Table of Contents

Table of Figures

Table of Tables

0

1 Introduction to SMIF Semantics

The following is a high-level description 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).

Note that this section amplifies the reference documentation in section 8. Section 8 should be consulted for specific concept definitions. The model is presented using the SMIF UML profile, which is documented in section 10???

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 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

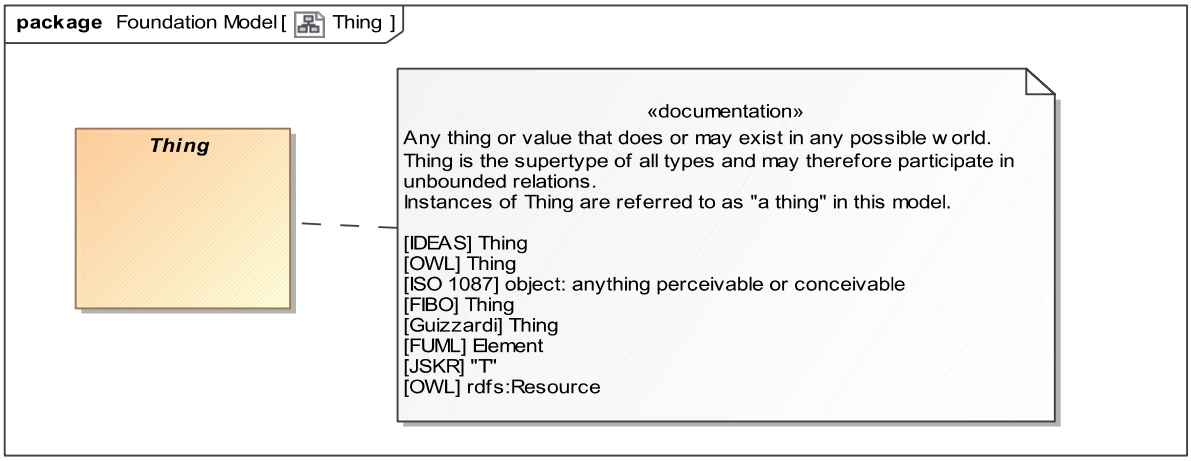


Figure 1.1: Thing

In many models it is convenient to have a sign for anything that could 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 (millennials 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

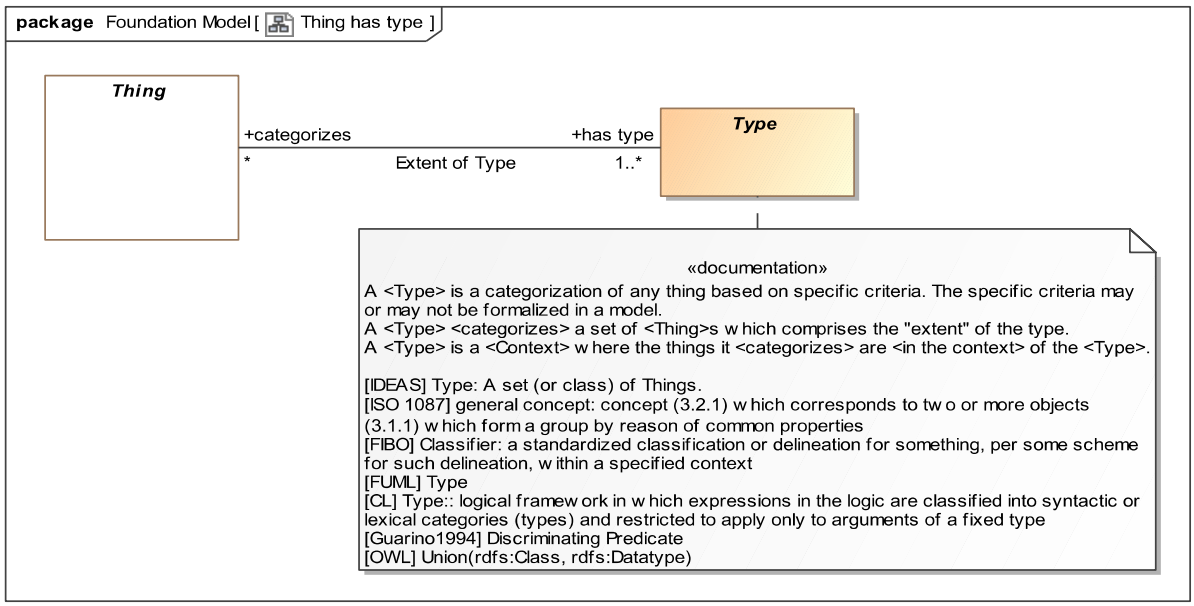


Figure 1.2: Thing has type

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

Example 1

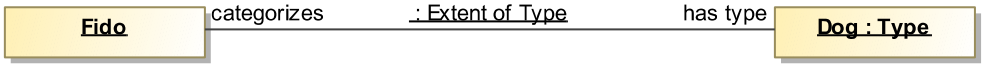


Figure 1.3: Fido is a dog example

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:

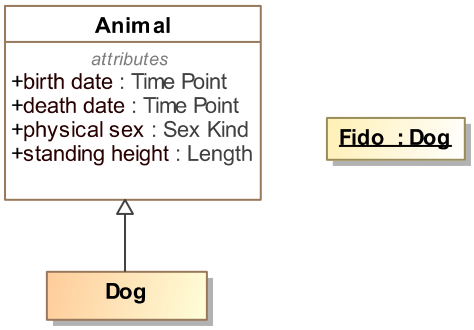


Figure 1.4: UML Type and Instance Example

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 Identifiable Entities and Values

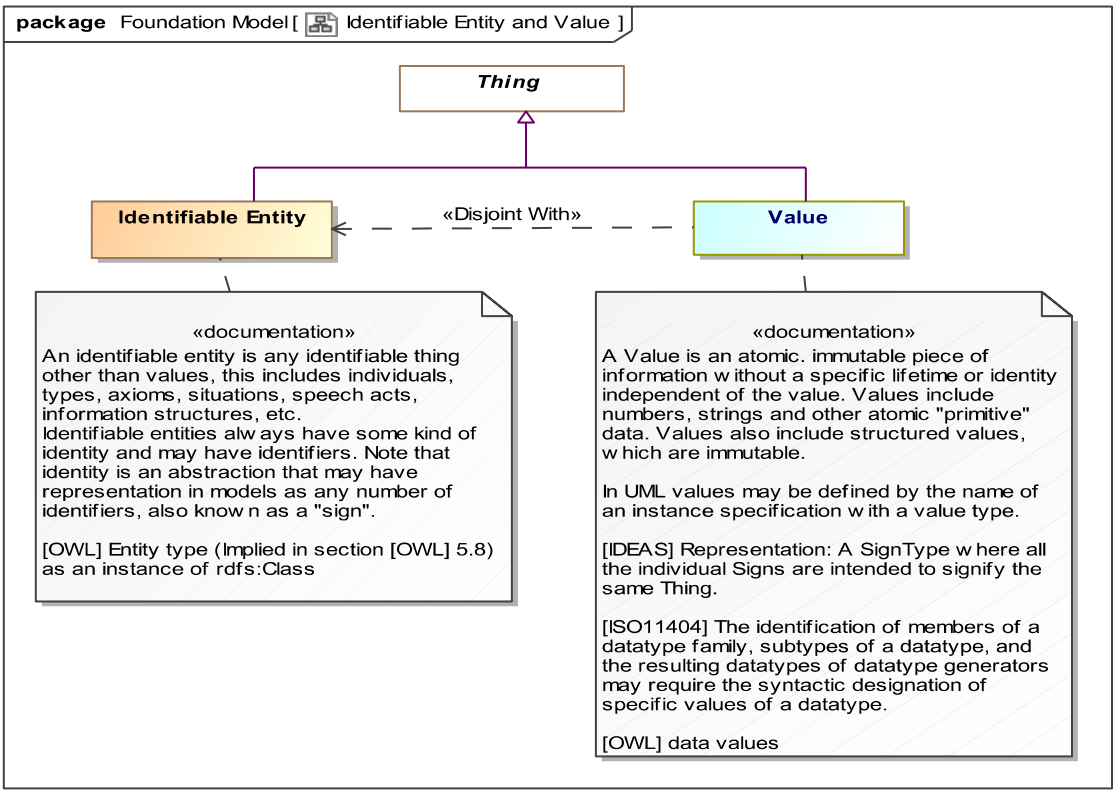


Figure 1.5: Identifiable Entities and Values

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

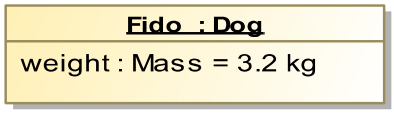


Figure 1.6: Identifiable Entity with Value Characteristic

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.

