities. Something can’t be a value and identifiable entity – these classifications are “disjoint”.This partitioning, like most of our concepts, is found in many other languages and ontologies – both modeling languages and human languages. Domain models typically use the same kind of partitioning and may use or map to the SMIF concepts.

Examples

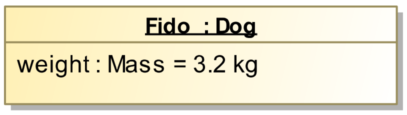


Figure 1.6: Identifiable Entity with Value Characteristic

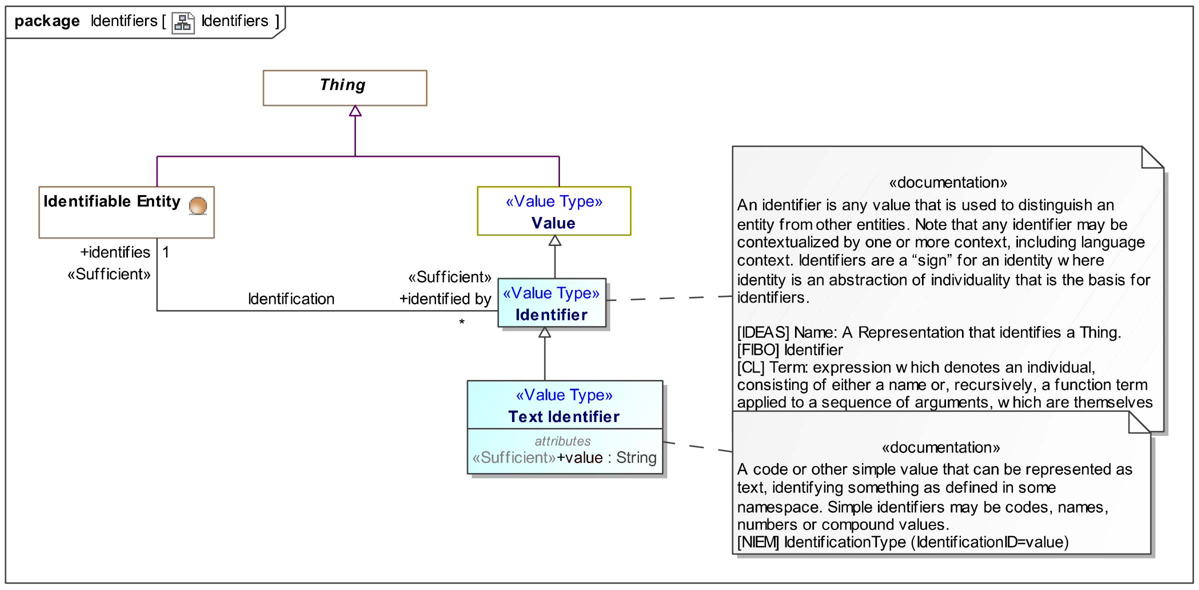
Returning to Fido for a moment, Fido is clearly an “Identifiable Entity” with a lifetime. We use values, like quantities, to define characteristics of identifiable entities – like their weight.

The above “Characteristic” for Fido, shown as the value of a UML property, states that the weight of Fido is 3.2 KG – a nice lap dog. This is how values are typically used with identifiable entities (like dogs, people or computers). We will see in more detail how characteristics are represented in the SMIF model later. We will also see how we can understand how the weight of Fido may change over time (what we see above is just a “snapshot” of Fido, (perhaps when he was a puppy)).

The rule we use is that Characteristics are always have values as their type - this clearly distinguishes characteristics from relationships between identifiable entities. This is a recommended convention but is not a SMIF constraint to allow for various methodologies.

1.2 Identifiers

1.2.1 Basic Identifiers



An identifier is any value that is used to distinguish an entity from other entities. Note that any identifier may be contextualized by one or more context, including language context. Identifiers are a “sign” for an identity w here identity is an abstraction of individuality that is the basis for identifiers.

[IDEAS] Name: A Representation that identifies a Thing.
[FIBO] Identifier
[CL] Term: expression w hich denotes an individual, consisting of either a name or, recursively, a function term applied to a sequence of arguments, w hich are themselves

A code or other simple value that can be represented as text, identifying something as defined in some namespace. Simple identifiers may be codes, names, numbers or compound values.
[NIEM] IdentificationType (IdentificationID=value)

Figure 1.7: Basic Identifiers

Even in our simple examples we have been naming things – giving them “Signs”, like “Dog” and “Fido”. Most models and data structures have ways to name things. SMIF defines the basic concept of an “Identifier” that <identifies> some identifiable entity. There is an “Identification” relationship between an identifiable thing and what it is <identified by>. Note that something may be identified by any number of identifiers, or none at all. Please keep in mind: *The “thing” that is identified is different from the values that identify it – one of its signs*. It is also different from a data record that provides information about something. Modelers need to be clear about what the elements in their model really represent – real world entities, data records or perhaps social conventions.

One of the design philosophies we have used in SMIF is that we should not "commit" to anything unless it is *necessarily true* for the concept we are defining – but when something is necessarily true, we should state it. In this case we don't want to commit to an identifier being text (it could be a picture, gesture or a sound). We do want to commit to identifiers being a kind of value as identifiers should not change. In a sub-type of Identifiers we make a stronger commitment - "Text Identifier" which has a string value. Text identifier is a sub-type of "Identifier that makes a *commitment* to the value being a string and *represents* the more abstract Identifier concept.

Examples

Returning to Fido again; "Fido:Dog" is really a double shortcut, we are asserting two "facts" - that Fido is a dog (which we saw above) and that Fido has the identifier "Fido". A full instance model would look like this:

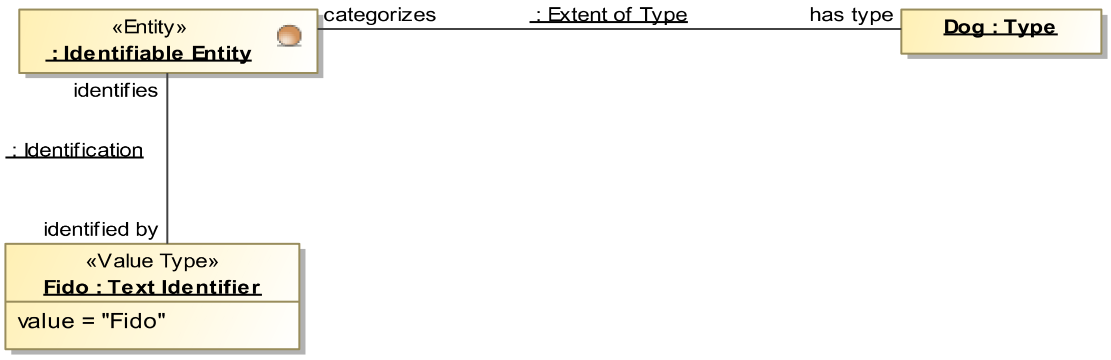


Figure 1.8: Fully expanded type and identifier instance model

Here we see that there is "some identifiable entity" that <has type> Dog and is <identified by> the text identifier "Fido" (noting that there could be other identifiers as well). Also note that the Identifier identifies the entity and that identifier has some *value*. The same value could be used in other identifiers – so at this level we are not saying anything about the identifiers string value "Fido" being unique.

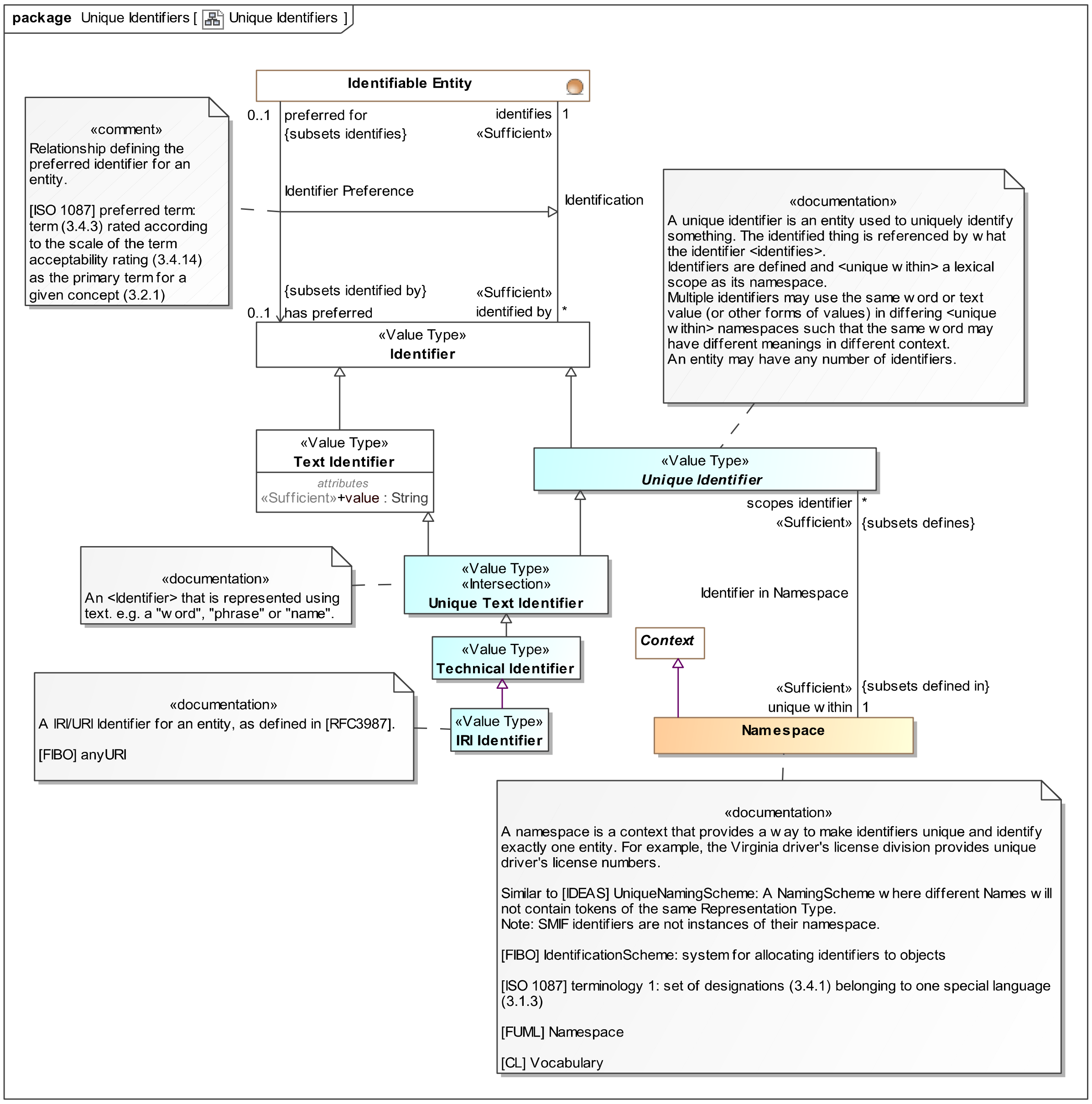
Relating this to some DBMS, we could store a "Record" that represents Fido and has a column representing names. In thinking about the DBMS, we want to distinguish the "real Fido" from records about Fido. The Fido element above is intended as a *sign for the real Fido* – not data about Fido. Likewise, the "Dog" type is intended to represent "real dogs", not dog records. Of course DBMS systems are real things also, but they contain data *representing* Fido – so *we distinguish a model element representing the real things and real relationships between them from records (data) about those things*. This separation of concerns is the foundation of information federation. We will explore this separation of concerns in more depth below.

