

Table of Contents

1	Introduction to SMIF Semantics.....	3
1.1	<i>The SMIF Conceptual Model Foundation.....</i>	3
1.1.1	Thing.....	4
1.1.2	Type.....	5
1.1.3	Kinds of Types.....	6
1.1.4	Identifiable Entities and Values.....	7
1.1.5	Identifiers.....	8
1.2	Temporal and Actual Entities.....	14
1.3	Situations.....	17
1.4	Context and Propositions.....	18
1.5	Properties, Characteristics and Relationships.....	20
1.5.1	Property Abstraction.....	20
1.5.2	Characteristics.....	21

Table of Figures

Figure 1.1: Thing.....	4
Figure 1.2: Thing has type.....	5
Figure 1.3: Fido is a dog example.....	6
Figure 1.4: UML Type and Instance Example.....	6
Figure 1.5: Identifiable Entities and Values.....	7
Figure 1.6: Identifiable Entity with Value Characteristic.....	8
Figure 1.7: Basic Identifiers.....	8
Figure 1.8: Fully expanded type and identifier instance model.....	9
Figure 1.9: Unique Identifiers.....	10
Figure 1.10: Full Identifiers Package.....	12
Figure 1.11: Full Identifier Example.....	13
Figure 1.12: Temporal and Actual Entities.....	14
Figure 1.13: Actual Thing Hierarchy Example.....	15
Figure 1.14: Temporal Instance Example.....	16
Figure 1.15: Situations - Top Level.....	17
Figure 1.16: Context and Propositions.....	18
Figure 1.17: Definition of Property Type & Property Binding.....	20
Figure 1.18: Definition of Characteristic & Characteristic Kind.....	22
Figure 1.19: Defining and Using a Characteristic.....	23
Figure 1.20: Instance Model Defining Characteristic & Value for an Entity.....	24

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 *The 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 concepts 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

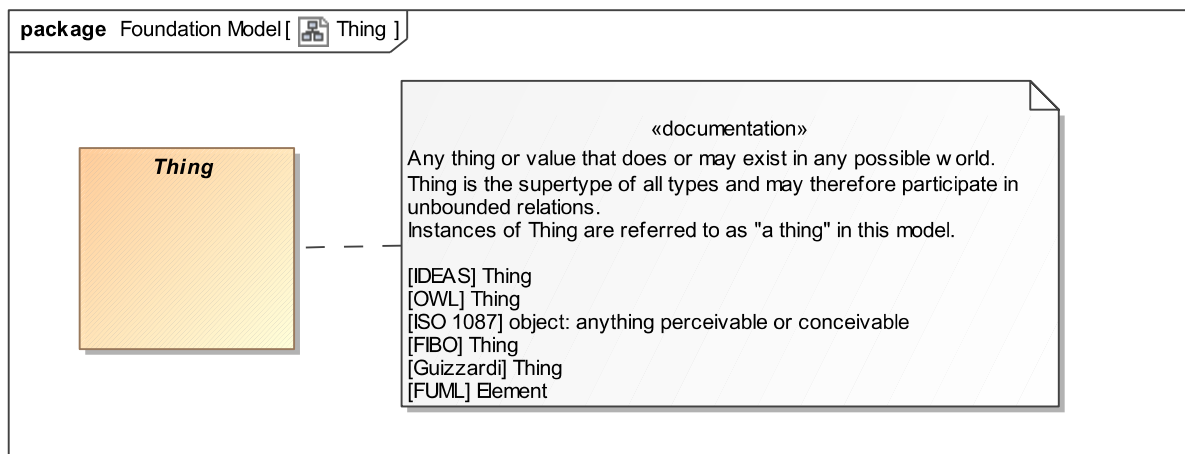


Figure 1.1: Thing

In many models it is convenient to have a sign for anything that could possibly be in any world view, any data repository or any model – the most general concept possible and therefore 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

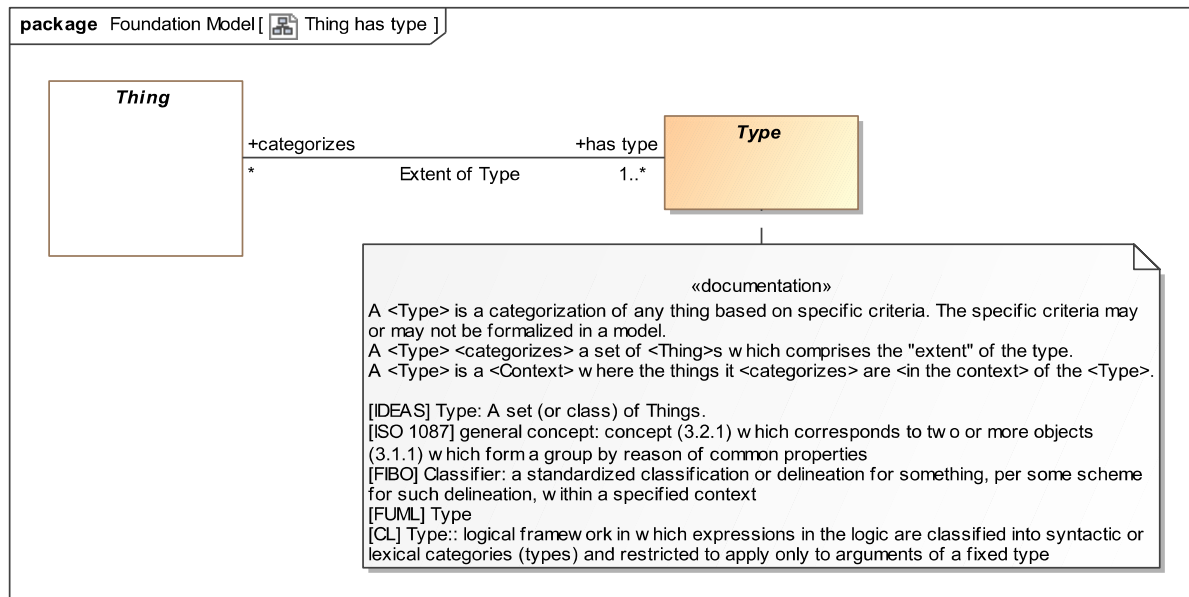


Figure 1.2: Thing has type

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 references, e.g. [FUML] to other standards with like concepts.