1.2 Identifiers

1.2.1 Basic Identifiers

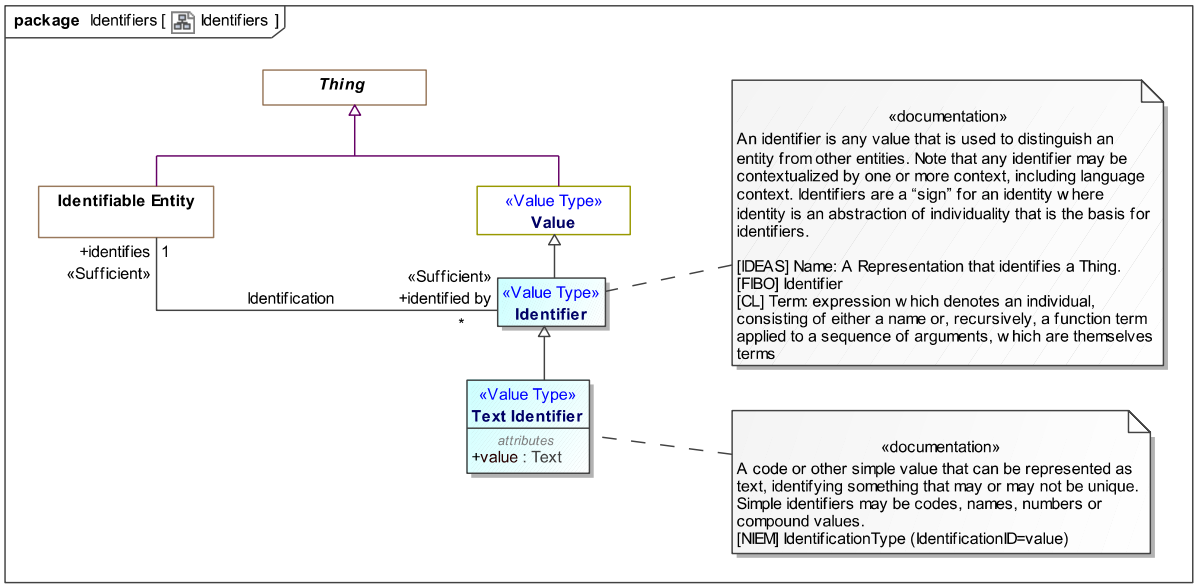


Figure 1.7: Basic Identifiers

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:

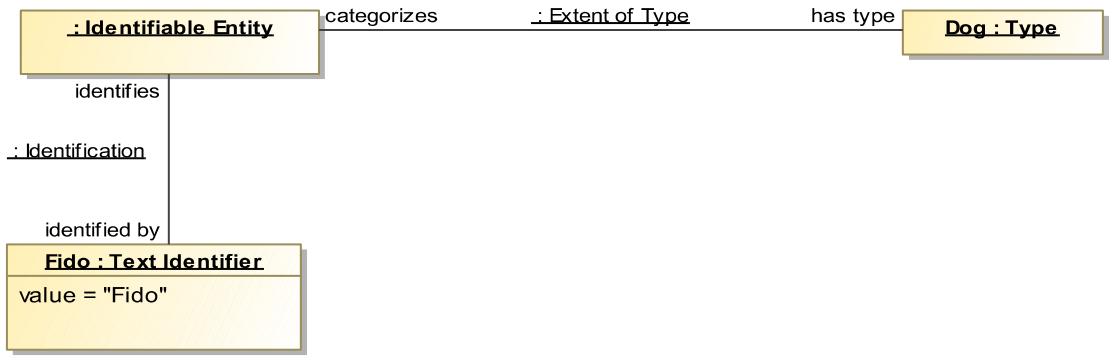


Figure 1.8: Fully expanded type and identifier instance model

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.

1.2.2 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.

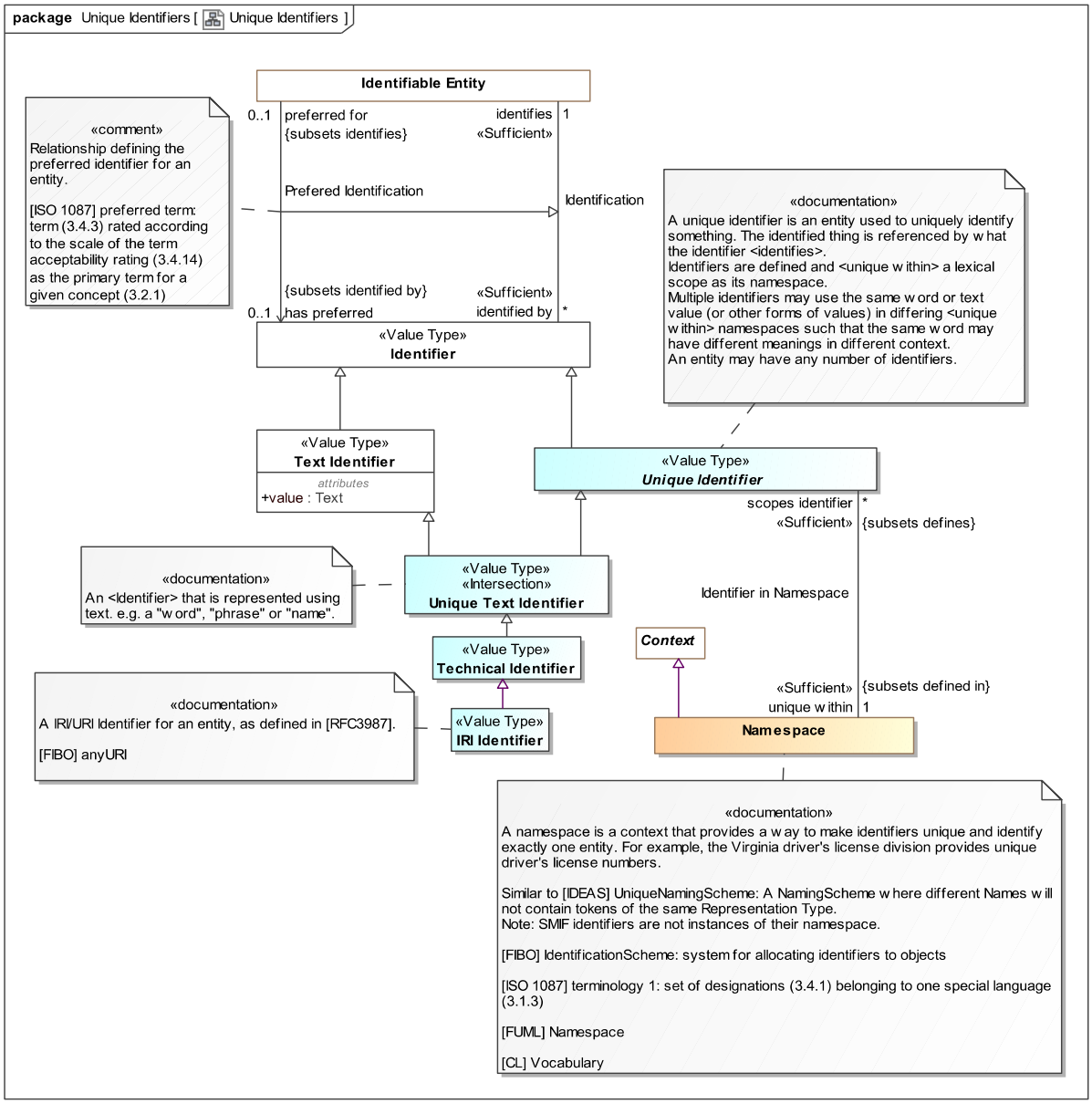


Figure 1.9: Unique Identifiers

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. Preference

is a less formal 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.