There are also other types and relationships in SMIF to be able to distinguish names, like "Fido" from controlled identifiers, like a dog-tag number - we will see more about this next.

1.2.2 Unique and Preferred Identifiers

Fido may have many names and identifiers, such as his dog tag number and the ID of the "Chip" that can be used to find Fido if he is lost. The dog tag and chip ID are expected to be unique. His name, Fido, could be used for many dogs – it isn't unique but it may be the name we would prefer to use when talking about him.

Since SMIF is intended to work with data from multiple sources that will identify the same things in different ways it is important to be able to hold many forms of identifiers and relate them to the same entity. It is also important, where identifiers are unique, to be able to understand the scope of that uniqueness – there needs to be some authority or convention that makes them unique. This same "multiple identity" problem exists when any application is "fusing" data from multiple sources – so a foundation model for identifiers has broad applicability.



«comment»
Relationship defining the preferred identifier for an entity.

[ISO 1087] preferred term: term (3.4.3) rated according to the scale of the term acceptability rating (3.4.14) as the primary term for a given concept (3.2.1)

«documentation»
A unique identifier is an entity used to uniquely identify something. The identified thing is referenced by what the identifier <identifies>.
Identifiers are defined and <unique within> a lexical scope as its namespace.
Multiple identifiers may use the same word or text value (or other forms of values) in differing <unique within> namespaces such that the same word may have different meanings in different context.
An entity may have any number of identifiers.

«documentation»
An <Identifier> that is represented using text. e.g. a "word", "phrase" or "name".

«documentation»
A IRI/URI Identifier for an entity, as defined in [RFC3987].

[FIBO] anyURI

«documentation»
A namespace is a context that provides a way to make identifiers unique and identify exactly one entity. For example, the Virginia driver's license division provides unique driver's license numbers.

Similar to [IDEAS] UniqueNamingScheme: A NamingScheme where different Names will not contain tokens of the same Representation Type.
Note: SMIF identifiers are not instances of their namespace.

[FIBO] IdentificationScheme: system for allocating identifiers to objects

[ISO 1087] terminology 1: set of designations (3.4.1) belonging to one special language (3.1.3)

[FUML] Namespace

[CL] Vocabulary

Figure 1.9: Unique Identifiers

In figure 1.9 we have shown some additional concepts to handle uniqueness and preference.

Note the "Identifier Preference" relationship that specializes the "Identification" relationship we have already seen. Also note the "ends" of this relationship "subset" the corresponding ends of "Identification". Relationships, as well as the ends of relationships, form a generalization hierarchy from more general to more specific (we don't always show this hierarchy in summary diagrams). The "Identifier Preference" relationship adds something to Identification, that the <has preferred> identifier has more priority for communication with people. This is a “soft” semantic, intended to assist in human understanding. When we show something we may not want to see all the identifiers, just the preferred one. Also note that what is preferred in one context may not be preferred in another, such as in another domain or language. later we will see how context can be used to impact what relationships are valid in any particular circumstance.

The other concept we are introducing is that of uniqueness. For some identification value to be unique it really needs to be unique within something – some authority or convention that provides that uniqueness. So a “Unique Identifier” is <unique within> some “Namespace”. A namespace could be technical, like a block of code, or social and based on an authority like names of streets within a town, in which case the town defines the namespace. The “URL” (Universal Resource Locator) is a well known kind of unique identifier, based on an IETF standard: 3987. Providing uniqueness in this way is also a form of contextualization, we will explore context more below.

Names and Terms

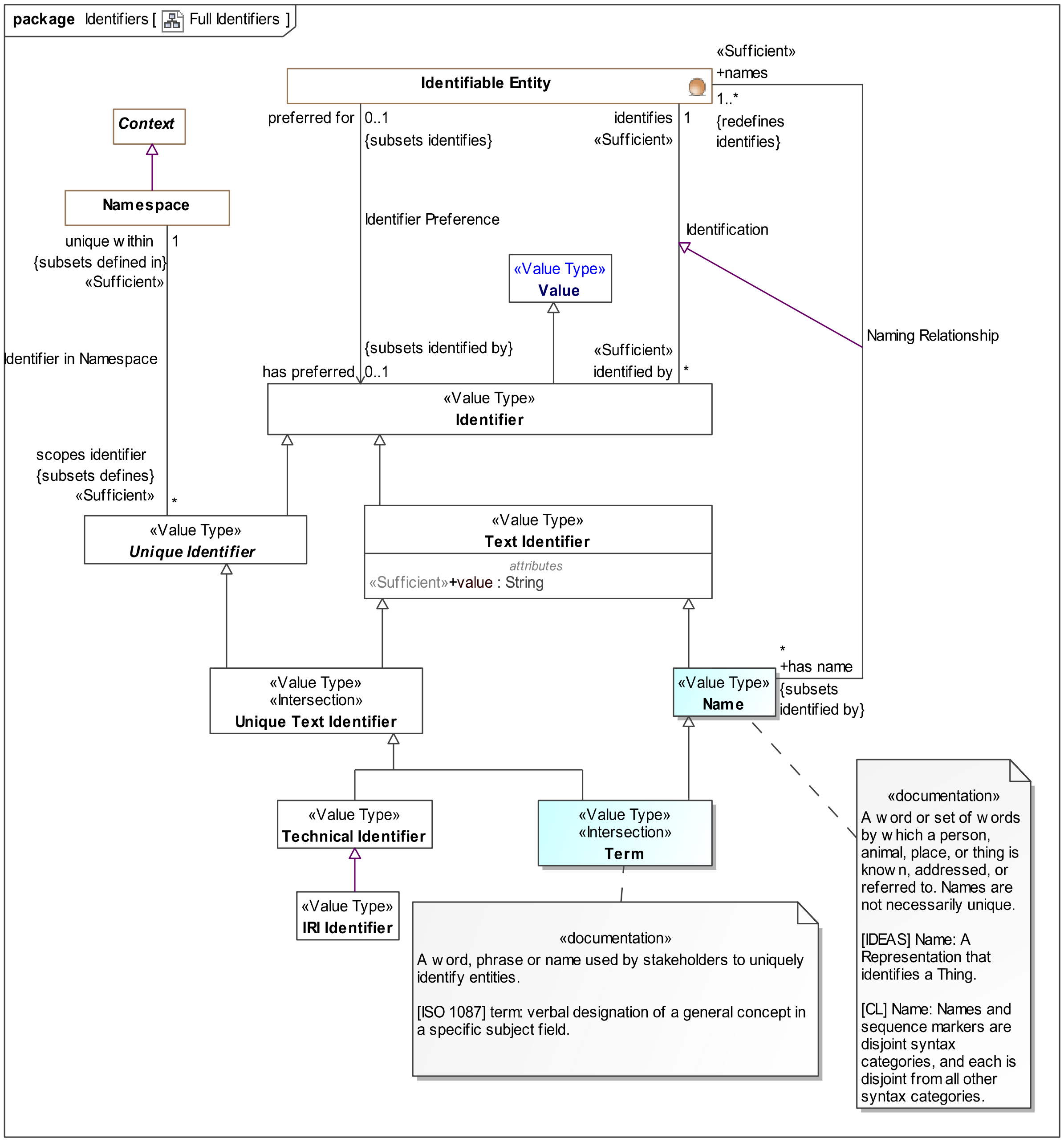


Figure 1.10: Full Identifiers Package

We complete our tour of identifiers by showing the complete identifiers package that includes “Name”, “Term” and the “Naming Relationship”. Names are identifiers intended to be meaningful to people – most often derived from natural language vocabulary or proper nouns. By typing an identifier as a name (of the concept) we expect that people will be able to relate the name to their intuitive understanding. Compare this with technical identifiers, which may be meaningless symbols. Combining the idea of a name with a unique identifier we get the concept of a “Term”. A term is a name that is unique within some namespace.

Considering these refinements of identifiers we may want to make our Fido example a bit more precise by defining “Fido” as a name and also including a unique identifier, like a Virginia dog tag number.