Examples

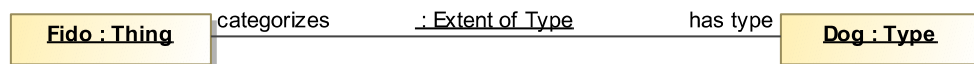


Figure 1.3: Fido is a dog example

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 we 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:

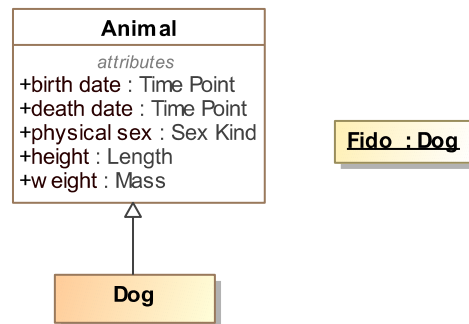


Figure 1.4: UML Type and Instance Example

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 Kinds of Types

--todo--

1.1.4 Identifiable Entities and Values

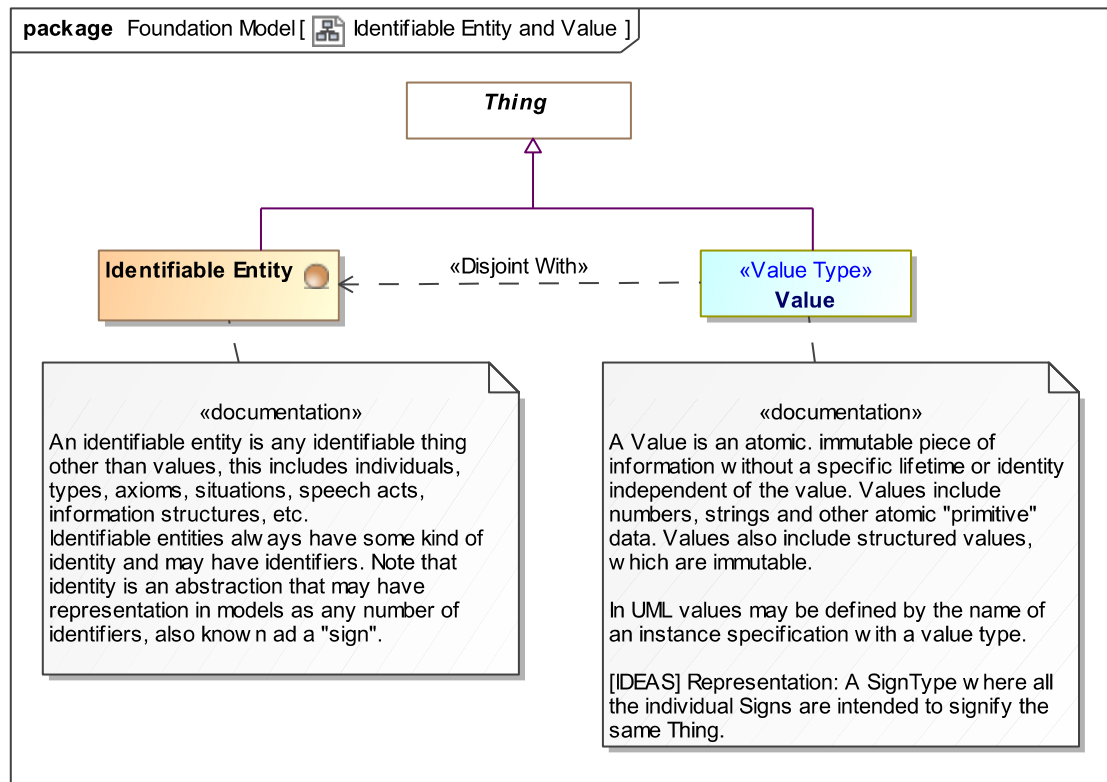


Figure 1.5: Identifiable Entities and Values

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

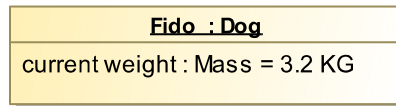


Figure 1.6: Identifiable Entity with Value Characteristic

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.