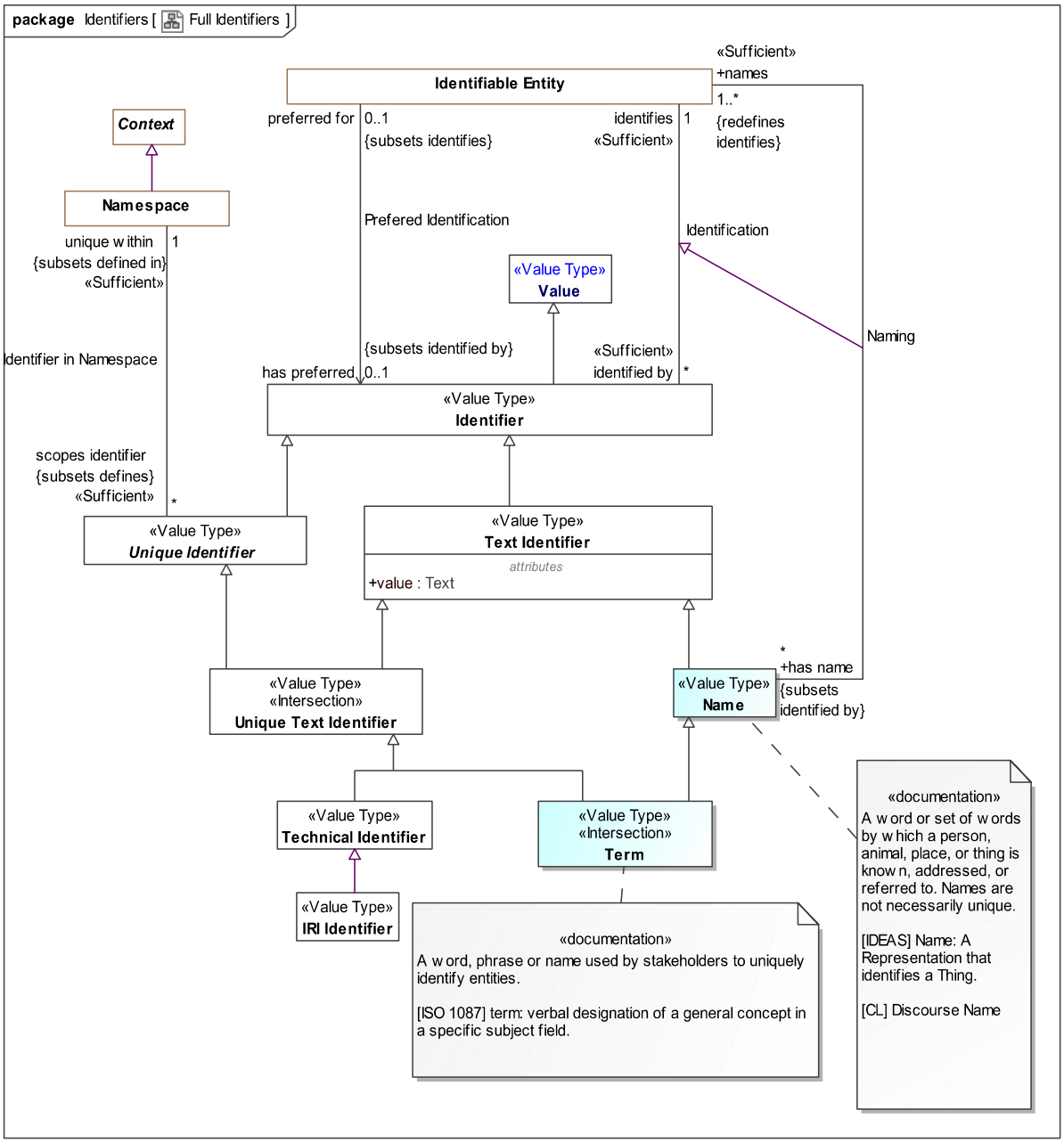


Figure 1.10: Full Identifiers Package

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.

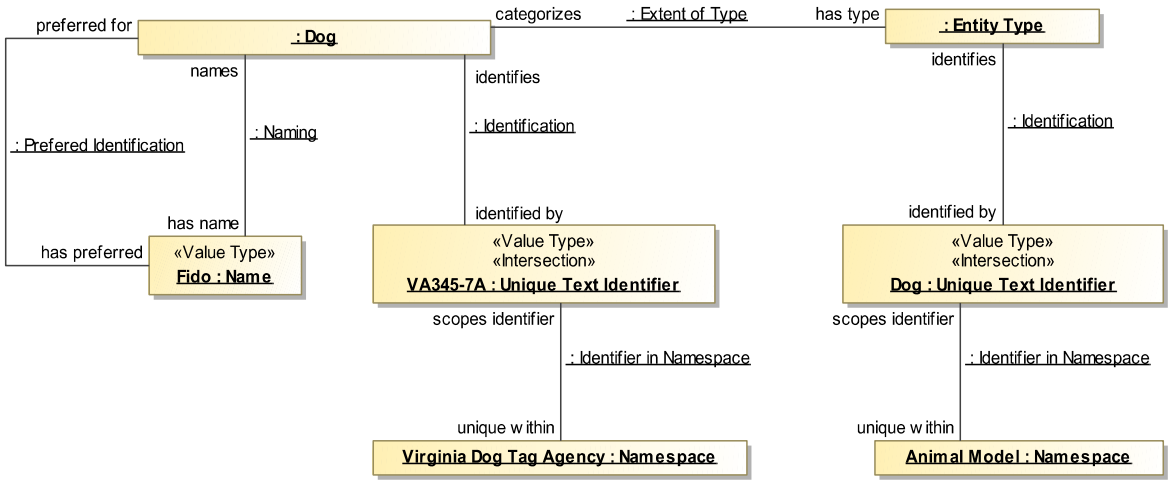


Figure 1.11: Full Identifier Example

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.