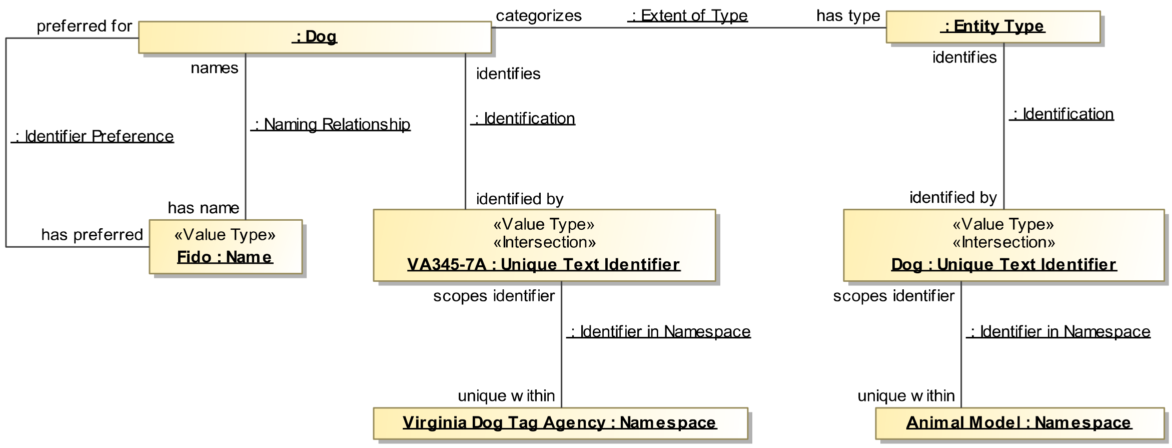


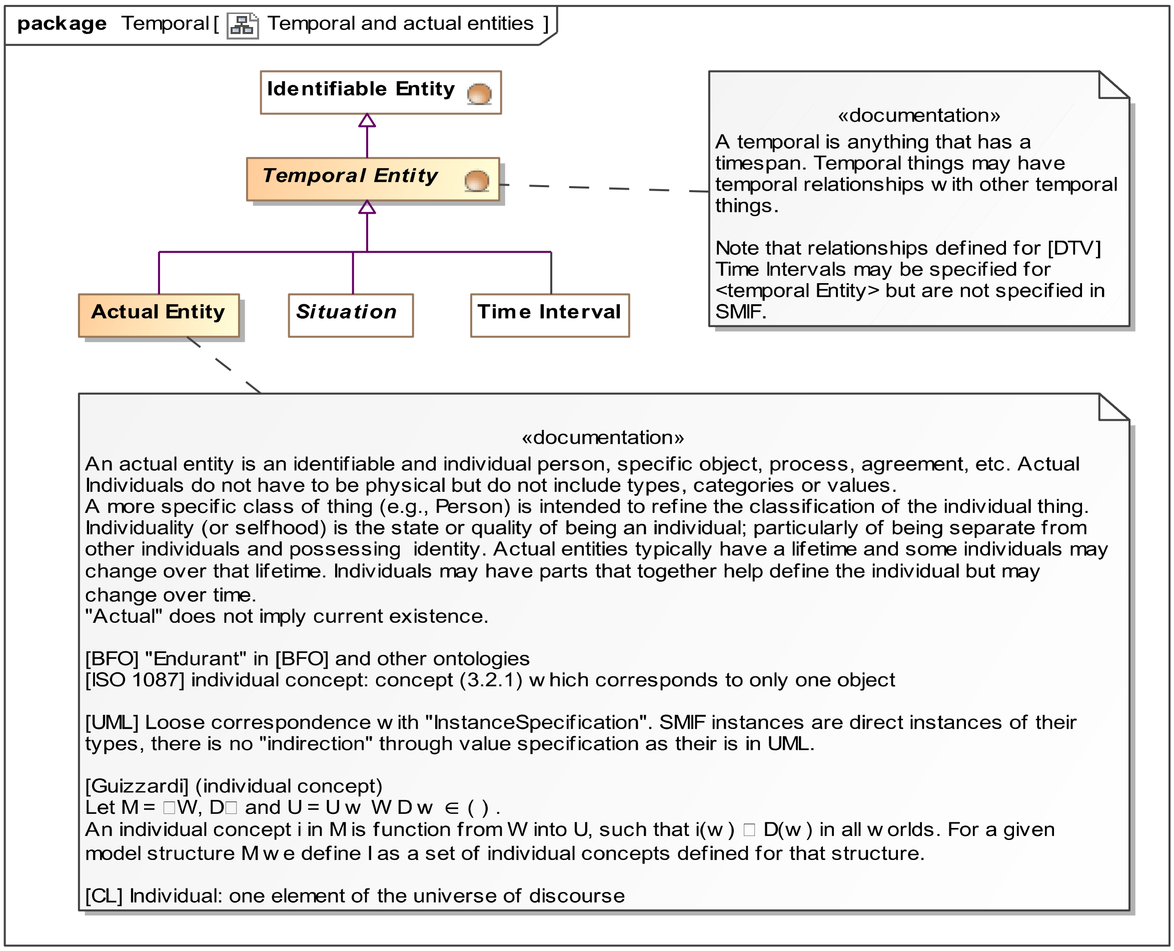
Figure 1.11: Full Identifier Example

From this example we can see that Fido is an identifiable entity that <has type> Dog. Dogs can have any number of identifiers and that some may be unique within specific namespaces such as the "Virginia Dog Tag Agency". We can also see that "Fido" is a human meaningful name that is the preferred identifier for Fido (in this context). Also note that the Entity Type "Dog" also has an identifier unique within some other namespace – in this case "Animal Model".

As noted above – this model for identifiers is used by the SMIF language and *may* also be used by or mapped to domain reference models that deal with names and identifiers.

1.3 Temporal and Actual Entities

As noted above, identifiable entities can be anything other than values. Most of the things we deal with have some kind of lifetime – they exist in time. Some of these can be considered “actual entities, or “individuals”. The next layer in the SMIF model defines temporal entities and actual entities.



«documentation»
A temporal is anything that has a timespan. Temporal things may have temporal relationships w ith other temporal things.

Note that relationships defined for [DTV] Time Intervals may be specified for <temporal Entity> but are not specified in SMIF.

«documentation»
An actual entity is an identifiable and individual person, specific object, process, agreement, etc. Actual Individuals do not have to be physical but do not include types, categories or values.
A more specific class of thing (e.g., Person) is intended to refine the classification of the individual thing. Individuality (or selfhood) is the state or quality of being an individual; particularly of being separate from other individuals and possessing identity. Actual entities typically have a lifetime and some individuals may change over that lifetime. Individuals may have parts that together help define the individual but may change over time.
"Actual" does not imply current existence.

[BFO] "Endurant" in [BFO] and other ontologies
[ISO 1087] individual concept: concept (3.2.1) w hich corresponds to only one object

[UML] Loose correspondence w ith "InstanceSpecification". SMIF instances are direct instances of their types, there is no "indirection" through value specification as their is in UML.

[Guizzardi] (individual concept)
Let M = □W, D□ and U = U w W D w ∈ ( ) .
An individual concept i in M is function from W into U, such that i(w) □ D(w) in all w orlds. For a given model structure M w e define I as a set of individual concepts defined for that structure.

[CL] Individual: one element of the universe of discourse

Figure 1.12: Temporal and Actual Entities

Temporal entity is primarily a placeholder in the SMIF model – no characteristics or relationships are defined directly on temporal entity. Other specifications, such as the threat/risk model, augment Temporal Entity (and most of the other foundation concepts) with specific relationships derived from the OMG [DTV] specification. However, specifying that “actual entity” is temporal assists in more precisely defining its semantics.

Actual entities are the individual things we deal with – they are not types , categories or sets; actual things. By actual we don’t mean necessarily physical, for example a “threat” may be consider an actual entity if it is an actual threat. A purchase may also be considered an actual entity. Actual also dopes not necessitate something happening “now” it could be in the past, current or a possible future. Most of the “interesting” things will be subtypes of “Actual Entity” - like person, tree or a person running.

Other kinds of temporal entities are situations and time intervals. Situations are discussed below, Time Interval is an example of how SMIF concepts can be specialized in other specifications, Time Interval is defined in [ThreatRisk] and only shown here to complete the example.

Examples

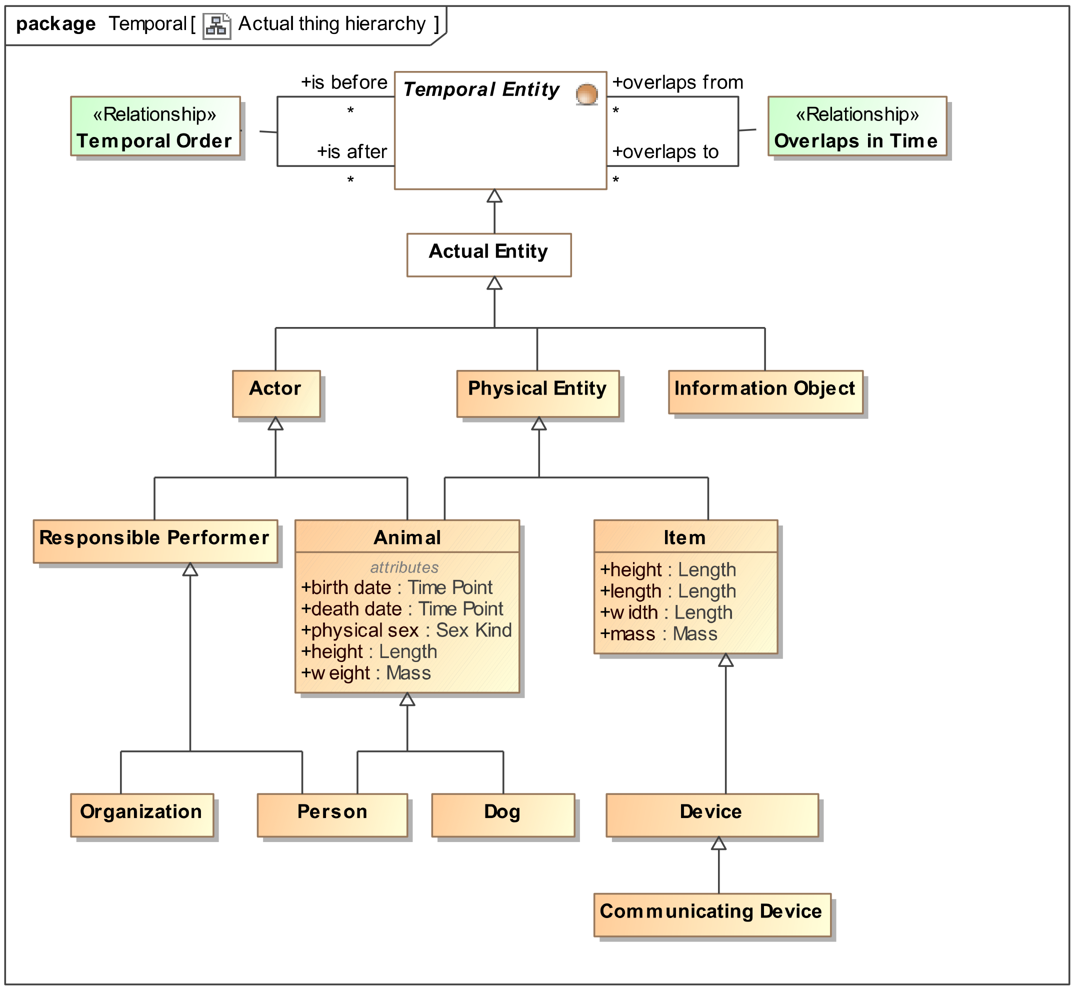


Figure ,1.13: Actual Thing Hierarchy Example

Figure 1.13 with types from [ThreatRisk] (except “Dog” -we made that up for these examples), show how a hierarchy of domain concepts *in another reference model* can *specialize and augment* “Actual Entity” and “Temporal Entity”. By using concepts in a reference model we get a lot for free, for example all of these entities can have identifiers. We then add what is missing; we are showing just two of the relationships defined in [ThreatRisk] for Temporal Entity: Temporal Order and Overlaps in Time. These relationships then apply to all subtypes of Temporal Order *in any model using or mapping to Temporal Entity.*

A frequent complaint that is heard: *“but I don’t care about the sex of animals!”, we will never agree on the “right” set of characteristics and relationships for anything!* This is one of the fundamental difference between a conceptual reference model and an application model; you use what you need and ignore the rest – you only agree on what you need to agree on. Every concept in a reference model is its “own thing” that can be used, or not, in any other model. Since it can be mapped to other models, these other way to say the same or related things may use different names, different structures or more or less relationships and attributes. The reference model is only there to “connect the dots” between concepts shared across different representations, applications or communities. If an application doesn’t need something, it is simply not mapped. If something is missing – augment the reference or add it in another reference model. Using reference models and mappings frees applications from the tyranny of having to do it "their way" when integrating with external system while still providing for interoperability and collaboration.

Based on such a hierarchy and relationships we can start to model interesting facts, for example:

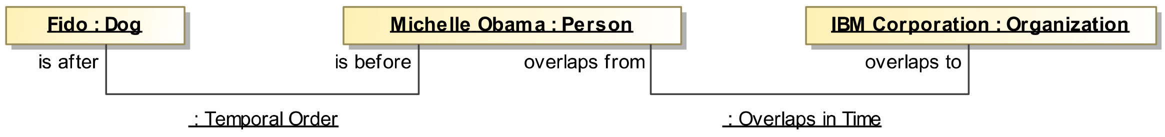


Figure 1.14: Temporal Instance Example

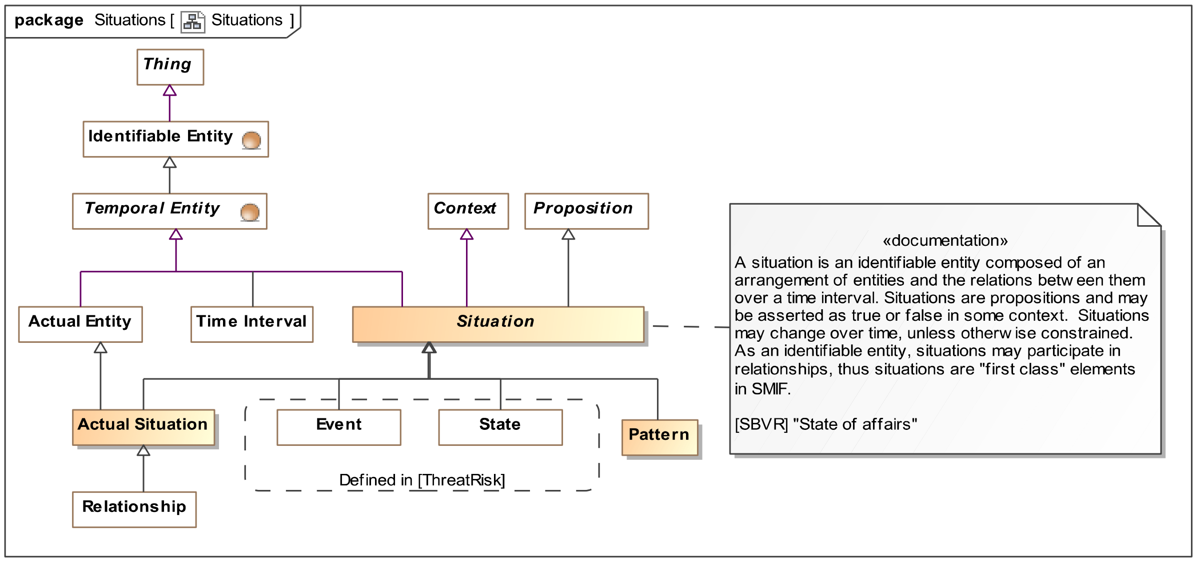
This example defines three "actual entities": Fido, Michelle Obama and IBM Corporation. It further states that the lifetime of "Fido" was before "Michelle Obama" and that there is some overlap in the lifetimes of "Michelle Obama" and "IBM Corporation". If anything concerning temporal relationships exists in some data repository, it can be mapped to concepts in [ThreatRisk], and/or any other reference model with similar concepts, like [FIBO]. Note that all the facts in this example would not need to come from the same source – we may have "mapped" data from multiple sources so as to be better able to "connect the dots" and reach new conclusions. Since these temporal relationships are based on the OMG date-time standard [DTV], that standard could be used to reason about temporal objects.