1.1.5 Identifiers

1.1.5.1 Basic Identifiers

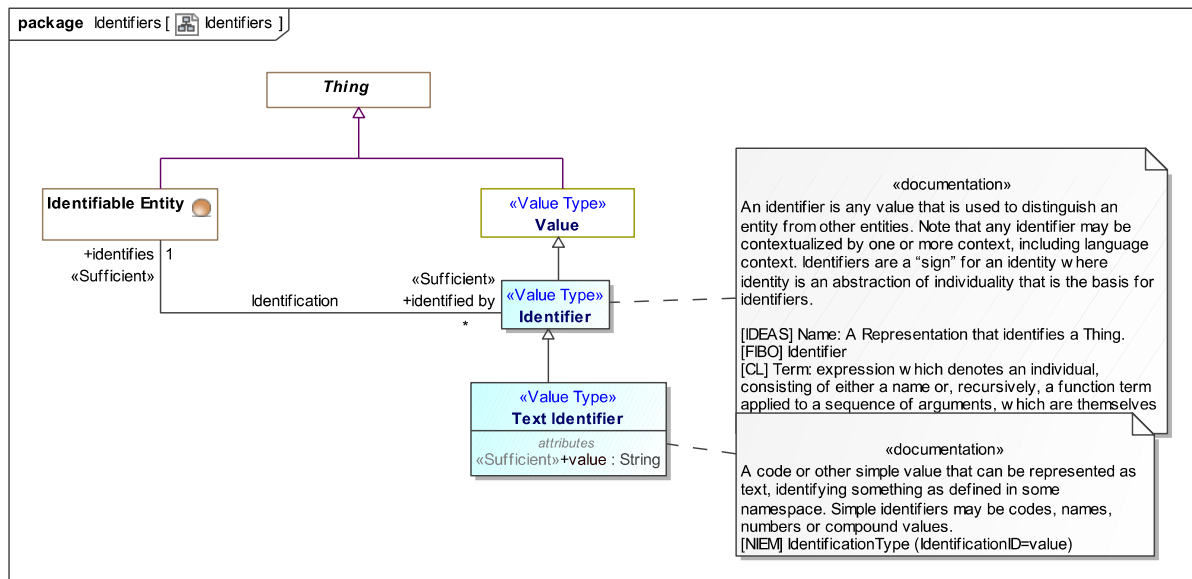


Figure 1.7: Basic Identifiers

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:

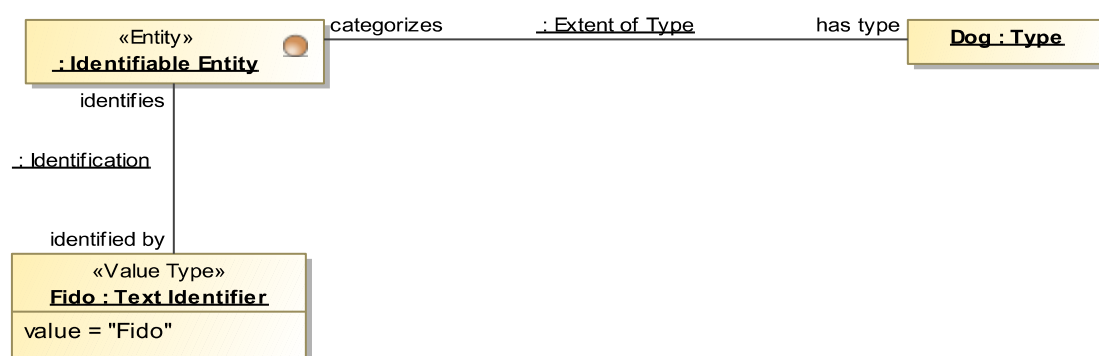


Figure 1.8: Fully expanded type and identifier instance model

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.

1.1.5.2 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.

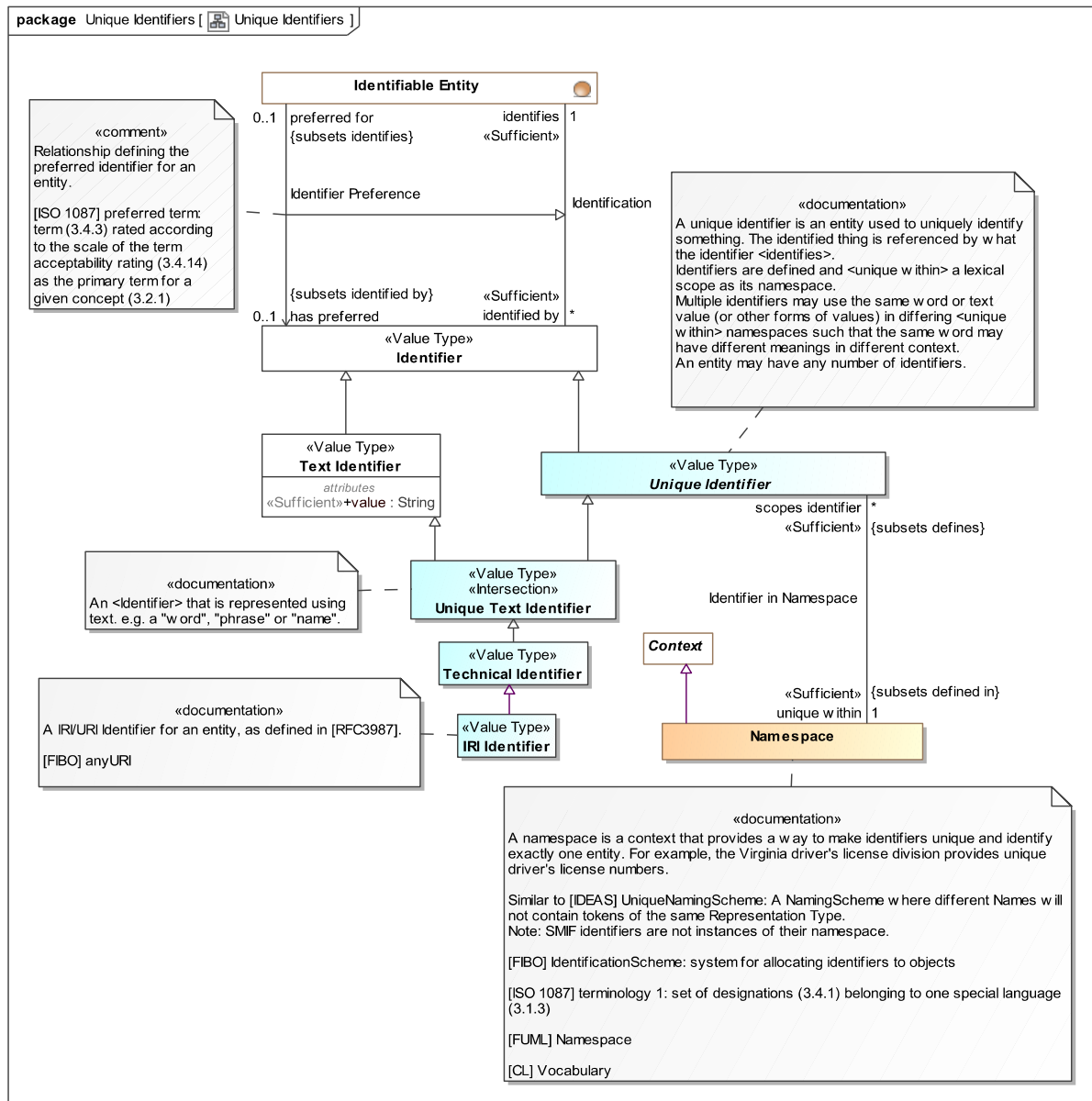


Figure 1.9: Unique Identifiers

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.