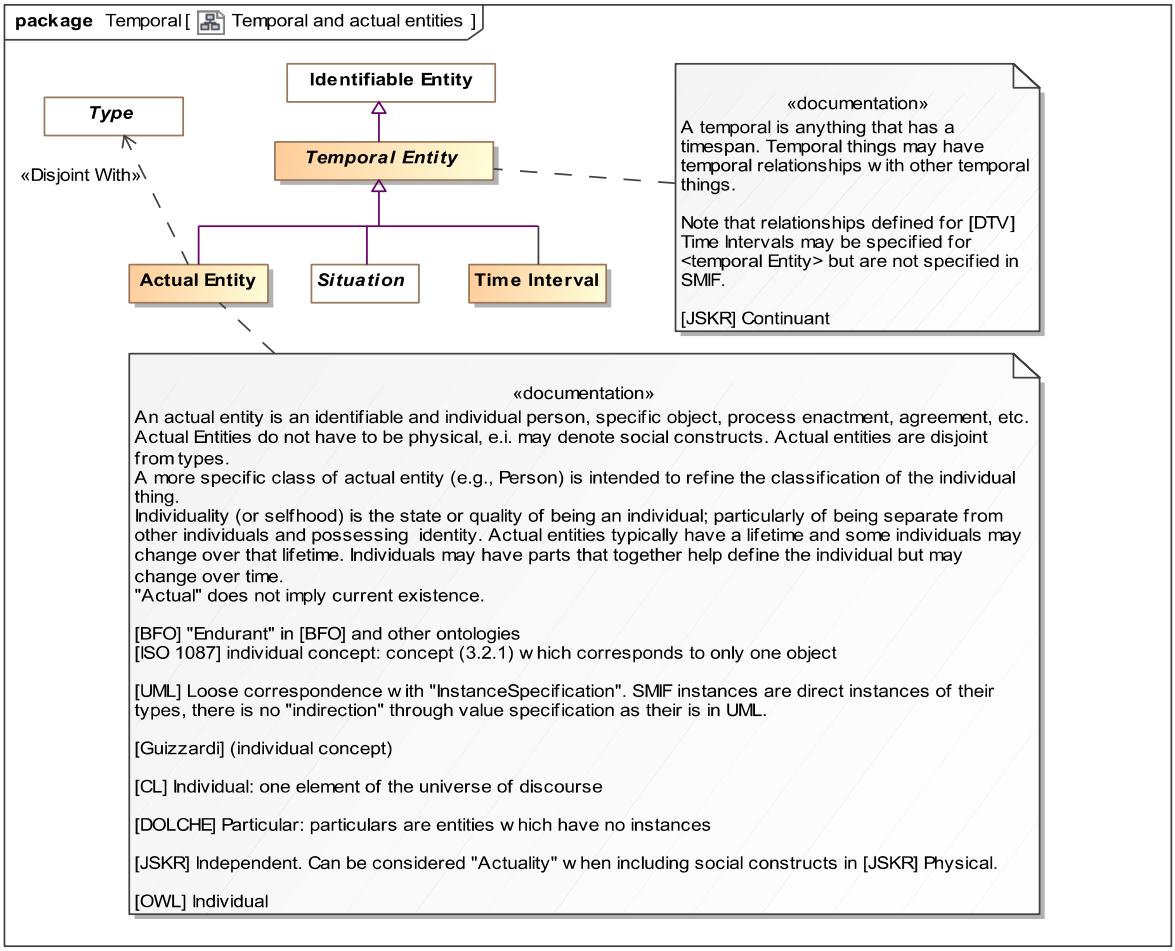


Figure 1.12: Temporal and Actual Entities

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

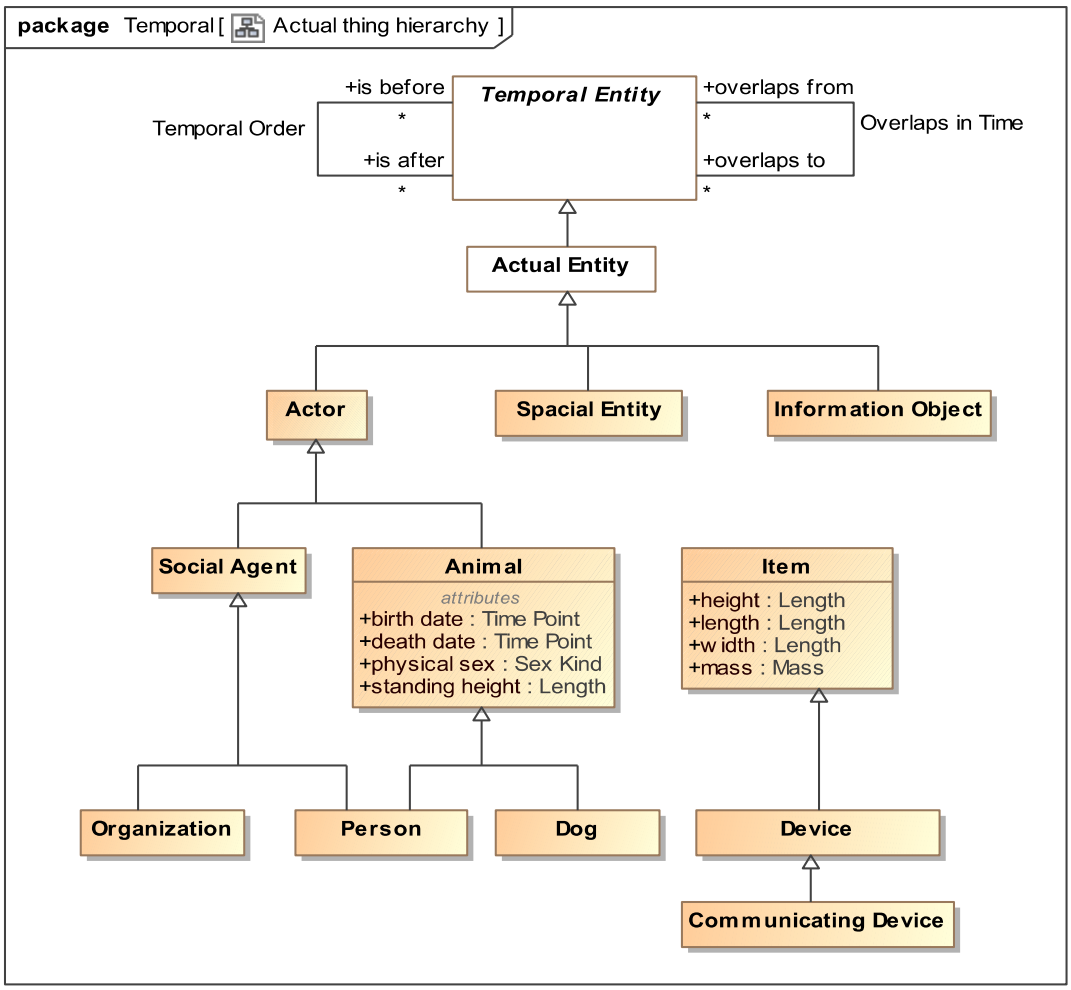


Figure ,1.13: Actual Thing Hierarchy Example

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:

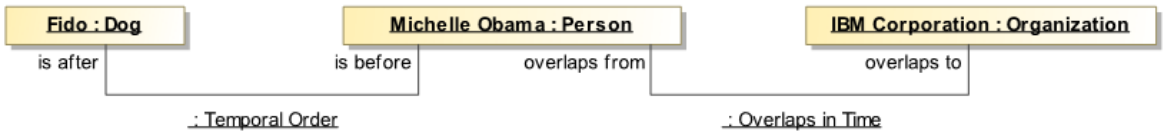


Figure 1.14: Temporal Instance Example

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.

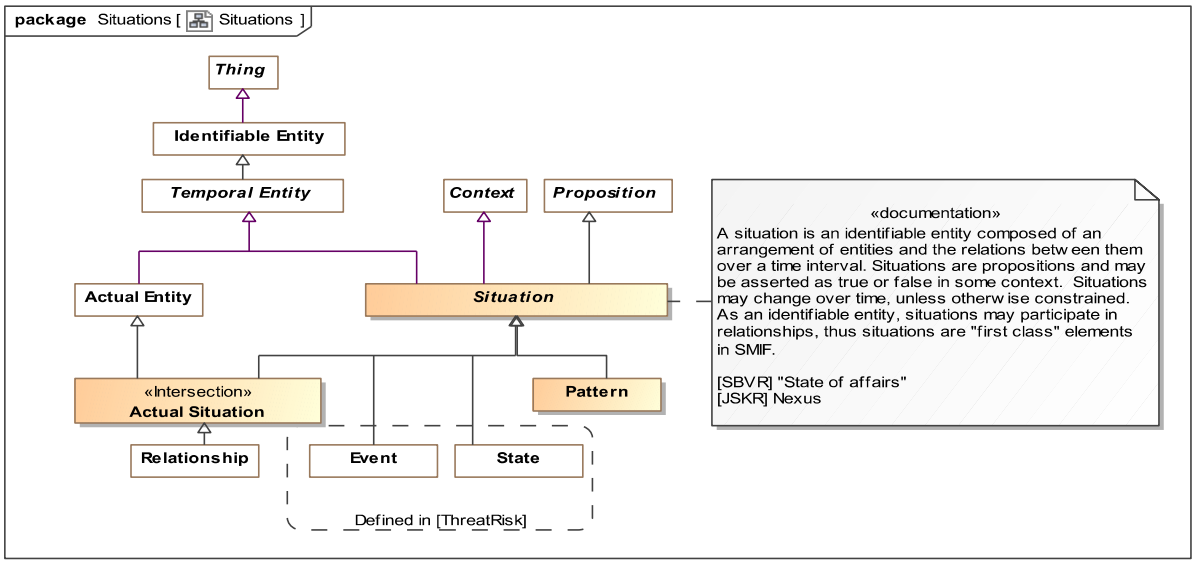


Figure 1.15: Situations - Top Level

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.

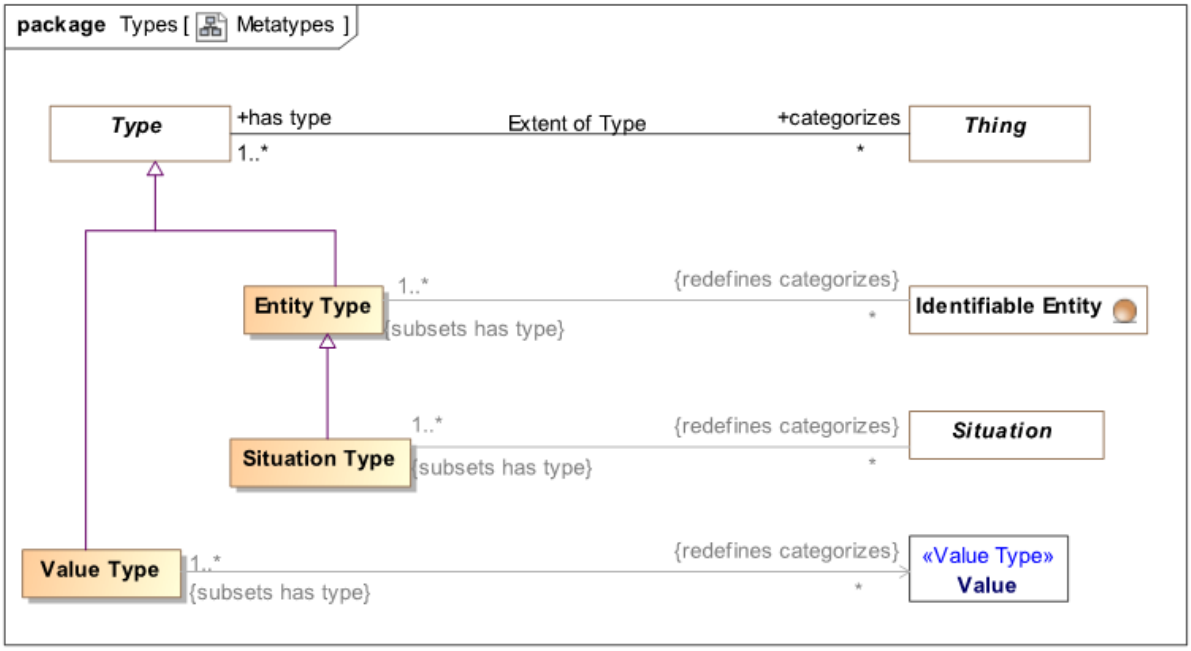


Figure 1.16: Metatypes

Figure 1.16 shows the metatypes (sometimes called “power types”) defined in SMIF for the foundation types of 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 Full Meta-Type Hierarchy

The following shows the complete hierarchy of metatypes.