We will delve into this in more detail later – but it is interesting to note that relationships are temporal objects as well. So it is possible to say when a relationship holds as well as the entities it holds between. This enables SMIF models to understand the different assumptions made about time or when something happened in various data models or ontologies. In formal terms, this enables SMIF models to be "4D" (where time is the 4th dimension) but also allows such time considerations to be implicit where they are not as interesting.

1.4 Situations (Upper Level)

Another kind of temporal entity is situations. Where "actual entities" are individuals, situations are configurations of individuals over some time-span. As configurations of individuals we can consider situations from the "outside" or the "inside". On the "outside" we just talk about the situation; the state of the reactor, the process of the hurricane developing, etc.

On the inside we need to consider how to represent these configurations, this uses "context" and "propositions". First we will consider situations from the outside, then on the inside.



«documentation»
A situation is an identifiable entity composed of an arrangement of entities and the relations between them over a time interval. Situations are propositions and may be asserted as true or false in some context. Situations may change over time, unless otherwise constrained. As an identifiable entity, situations may participate in relationships, thus situations are "first class" elements in SMIF.

[SBVR] "State of affairs"

Figure 1.15: Situations - Top Level

What are situations? A terrorist entering and airport. A policeman at a concert. A rock falling, a cup full of water. Even relationships are situations – the situation of one thing being related to another in some specific way for some period of time, such as a cup holding water or a person in a house. In the SMIF language, relationships and characteristics are some of the primary uses of situations. This allows relationships to be involved in other relationships, such as when they happened, why, where information came from, or who was involved.

Situations include both occurrences of things happening (called events in this example) as well as static conditions, such as a cup sitting on a desk (called states). As they are not needed for the definition of the SMIF language Event and State are not defined directly in the SMIF model – we show subtypes of Situation defined in [ThreatRisk] as examples.

Situations include all conditions and processes; actual as well as possible. Possible situations can be patterns. Patterns provide for possible situations with some variable aspect that can then match multiple actual situations.

It is expected that situations will be augmented in reference models, [ThreatRisk] augments situations with concepts like causation – an accident causing injury.

From the "outside", situations look like any other entity so we will not introduce another example. We will introduce some other concepts prior to exploring situations from the "inside" - describing the configuration of things.

1.5 Kinds of Types (Metatypes)

1.5.1 SMIF Language Metatypes

We have covered some basic kinds of things: Values, Identifiable Entities & Situations. For these fundamental kinds there are specific "meta types" for each – subtypes of the abstract concept of a Type. These meta types provide a way to properly define other types.

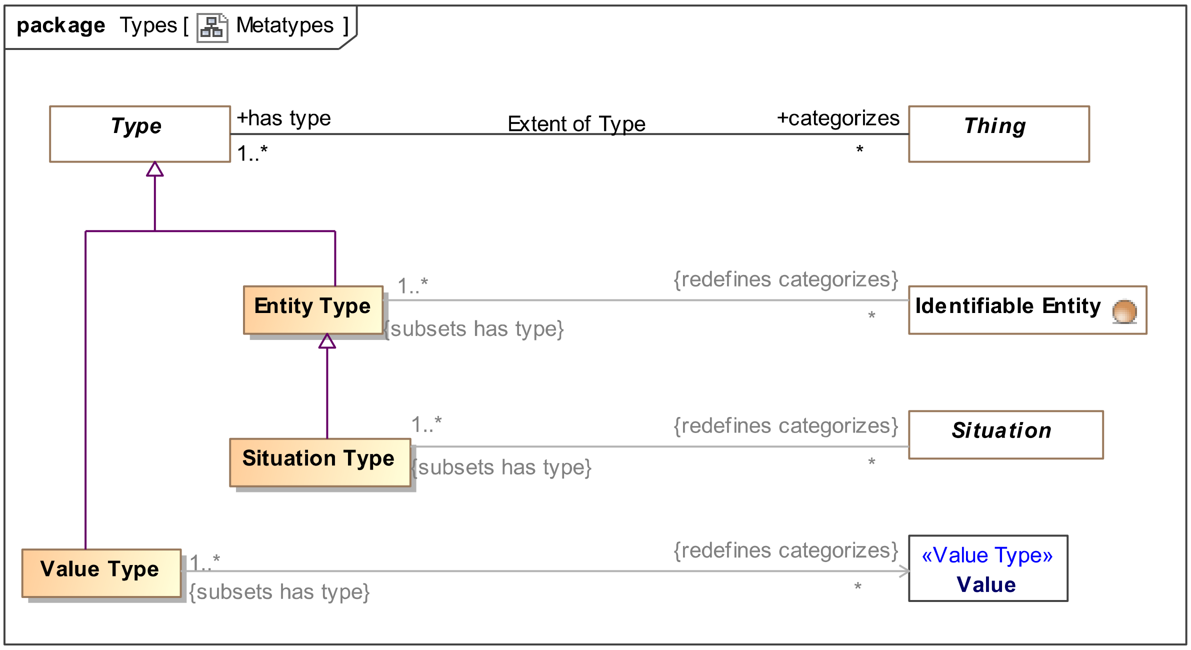


Figure 1.16: Metatypes

Figure 1.16 shows the metatypes (sometimes called "power types") defined in SMIF for the foundation types if Identifiable Entity, Situation and Value. Providing metatypes allows more precise definition of each type and also provides for rules about when and where each can be used.

Note that each metatype "redefines" the kind of thing that the metatype can <categorize>. For example, all Value Types categorize values. In addition, the metatypes are a required type of the type that categorize. For example, each instance of a Value must have at least one type that is a "Value Type".

Each of the SMIF language metatypes has a corresponding stereotype in the UML profile.

Additional metatypes are defined in SMIF and will be introduced in the appropriate sections, below.

1.5.2 Domain Specific Metatypes

Kinds of types can be defined for domain specific needs as well as the language elements such as we have seen above. Domain models typically need to define types or categories of things. "Entity Type" can be specialized for this kind of domain specific need.

Example

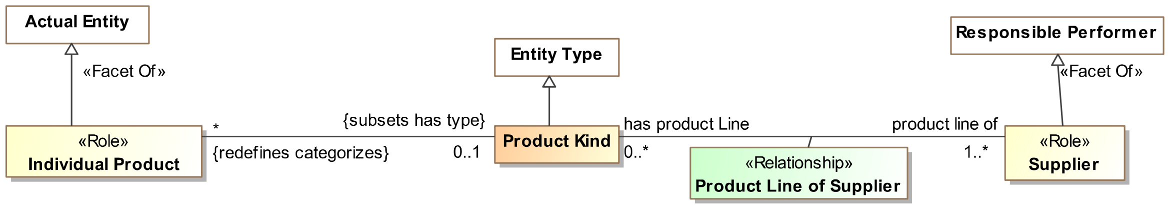


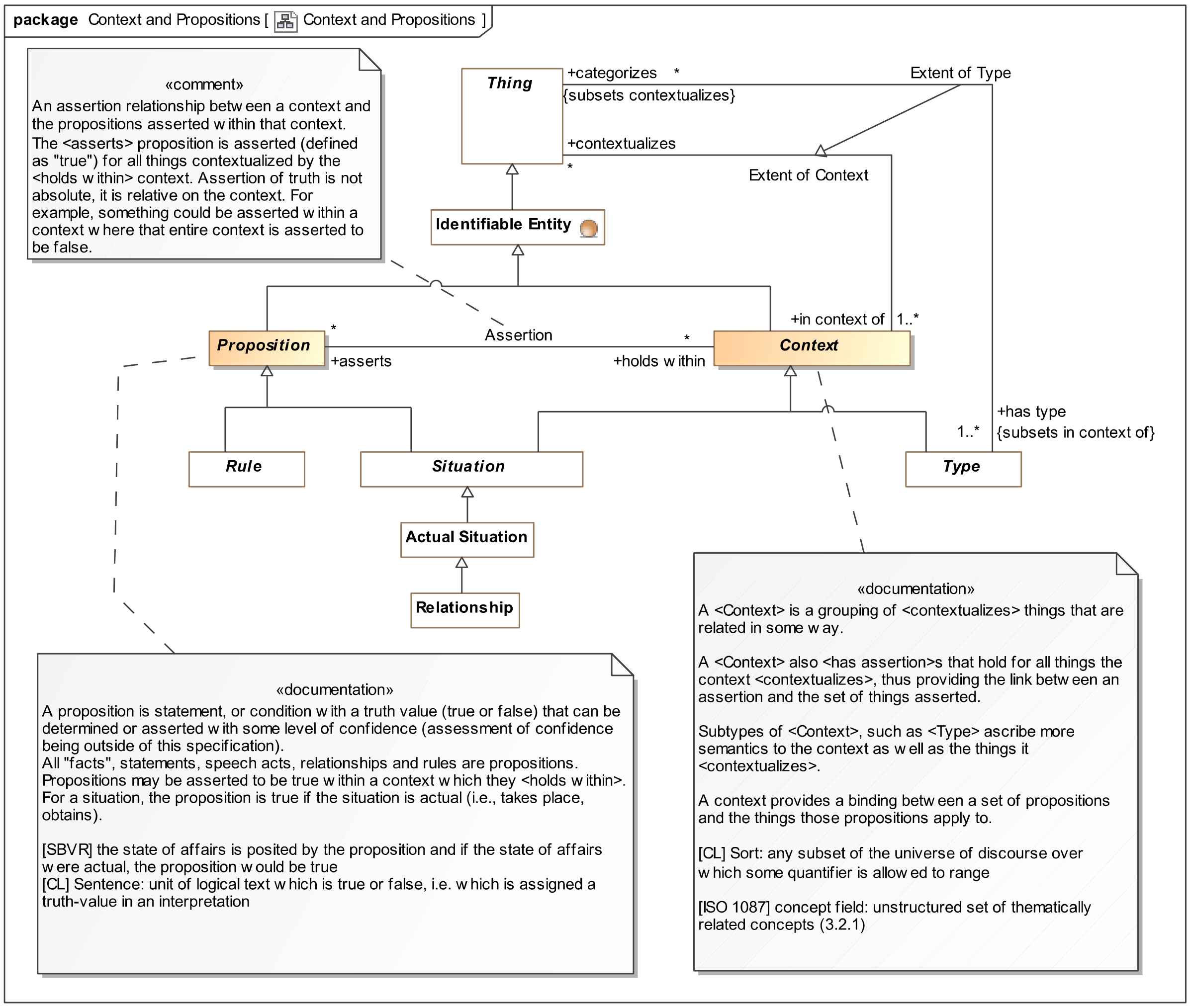
Figure 1.17: Domain Specific Metatype Example

The above example uses some SMIF features we have not yet reviewed, the profile documentation may be consulted as required.