1.1.5.3 Names and Terms

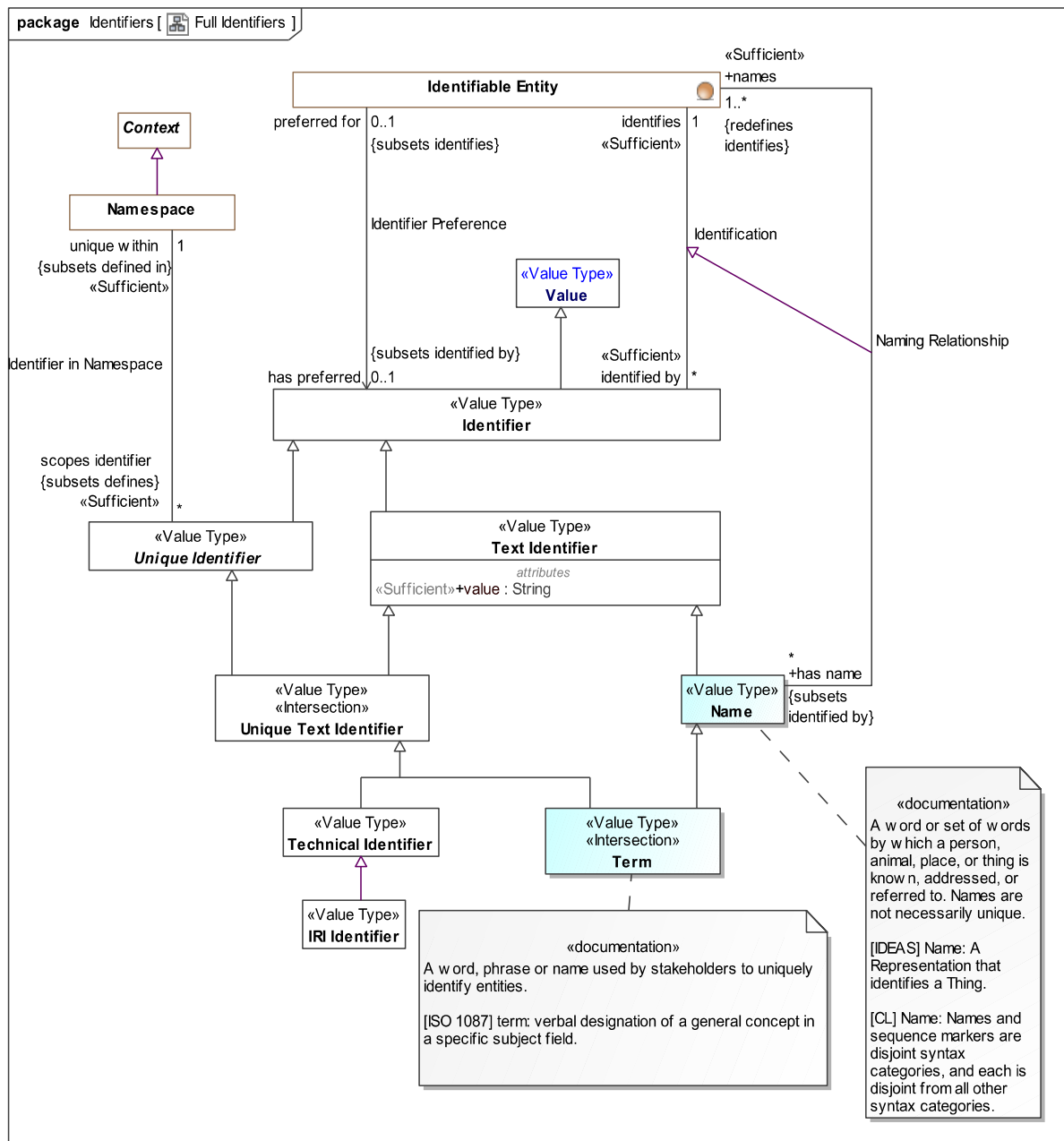


Figure 1.10: Full Identifiers Package

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.

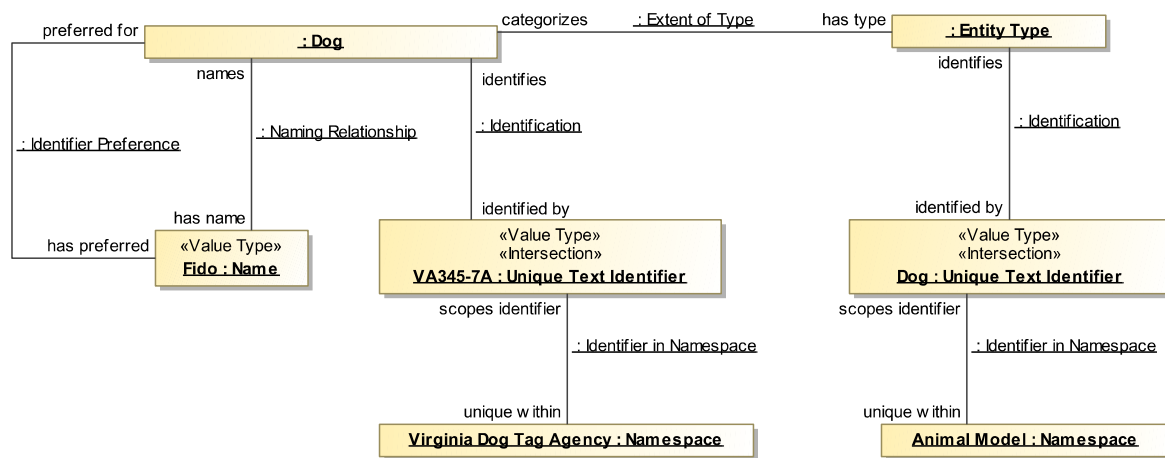


Figure 1.11: Full Identifier Example

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.