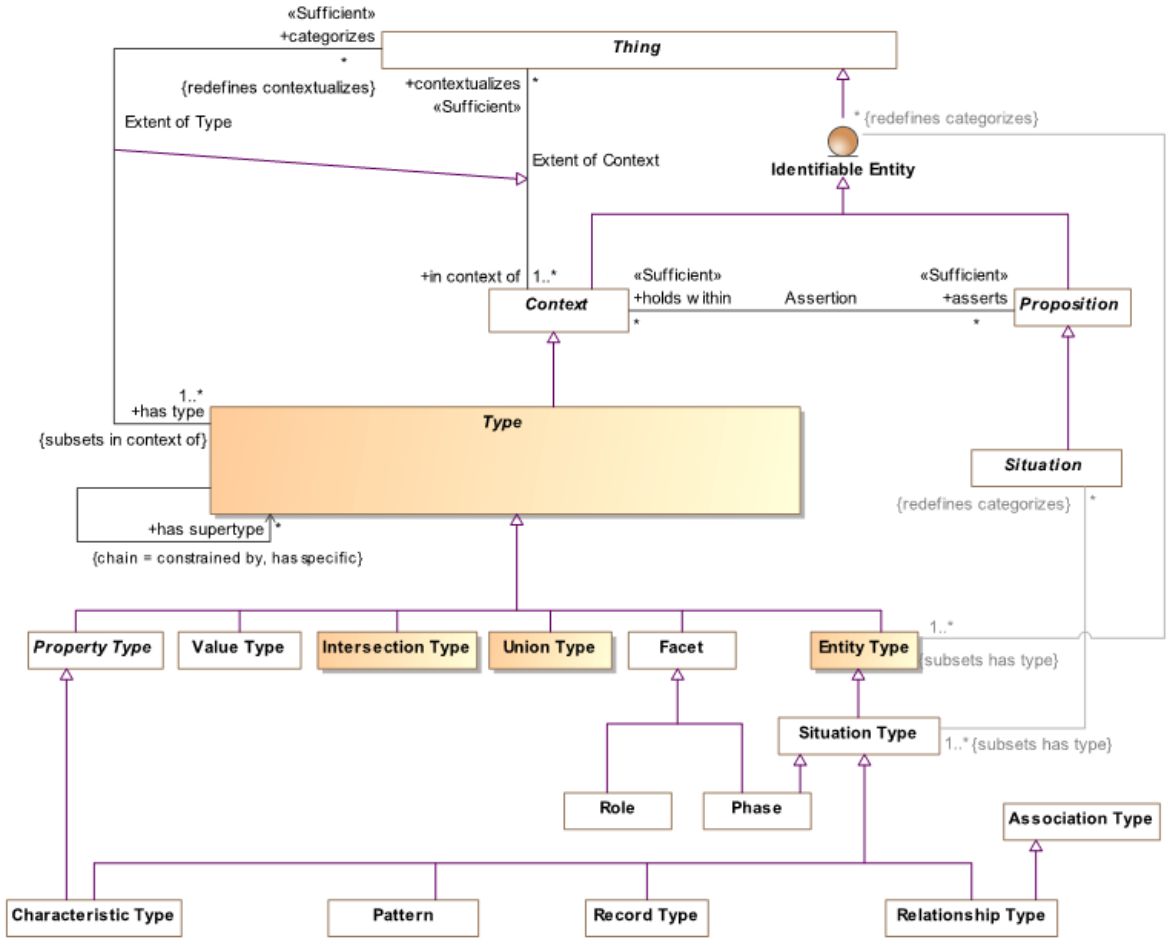


Figure 1.17: Metatypes

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

1.5.3 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

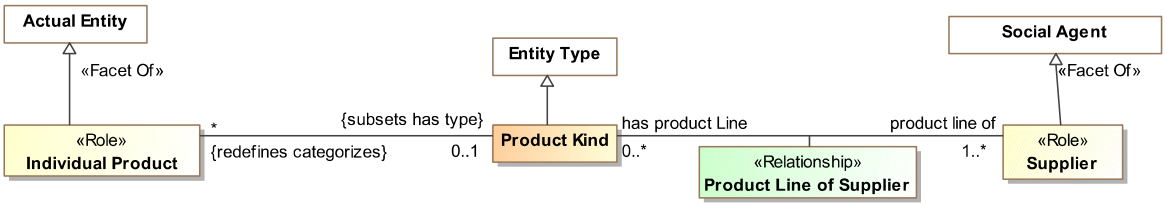


Figure 1.18: Domain Specific Metatype Example

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

Figure 1.18 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.

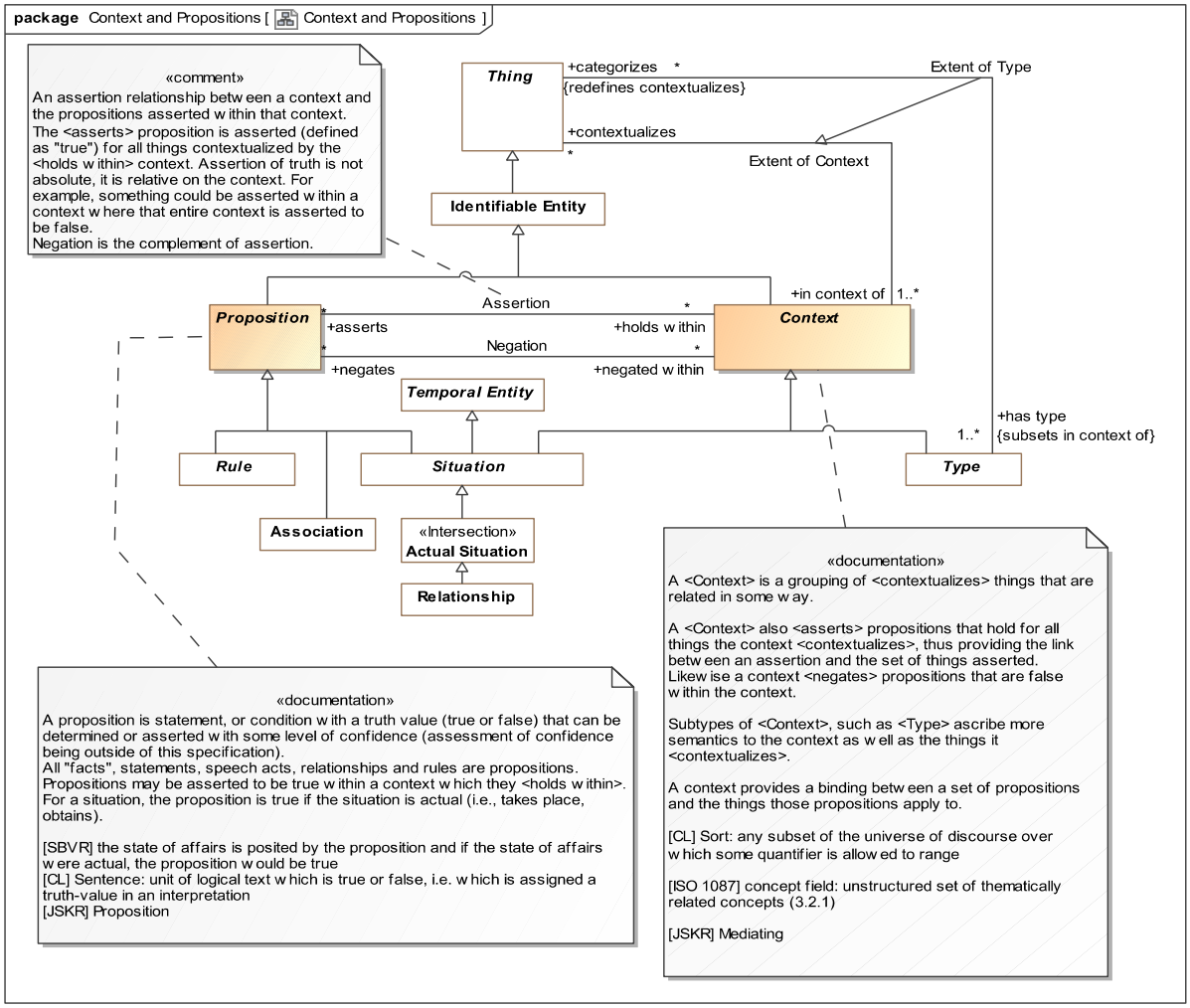


Figure 1.19: Context and Propositions

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>.

“Negation” is a relationship that is the inverse of “Asserts”, it asserts that something must NOT be true.

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.

Please see section 1.8 for examples of assertions negations in a context.

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.