Figure 1.17 defines a "Product Kind" as a subtype of "Entity Kind" - a domain specific metatype. Note that Product Kind redefines what it categorizes to be ab "Individual Product" (being a product is a role of an actual entity). We can now crate a relationship between a supplier and a product kind to represent that the suppler offers such a kind of product as a product line. Using existing concepts of typing and categorization in this way alleviates the need for domain models to "re invent" categorization mechanisms and provides for deeper semantics of what such categorization means.

1.6 Context and Propositions

Note that situations are subtypes of “Context” and “Proposition”. To understand the “inside” of situations these need some explanation. This section may be a bit of a challenge, but take time to understand it as these are essential concepts that form the foundation of **semantics** in SMIF.



«comment»
An assertion relationship betw een a context and the propositions asserted w ithin that context. The <asserts> proposition is asserted (defined as "true") for all things contextualized by the <holds w ithin> context. Assertion of truth is not absolute, it is relative on the context. For example, something could be asserted w ithin a context w here that entire context is asserted to be false.

«documentation»
A proposition is statement, or condition w ith a truth value (true or false) that can be determined or asserted w ith some level of confidence (assessment of confidence being outside of this specification).
All "facts", statements, speech acts, relationships and rules are propositions.
Propositions may be asserted to be true w ithin a context w hich they <holds w ithin>.
For a situation, the proposition is true if the situation is actual (i.e., takes place, obtains).

[SBVR] the state of affairs is posited by the proposition and if the state of affairs w ere actual, the proposition w ould be true
[CL] Sentence: unit of logical text w hich is true or false, i.e. w hich is assigned a truth-value in an interpretation

«documentation»
A <Context> is a grouping of <contextualizes> things that are related in some w ay.

A <Context> also <has assertion>s that hold for all things the context <contextualizes>, thus providing the link betw een an assertion and the set of things asserted.

Subtypes of <Context>, such as <Type> ascribe more semantics to the context as w ell as the things it <contextualizes>.

A context provides a binding betw een a set of propositions and the things those propositions apply to.

[CL] Sort: any subset of the universe of discourse over w hich some quantifier is allow ed to range

[ISO 1087] concept field: unstructured set of thematically related concepts (3.2.1)

Figure 1.18: Context and Propositions

Propositions are anything to which a “truth value” can be assigned, even a probability of truth (probability is not defined within the SMIF language but can be added by augmentation in related reference models). Being able to be assigned a truth value does not make something true or even asserted. The assertion of a proposition is relative to a context it <holds within>.

But, context of what? What set of things does the assertion apply to? A context <contextualizes> any number of things; within a context the things it <contextualizes> are bound by what the context <asserts>. Context is the link between propositions and those things the propositions apply to. The context becomes the *interpretation* of the proposition. Since a thing is <in context of> any number of context that then <asserts> some set of propositions the contextualized interpretation of any thing can be established. In summary, a context <asserts> propositions for the things it <contextualizes>.

Note that Situation is both a proposition (it is something that may or may not be true) and a context (it asserts some configuration of things, defined by other propositions). Later we will see how relationships, characteristics and ultimately “Property Bindings” bottom out this recursive loop.

Examples of context include a document (as it asserts statements within that document), a political authority such as a state or country, a query, a process or a condition. My Coffey cup on my desk is a situation, my weight at any particular time, the solar system, the SI system of units, the lifetime of George Washington, Etc.

Besides situations, propositions also include rules and “universal truths”, like 2 > 1. Rules can be natural (the conversion factor of weight to mass on the surface of the earth) or asserted by authority (no radar detectors can be used in Virginia). Note that certain conversion factors from weight to mass <holds within> the context of the surface of the earth, this is the context of those conversions.

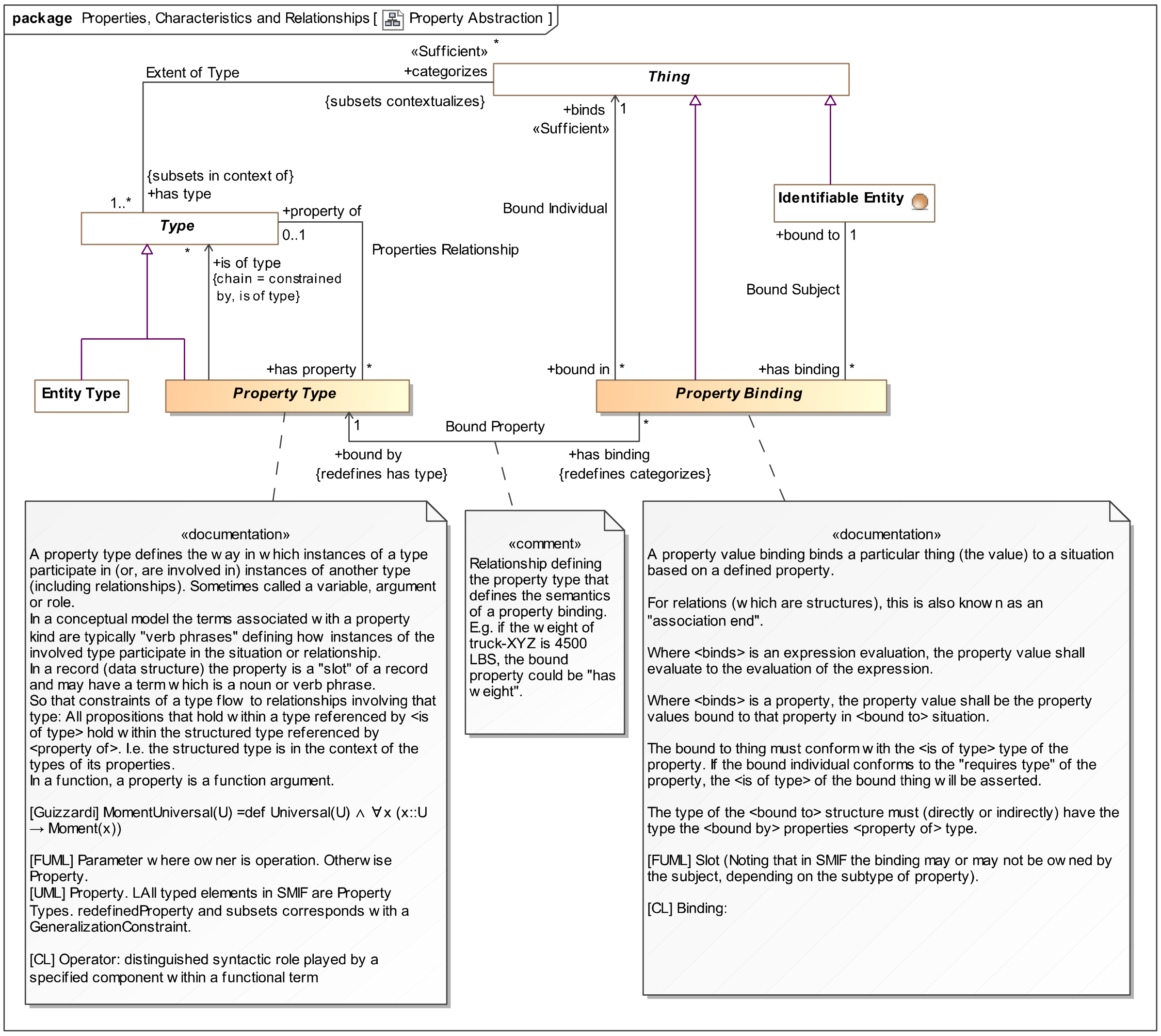
We previously noted the “Extent of Type” relationship between a type and the things it categorizes. Type is a special form of Context and Extent of type is a specialized form of “Extent of Context”. Type is one way of asserting propositions on things, things <categorized> by that type are in context of that type.

--thinking about example--

1.7 Properties, Characteristics and Relationships

1.7.1 Property Abstraction

Many of our concepts deal with variant parts. The weight of something physical, the buyer and seller of a purchase, the cells of a DBMS record, the arguments of a function. We tend to call these properties, variables, arguments, or association ends or fields or parts. SMIF defines an abstraction that provides for these “thing with variant” situations; Property Types and Property Bindings. We will introduce the abstractions first and then the concrete uses of the abstractions.



«documentation»
A property type defines the way in which instances of a type participate in (or, are involved in) instances of another type (including relationships). Sometimes called a variable, argument or role.
In a conceptual model the terms associated with a property kind are typically "verb phrases" defining how instances of the involved type participate in the situation or relationship.
In a record (data structure) the property is a "slot" of a record and may have a term which is a noun or verb phrase.
So that constraints of a type flow to relationships involving that type: All propositions that hold within a type referenced by <is of type> hold within the structured type referenced by <property of>. I.e. the structured type is in the context of the types of its properties.
In a function, a property is a function argument.

[Guizzardi] MomentUniversal(U) =def Universal(U) ∧ ∀x (x::U → Moment(x))

[FUML] Parameter where owner is operation. Otherwise Property.
[UML] Property. LAll typed elements in SMIF are Property Types. redefinedProperty and subsets corresponds with a GeneralizationConstraint.

[CL] Operator: distinguished syntactic role played by a specified component within a functional term

«comment»
Relationship defining the property type that defines the semantics of a property binding. E.g. if the weight of truck-XYZ is 4500 LBS, the bound property could be "has weight".

«documentation»
A property value binding binds a particular thing (the value) to a situation based on a defined property.

For relations (which are structures), this is also known as an "association end".

Where <binds> is an expression evaluation, the property value shall evaluate to the evaluation of the expression.

Where <binds> is a property, the property value shall be the property values bound to that property in <bound to> situation.

The bound to thing must conform with the <is of type> type of the property. If the bound individual conforms to the "requires type" of the property, the <is of type> of the bound thing will be asserted.

The type of the <bound to> structure must (directly or indirectly) have the type the <bound by> properties <property of> type.

[FUML] Slot (Noting that in SMIF the binding may or may not be owned by the subject, depending on the subtype of property).

[CL] Binding:

Figure 1.19: Definition of Property Type & Property Binding

Property Type and Property Binding form a special type-instance relationship. A property binding provides <binds> a value for a <bound by> property type within the identifiable entity it is <bound to> . This is similar to the idea of a “triple” in [RDF]. The Property Type defines the meaning of these properties bindings for the type it is a <property of>. As such, the property binding is an instance of its property type. Recognizing properties as types allows us to use the same type/instance and type hierarchy tools we have seen for entities with properties.