1.2 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.

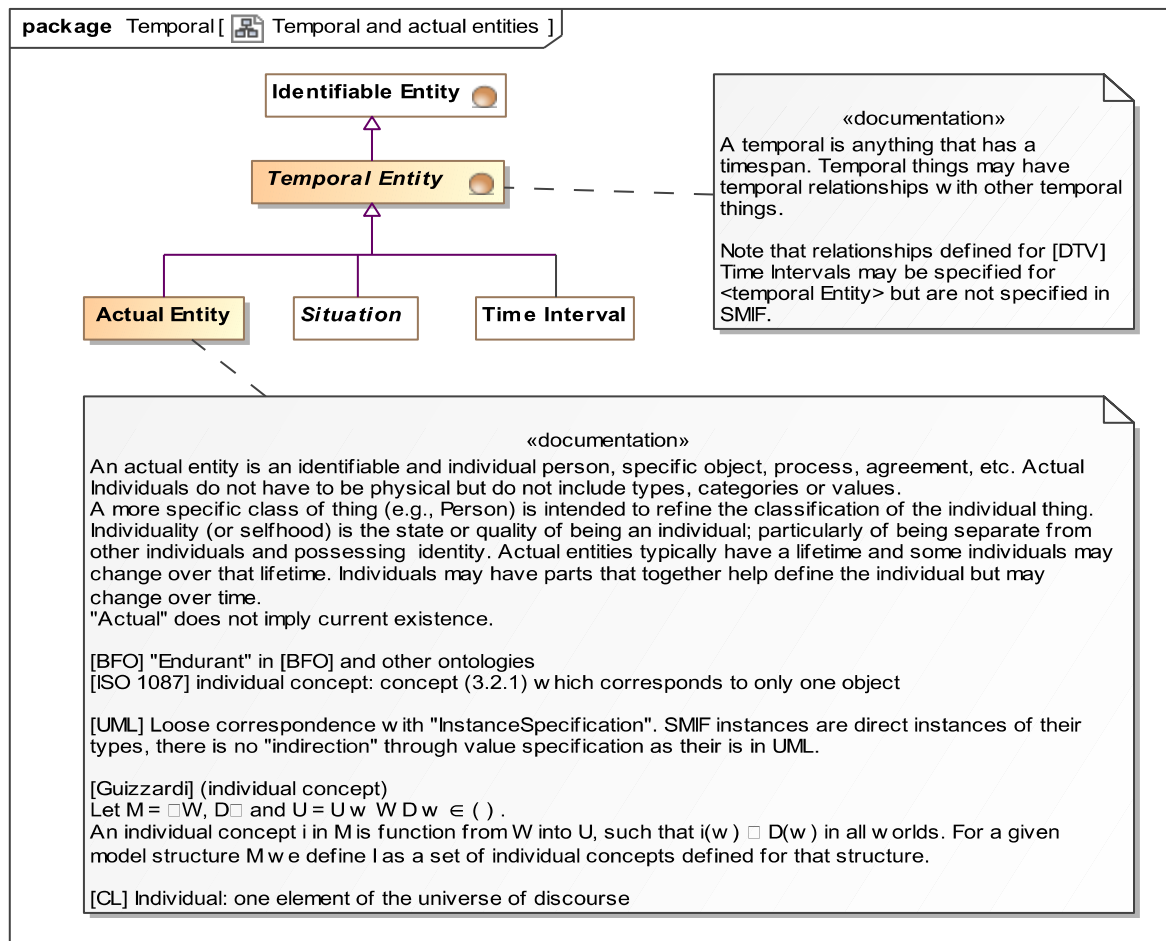


Figure 1.12: Temporal and Actual Entities

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 does 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