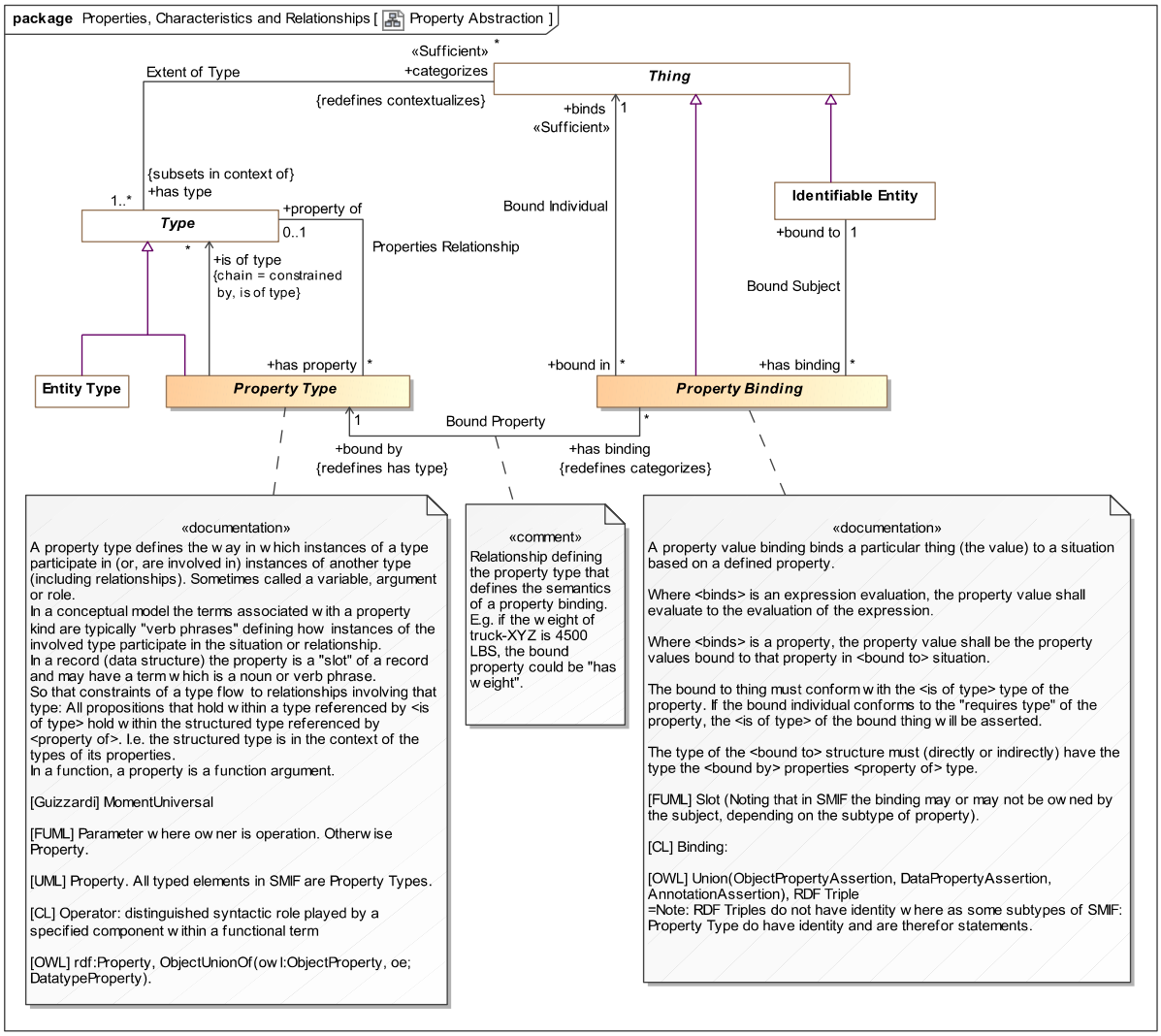


Figure 1.20: Definition of Property Type & Property Binding

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 some quality inherent in something, they 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 inheres in the entity type, there is no intervening relationship or structure. Where these is “something in the middle” between two things an “Association” or “Relationship” should be used. Relationships are described in section 1.7.3. Those 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, “Association” is the right construct to use in SMIF. Where there is a relating class, Relationship is the correct construct.

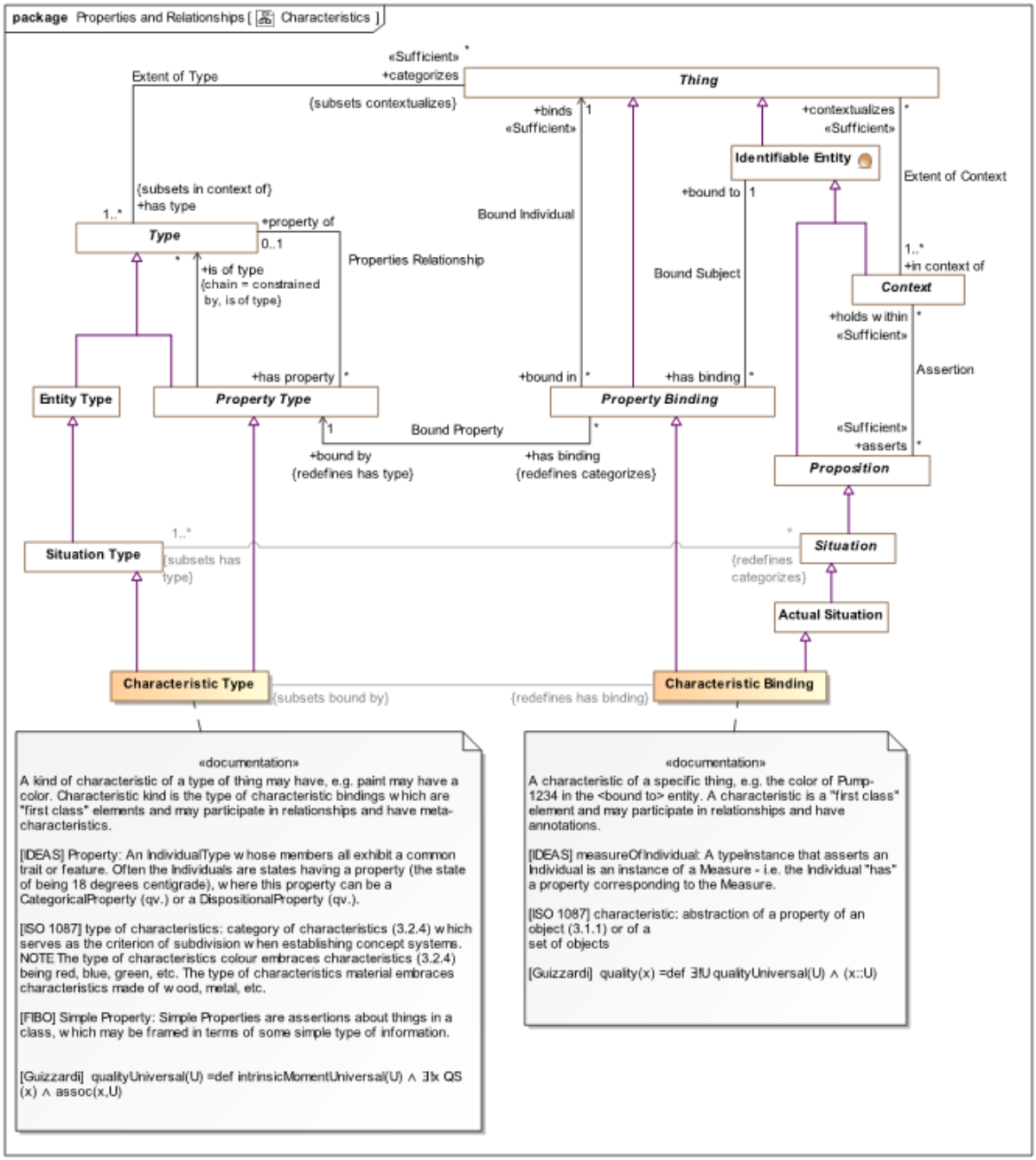


Figure 1.21: Definition of Characteristic & Characteristic Kind

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 s*ame 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

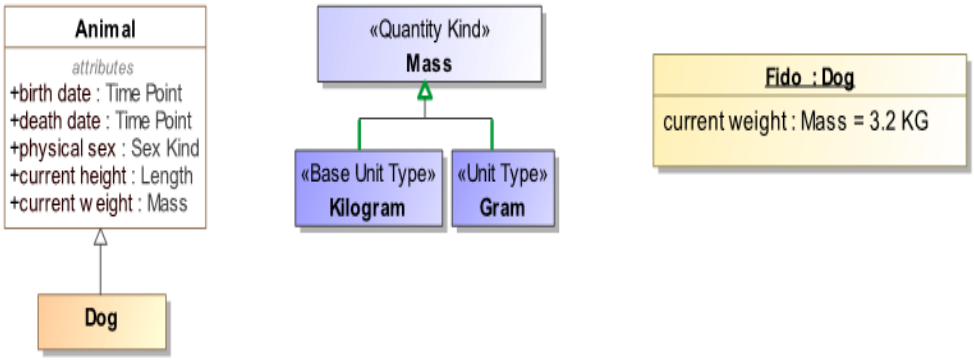


Figure 1.22: Defining and Using a Characteristic

Figure 1.22 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.