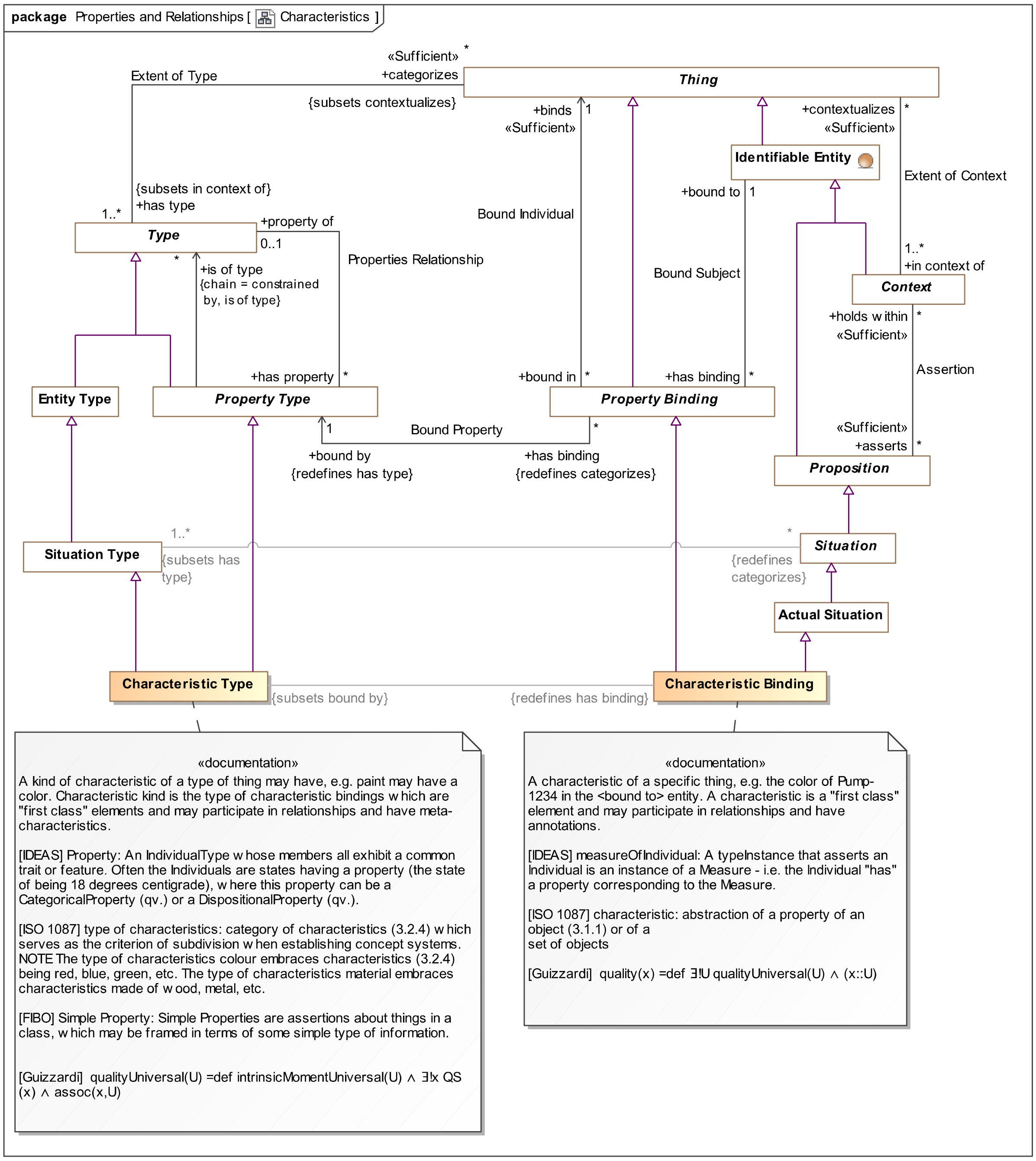
A Property type is a <property of> some type, corresponding to the “domain” of the property in [RDF]. Property of constrains the types of entities that a property binding can be <bound to>. Likewise, a property <is of type> a type that a property binding <binds> to that corresponds with the range of a property in [RDF]. Note that <is of type> is defined by a “chain” through a rule. We will define these rules in more detail below.

The following sections show how the concrete subtypes of property and property binding are used.

1.7.2 Characteristics

Characteristics are something inherent in something else that describe a quality of that thing and help differentiate that thing from other things. Other terms are “property” or “attribute”. There are characteristic types and characteristic bindings. Typical examples would be the weight of a person or the color of a ball. Characteristics correspond to a reified property in [RDF] but may be interpreted as a simple [RDF] triple if context or time-variant capabilities are not required.

Characteristics should be used when the property type directly adheres in the entity type, there is no intervening relationship or structure. Where these is “something in the middle” between two things a “Relationship” should be used. Relationships are described in section 1.7.3. This familiar with RDF or OWL may be used to defining properties in “pairs” that have an “inverse”. Where there are or could be such pairs, “Relationship” is the right construct to use in SMIF.



«documentation»
A kind of characteristic of a type of thing may have, e.g. paint may have a color. Characteristic kind is the type of characteristic bindings w hich are "first class" elements and may participate in relationships and have meta-characteristics.

[IDEAS] Property: An IndividualType w hose members all exhibit a common trait or feature. Often the Individuals are states having a property (the state of being 18 degrees centigrade), w here this property can be a CategoricalProperty (qv.) or a DispositionalProperty (qv.).

[ISO 1087] type of characteristics: category of characteristics (3.2.4) w hich serves as the criterion of subdivision w hen establishing concept systems. NOTE The type of characteristics colour embraces characteristics (3.2.4) being red, blue, green, etc. The type of characteristics material embraces characteristics made of w ood, metal, etc.

[FIBO] Simple Property: Simple Properties are assertions about things in a class, w hich may be framed in terms of some simple type of information.

[Guizzardi] qualityUniversal(U) =def intrinsicMomentUniversal(U) ∧ ∃!x QS (x) ∧ assoc(x,U)

«documentation»
A characteristic of a specific thing, e.g. the color of Pump-1234 in the <bound to> entity. A characteristic is a "first class" element and may participate in relationships and have annotations.

[IDEAS] measureOfIndividual: A typeInstance that asserts an Individual is an instance of a Measure - i.e. the Individual "has" a property corresponding to the Measure.

[ISO 1087] characteristic: abstraction of a property of an object (3.1.1) or of a
set of objects

[Guizzardi] quality(x) =def ∃!U qualityUniversal(U) ∧ (x::U)

Figure 1.20: Definition of Characteristic & Characteristic Kind

Note that “Characteristic Binding” is an “Actual Situation”. This makes Characteristic bindings – the weight of the person or the color of the ball, subject to context and time (as a temporal entity) – so the *same entity* could have *different values* for the *same characteristic type* at *different times or from different sources*. The expectation is that the type of characteristics will be a value type, but to allow for diversity in approaches, this is a recommendation, not a rule.

Examples

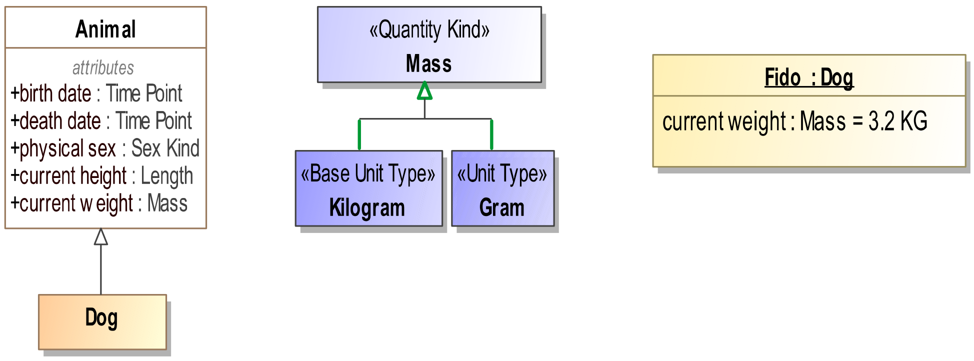


Figure 1.21: Defining and Using a Characteristic

Figure 1.21 shows the definition and use of a characteristic we have seen above. Note that in the conceptual reference model we have used "Mass" as the type of weight. This is to allow for the many different units and representations of mass that may be used in various data sources. However an actual mass value, such as the weight of Fido, must use some concrete unit, in this case Kilograms. This separation of the abstract "Quantity Kind" from a specific system of units is considered best practice for conceptual reference models. SMIF machinery is then able to comprehend, integrate and translate between various units of the same Quantity Kind. The concept "Quantity Kind" is derived from the [JCGM 200:2008] standard and is a part of the SMIF language. Specific quantity kinds and units, such as Mass and Kilogram, are defined in reference models that use SMIF like [ThreatRisk] and [FIBO].

We can now explore the representation of these concepts in terms of the SMIF conceptual model. In this example we will add the fact that this weight was valid during the year 2005 as defined by an ISO date.

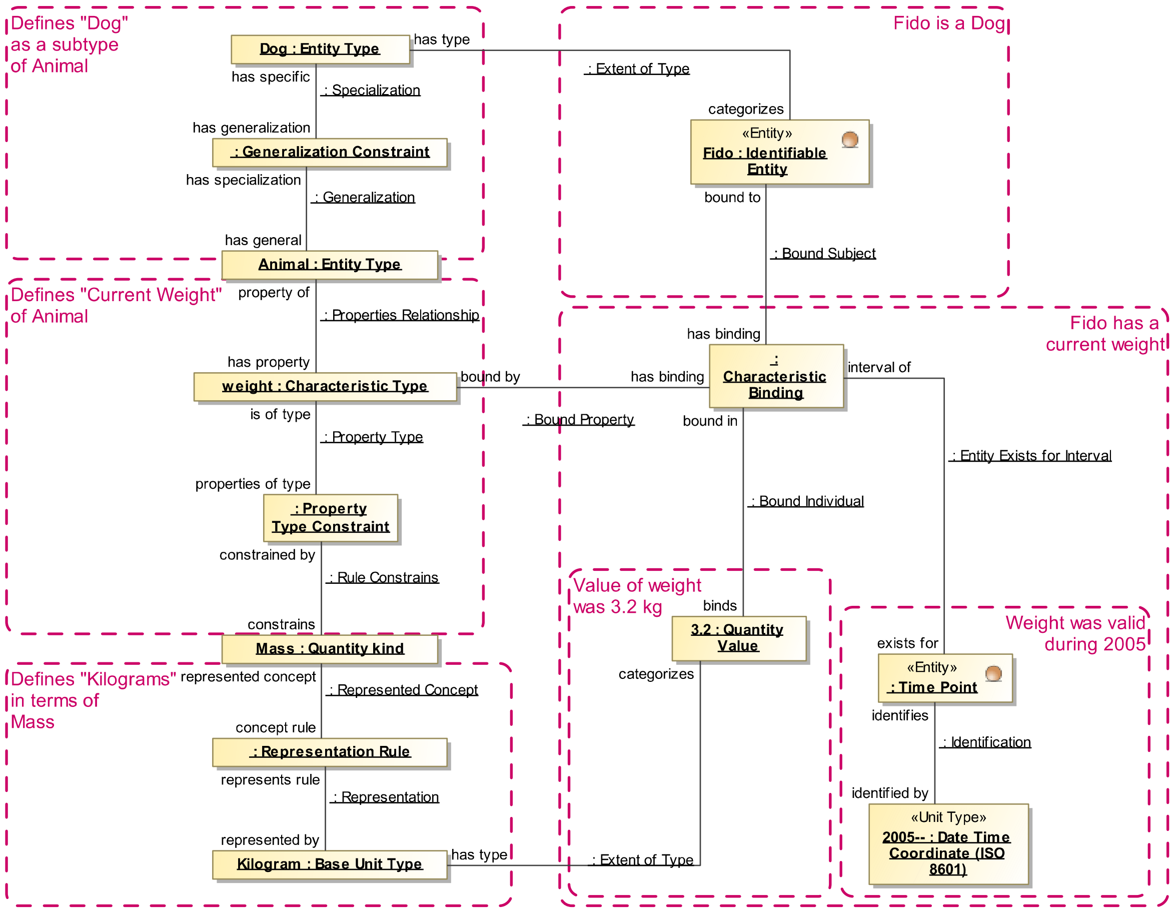


Figure 1.22: Instance Model Defining Characteristic & Value for an Entity

Figure 1.22 is an example instance model showing the definition of “Animal” with a “ weight” property, the definition of “Mass” and it’s unit “Kilograms” on the left. On the right is “Fido” having type “Dog” and a property value of “ weight” being 3.3 kg during 2005.