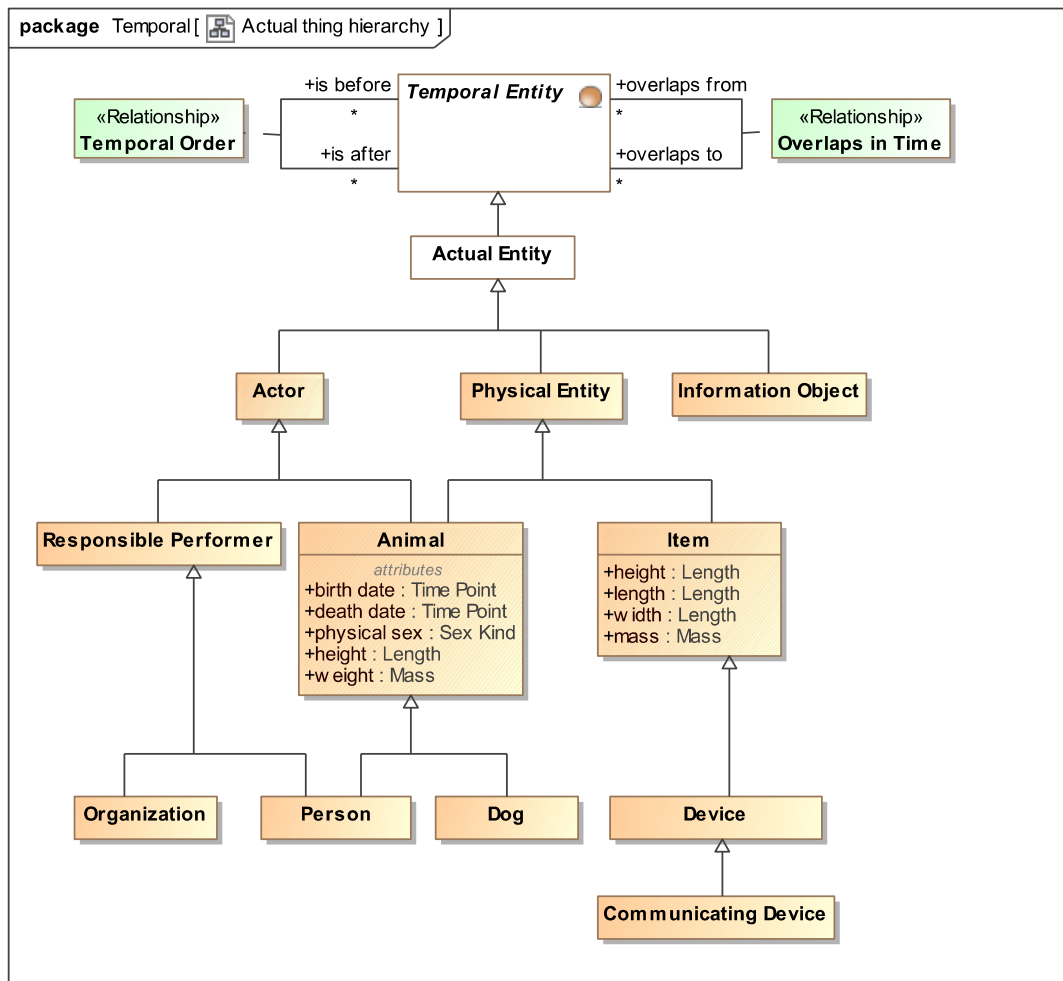


Figure 1.13: Actual Thing Hierarchy Example

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*.

I hear someone screaming - “*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 your 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:

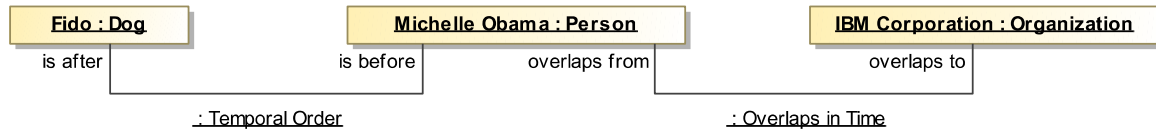


Figure 1.14: Temporal Instance Example

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.

1.3 Situations

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.

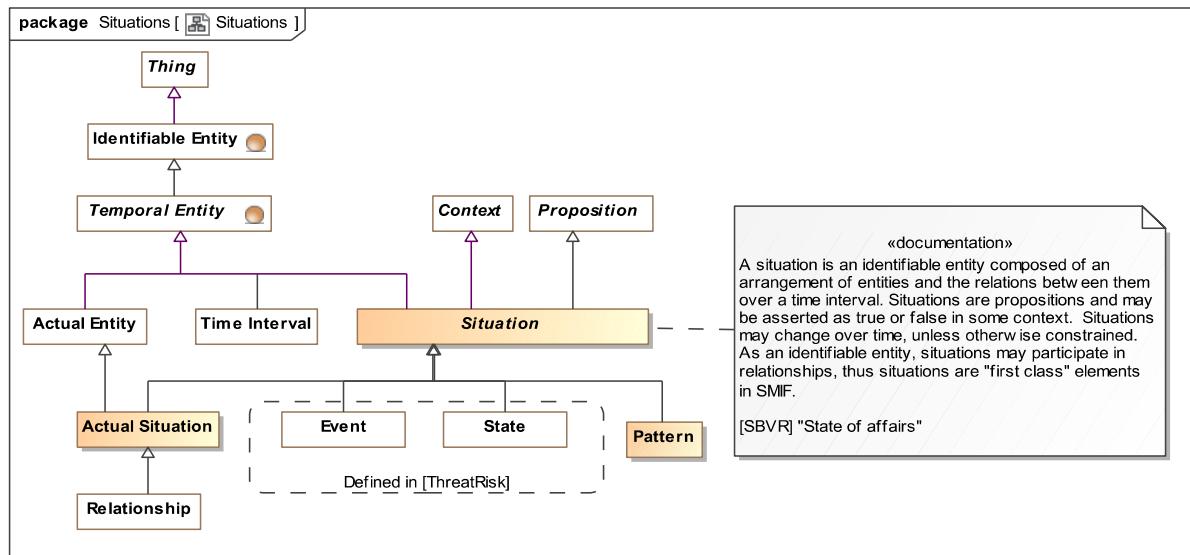


Figure 1.15: Situations - Top Level

So what are situations? A terrorist entering an airport. A policeman in an airport. 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.

1.4 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 the time to understand it as these are essential concepts that form the foundation of **semantics** in SMIF.