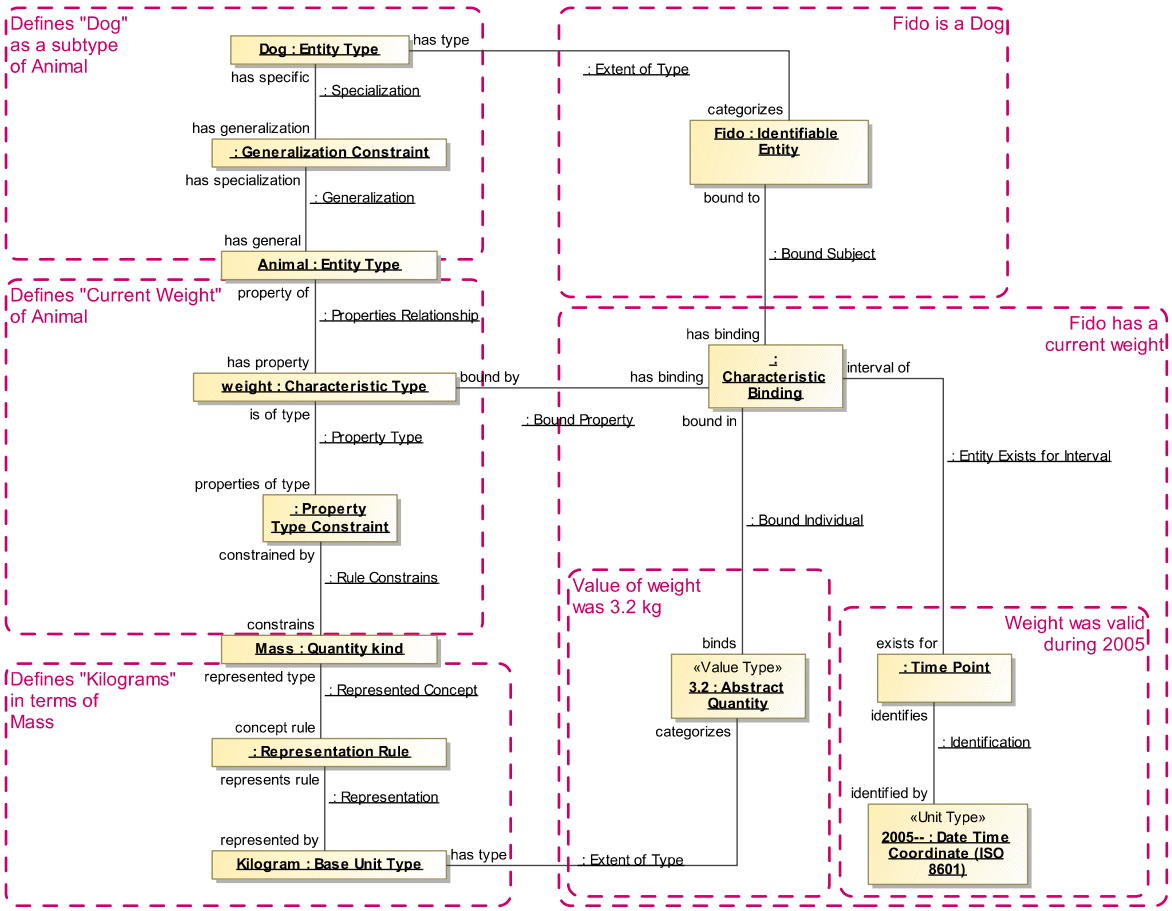


Figure 1.23: Instance Model Defining Characteristic & Value for an Entity

Figure 1.23 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.

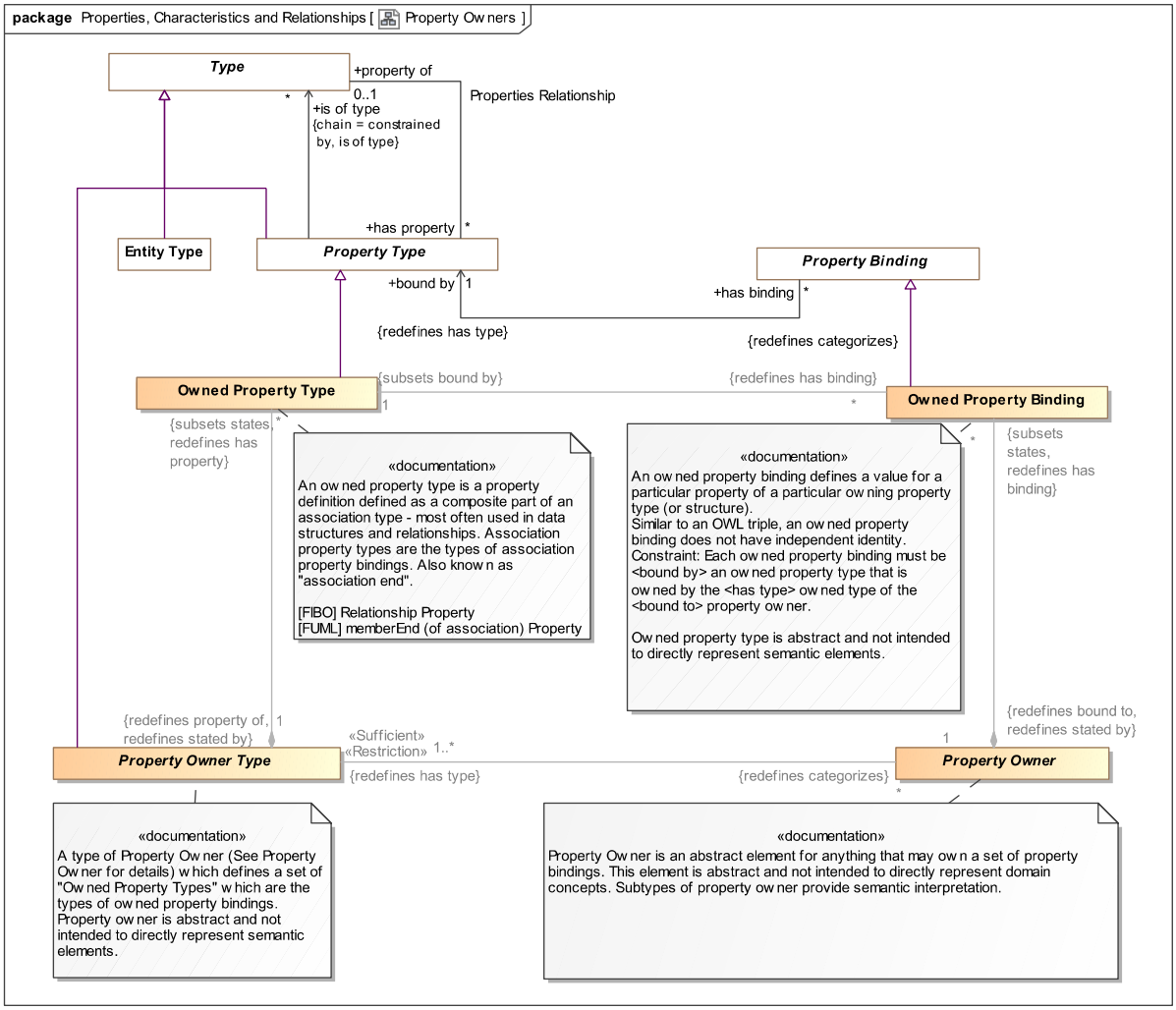
This model uses some rules not yet defined: Generalization Constraint, Property Type Constraint and Representation Rule – you may refer to the reference section for details on these rules. Time point and Date Time Coordinate come from the [ThreatRisk] example model. Rules are used rather than simple 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 Property Owner Abstraction

Many of the SMIF concept type defines and “own” sets of property types where the instances of these types “own” a set of property bindings. Such “Property Owners” are composite semantic units, where all the bindings are considered together. Examples of such property owners are associations, relationships and records. Property Owners are defined to aid in the definition of these composite semantic units. Property owner is abstract as the true semantic of each kind of property owner is defined in the appropriate subtype.



Property owners own owned property bindings, so a property owner is just a set of such bindings without further semantic interpretation. There is a corresponding type for each as Property Owner Type and Owned Property Type.

Property owners are used in associations and relationships as is seen below.

1.7.4 Associations and Relationships

We will introduce associations and relationships together, as they are are similar in that they “relate” things. The difference is one of independence and lifetime. Relationships are “first class situations”, they have their own timeframe and identity. For example, a marriage can be such a relationship. Associations are similar to

relationships but their lifetime is co-existent with the lifetimes of the related entities. In many cases they are definitional for one or both ends of the association. This is also known as an “Intrinsic Relation” [Guizzardi].