This model uses some rules not yet defined: Generalization Constraint, Property Type Constraint and Representation Rule – you may refer to the reference section for details on these rules. Time point and Date Time Coordinate come from the [ThreatRisk] example model. Rules are used rather than simple relationships to allow for these constraints to be specified in context other than the defining ones – providing for an “open world assumption”.

Focusing on the definition of a characteristic – weight, there is a Characteristic Type (named “weight”) that is a <property of> an Animal (an entity type). This property is constrained to have a value of type “Mass” (a quantity kind). Mass can be <represented by> “Kilogram” (a Unit Type). Dog (an entity type) is a subtype of “Animal”.

Focusing on “Fido”; Fido <has type> Dog and one characteristic is shown here as an unnamed characteristic binding, <has binding> that is <bound by> weight and <binds> 3.2 kg as the value. This characteristic binding <exists for> (is valid for) 2005 as defined by an ISO date. Other bindings of weight for Fido could be represented across other time points or time intervals. We could also attach “source” and confidence information to these characteristics to aid in evaluating its trustworthiness.

1.7.3 Relationships

Relationships, and the corresponding Relationship Type, can be considered part of the foundation of the SMIF language. In fact, many SMIF conceptual reference models could be thought of as “relationship oriented” rather than “object oriented”. This is because much of the semantics of a domain is captured in how things relate.

Relationships in SMIF are considered “first class” entities, or as described in [Guizzardi2015] “Full-Fledged Endurants” where “a relationship is the particular way a relation holds for a particular set of relata”. This means they have their own meaning, identity and life-cycle. Relationship types can specialize other relationship types. Relationships can have characteristics and participate in other relationships – of particular importance are other relationships that define when a subject relationship is valid or when it is not. While a relationships as a “two ended line” is the most common, relationships can have any number of “ends” that relate involved things. This is called an “n-ary” relationship in the literature.

While relationship instances may become “true or false” in certain time-frames or context, each relationship instance is considered atomic and invariant. That is; the “ends” of the relationship instance never change. For example, if we have the relationship “John is located in Virginia”, we could say this is true from 2012-2016 but we would never change “John” or “Virginia” *for that relationship instance*. If we wanted to say “John is located in Mexico”, that would be another relationship instance with its own life-cycle, perhaps in 2017. This allows us to “track” John over time or to just consider where John is right now. It also allows us to attach metadata to each relationship instance, for example, who said that John is located in Mexico in 2017? By recognizing relationships as first-class situations and temporal entities, SMIF provides a way to account for time and change over time – this is known as “4D” in semantic literature, where the 4th dimension is time. Relationships can also be used as more of a “snapshot in time” where 4D is not of concern. Relationships with no time constraints or time-dependent context are considered to be true indefinitely.

The “ends” of a relationship are represented as properties. Each property defines a related thing. The naming convention we use in SMIF is that these ends are named as verb phrases that are the view of an end *from the other ends* as we will see in the examples.

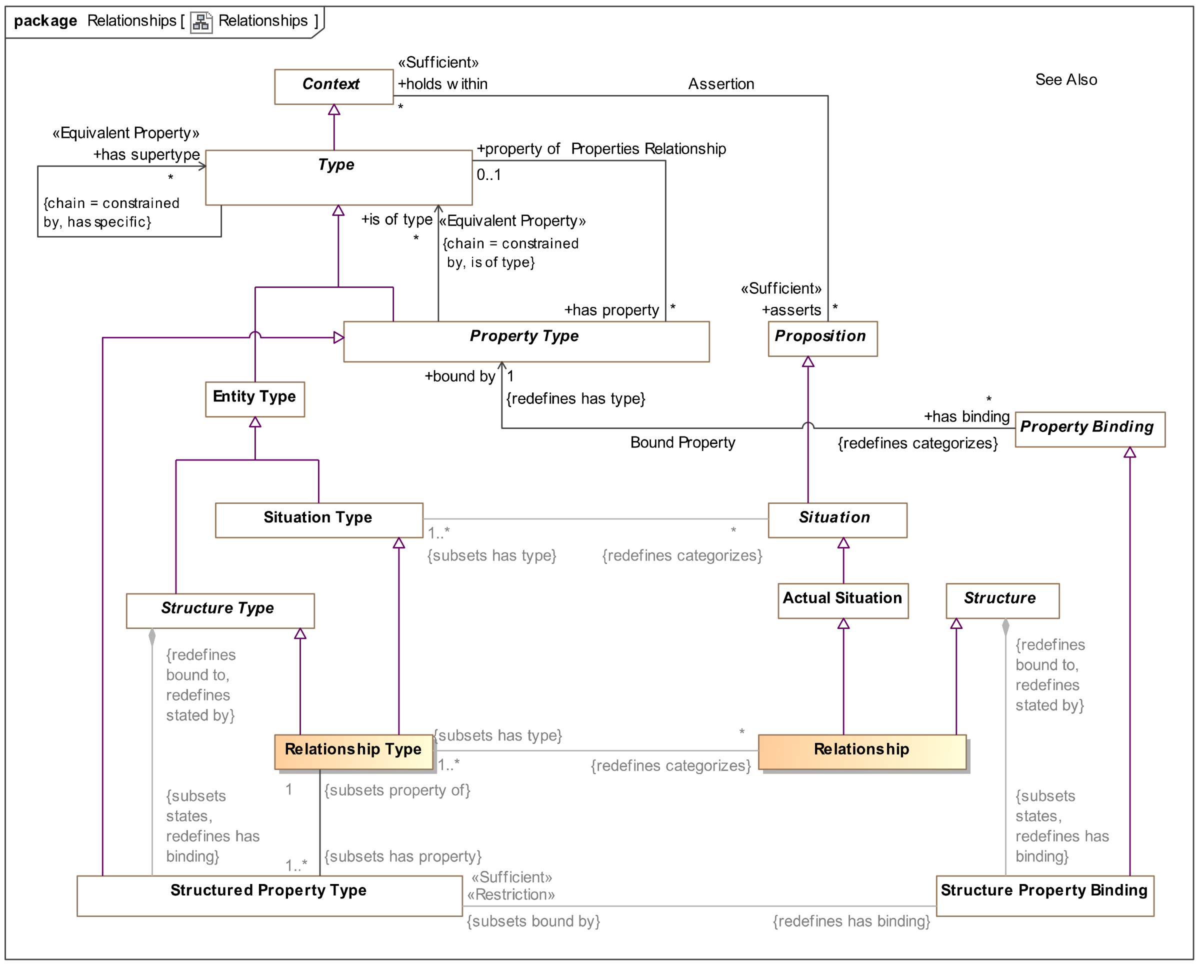


Figure 1.23: Defining Relationships

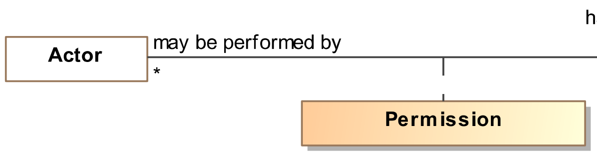
Figure 1.23 shows the SMIF conceptual model defining relationships, building on concepts we have already seen such as situations and properties.

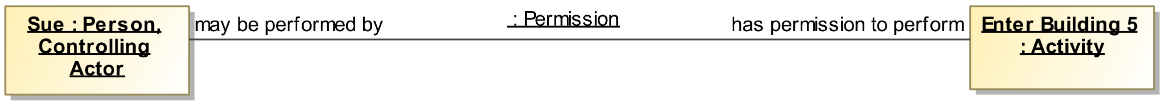
Relationship types build on “structures” and “structured properties” as the ends of relationships. In the Characteristics section we saw that each characteristic is an independent situation. On the other hand, structured properties are always “in” something else – in this case a relationship. There is no way to say that a relationship exists in time “A” where as one of its ends exists in time “B” - relationships are an atomic unit. This is why the “Structure Property Binding” is shown as being “owned” by the “Structure”. There are other structures besides relationships, which we will explore later.

A relationship is a special kind of situation involving the related elements, each identified by a binding owned by the relationship. Likewise the Relationship Type is a kind of situation type that has a set of structured property types (it is legal so share structured property types between relationship types).

As we noted above, since relationships are situations you can define other relationships that involve relationships. For example the relationship that Sue possesses Key-card-A8988 which enables her to enter building 5. This possession could be altered by “Theft RC2016-998” which stole that key card.

Example 1




Sue : Person, Controlling Actor
may be performed by
: Permission
has permission to perform
Enter Building 5 : Activity

The “Permission” relationship is defined between an actor and an activity in [ThreatRisk]. We also see an instance of this relationship in the UML profile as “Sue” having permission to “Enter building 5”. Next we will look at the relationship definition and instances in terms of the SMIF conceptual model.