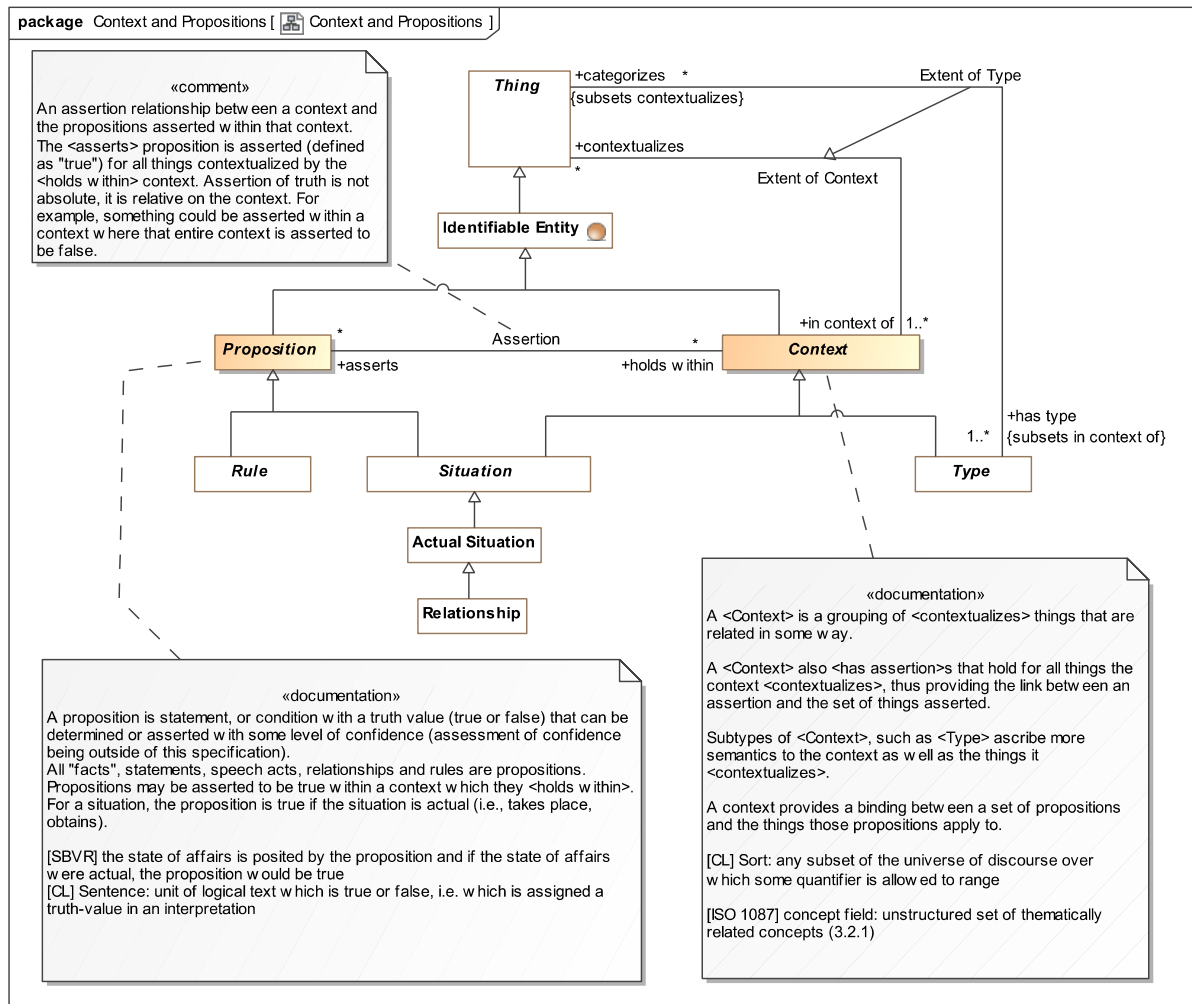


Figure 1.16: Context and Propositions

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.

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.

1.5 Properties, Characteristics and Relationships

1.5.1 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.