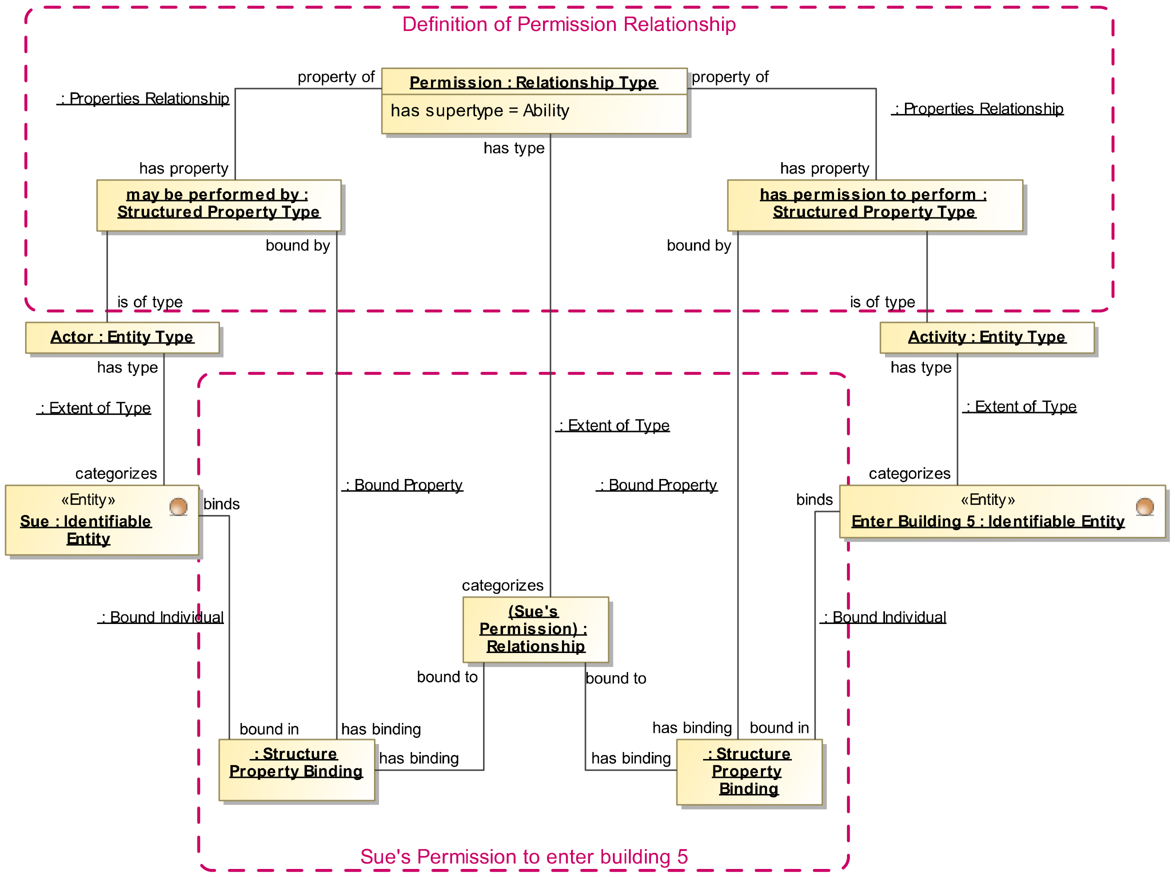

Definition of Permission Relationship
Permission : Relationship Type
has supertype = Ability
property of
property of
: Properties Relationship
: Properties Relationship
has property
has property
has type
may be performed by : Structured Property Type
has permission to perform : Structured Property Type
bound by
bound by
is of type
is of type
Actor : Entity Type
Activity : Entity Type
has type
has type
: Extent of Type
: Extent of Type
: Extent of Type
categorizes
categorizes
«Entity» Sue : Identifiable Entity
«Entity» Enter Building 5 : Identifiable Entity
binds
binds
: Bound Property
: Bound Property
: Bound Individual
: Bound Individual
categorizes
(Sue's Permission) : Relationship
bound to
bound to
bound in
has binding
has binding
bound in
has binding
has binding
: Structure Property Binding
: Structure Property Binding
Sue's Permission to enter building 5

Figure 1.24: Defining and Using a Relationship

Assuming that “Actor” and “Activity” are already defined, we create a new “Relationship Type” with a name of “Permission”. Permission has two “Structured Property Types”: “may be performed by” that <is of type>

"Actor" and "has permission to perform" that <is of type> "Activity". (Note that we are using the "shortcut" property chain "<is of type>, which implies a property type constraint).

To represent an instance of "Permission", giving "Sue" permission to enter building 5 we create a relationship which is an instance of "Permission" which represents (is a sign for) Sue's permission. We will assume that "Sue" and the permission to enter building 5 already exist. Sues' permission relationship has two "Structured Property Bindings": One that binds Sue to "may be performed by" and the other that binds "Enter Building 5" to "has permission to perform". Of course both Sue and "Enter building 5" could be bound in other relationships.

Example 2

Building on the example above, we would like to represent the idea of a "Credential". A credential attests can attest to a permission or other kind of ability.

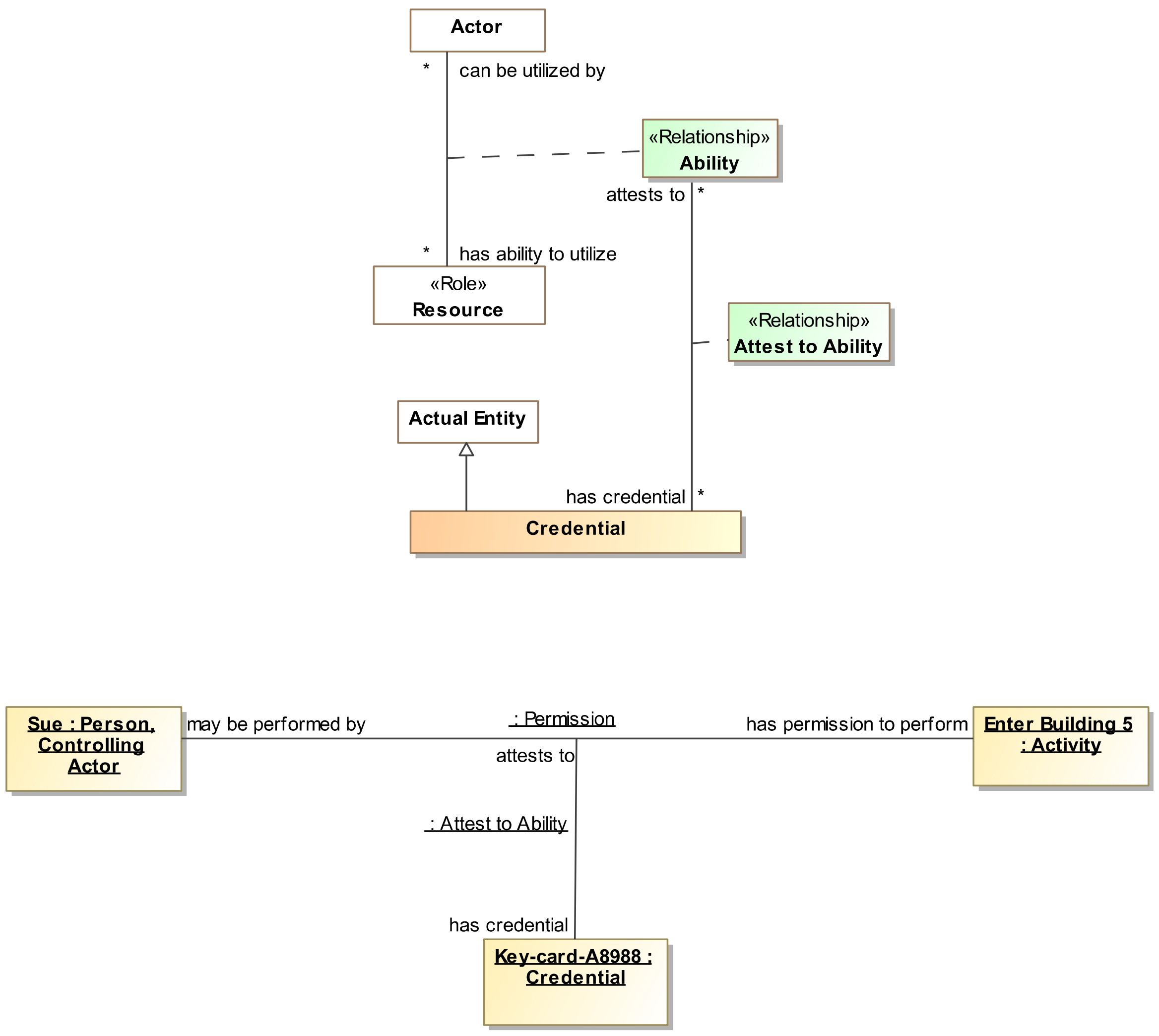


Figure 1.25: Relationship Involving Relationships

A permission like we used above is a subtype of "Ability" (I know it it not in the diagram, trust me). An ability is a relationship between an actor and some resource they can use – in the case above the ability was "Permission" and the resource was an activity. Note that there is a relationship type “Attest to Ability” between a credential and such an ability. A credential <attests to> some ability. This shows how relationships can be “first class” elements and the subject of other relationships.

Below the class model we see the instanced defined above, where “Sue” <has permission to perform> “Enter Building 5”. We also see that “Key-card-A8988” <attests to> this ability in a “Attest to Ability” relationship. Note that the notation used here, UML instance diagrams, is not what we would show to stakeholders – they would most likely see a custom user interface.

We will now look at the above in terms of the SMIF model.

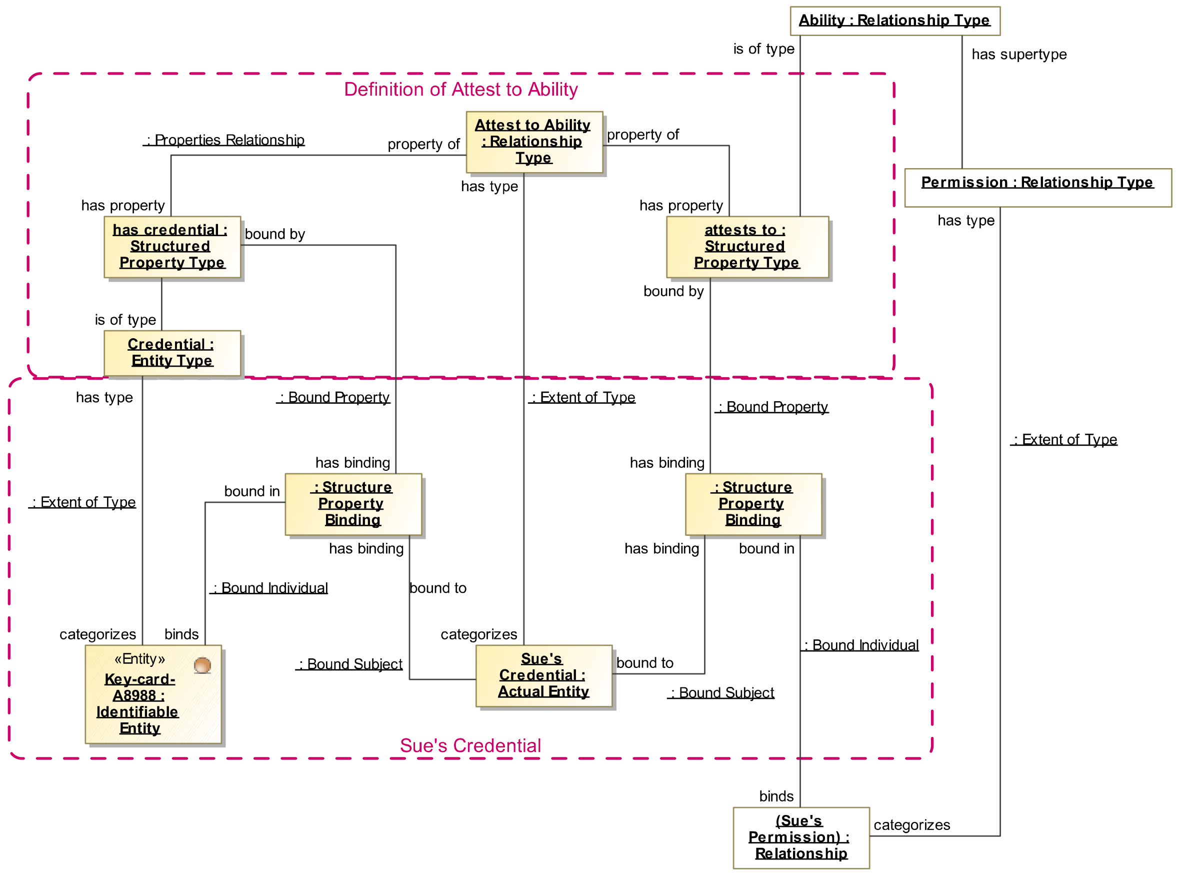


Figure 1.26: Relationship Involving Relationships – instance model

In a pattern very much like the definition and use of “Permission” we see the definition and use of “Attest to Ability”. The interesting addition is that the “<attests to> end of “Attest to Ability” has a type of “Ability”, a relationship type that is a supertype of “Permission”. This allows “Sue’s Credential”, to <attest to> “Sue’s Permission to enter building 5”.

The result of the above is that we have properly represented that sue has a permission as well as a credential for that permission. Consider the additional types and relations that could build on this foundation:

- We could have a “Possession” relationship, representing that Sue is in possession of her credential.
- We could represent an incident where the credential is stolen and possession is transferred to a terrorist, thus providing access to an attacker.
- We could then represent and evaluate various threat scenarios relating to such a stolen credential.