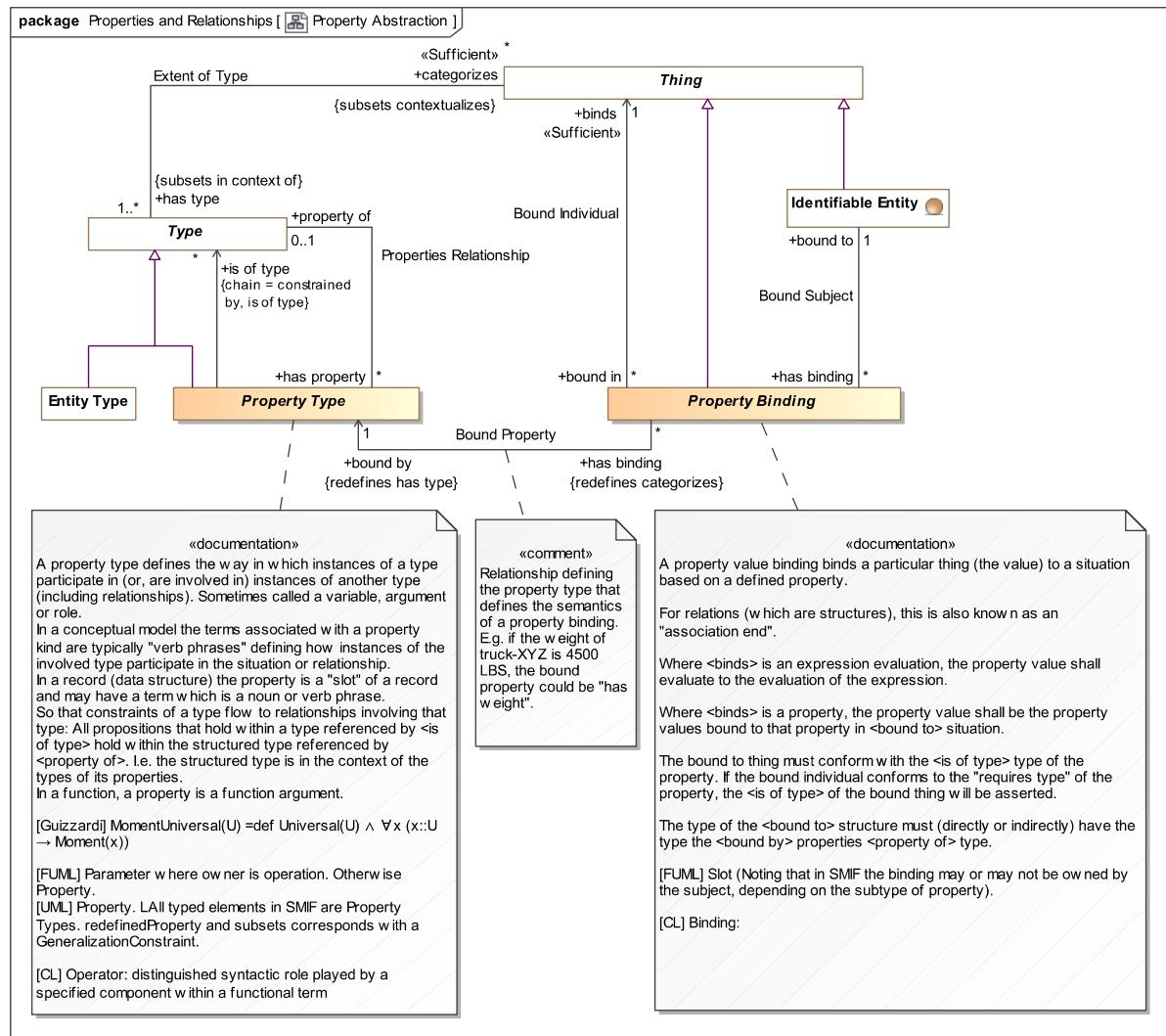


Figure 1.17: Definition of Property Type & Property Binding

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.

1.5.2 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.

Examples

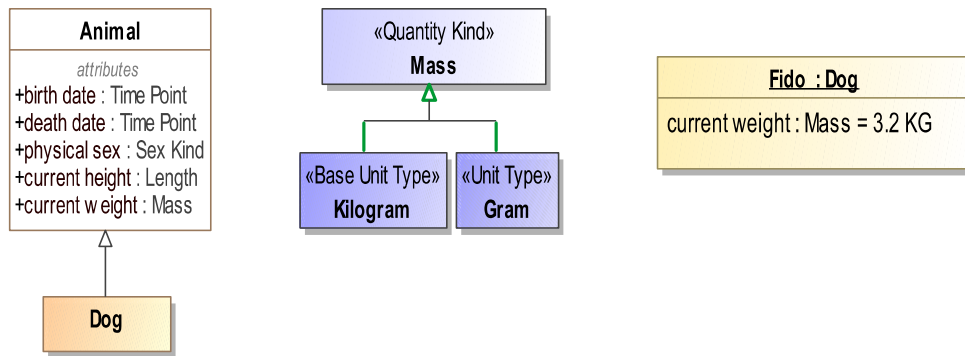


Figure 1.19: Defining and Using a Characteristic

Figure 1.19 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.

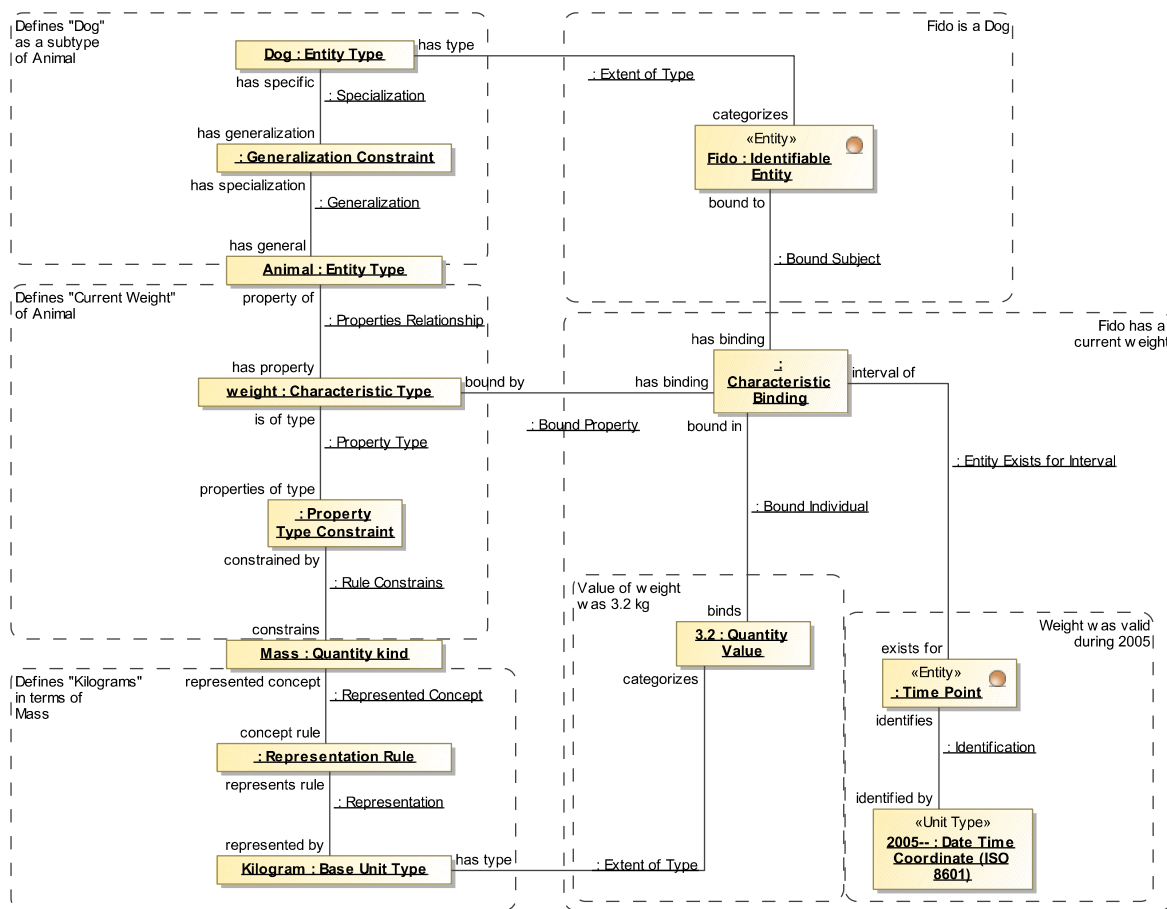


Figure 1.20: Instance Model Defining Characteristic & Value for an Entity

Figure 1.20 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 Unity 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.