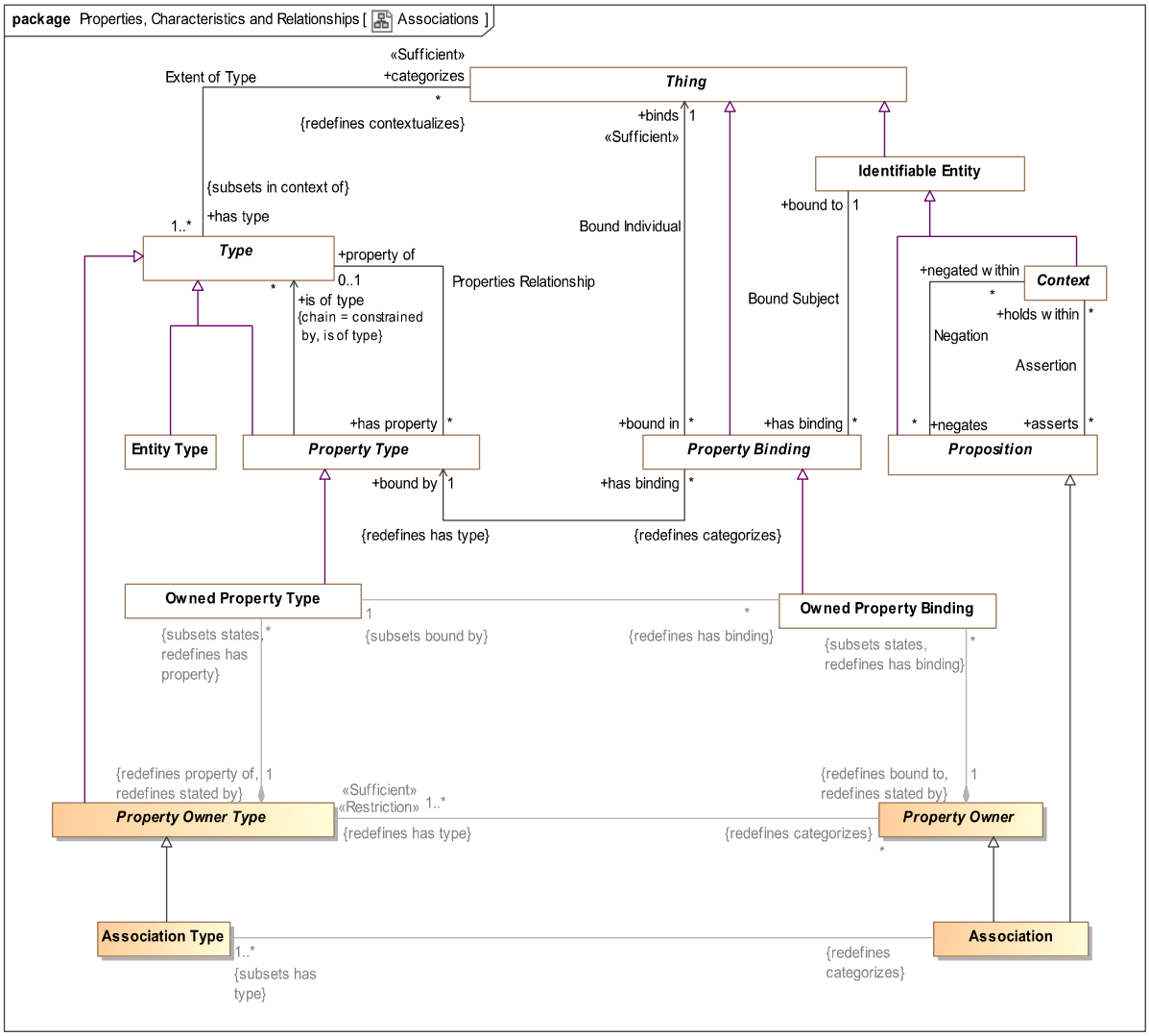


Figure 1.24: Associations

Associations are “Propositions” in that they can be true or false and asserted or negated within a context. Each association has a set of “Owned Property Bindings” that define the related things. Associations are defined with association types that define a set of owned property types, the ends of the association.

Example

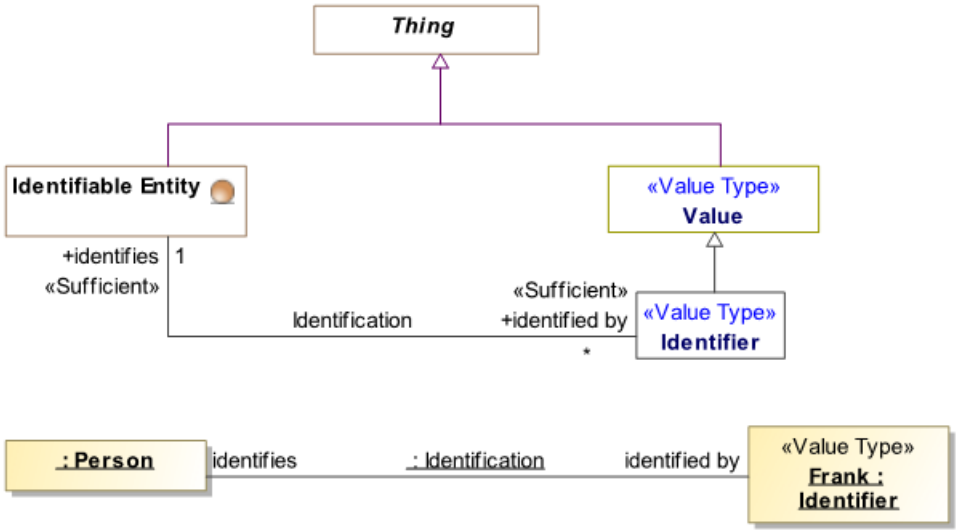


Figure 1.25: Association Example

We have already seem multiple associations, in figure 1.25 we repeat the definition of the “Identification” association and an instance of it showing “Frank” <identifies> a person. Identification is inherent in an identifier, it can’t exist without it and what an identifier identifies does not change (else it would be another identifier). This Identification association is “Existentially dependent” on both entities and it also serves to define those entities. The above is shown in terms of the SMIF UML profile, as instances of the SMIF conceptual model it would be:

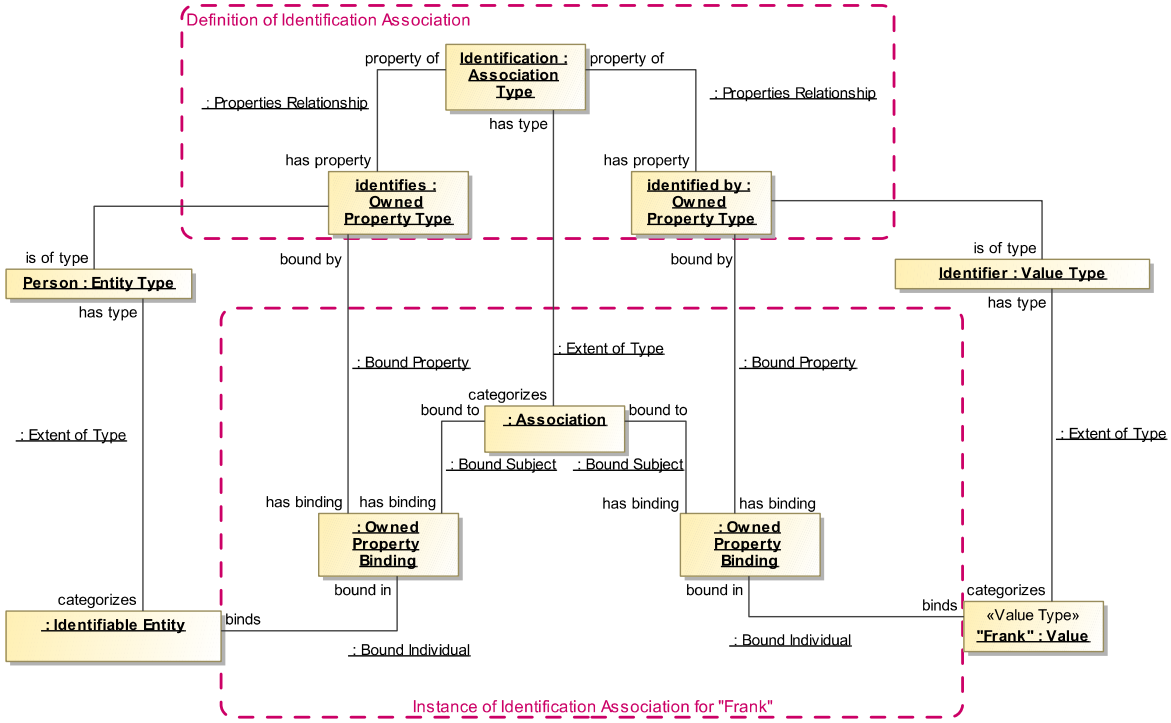


Figure 1.26: SMIF model instances for an association

In 1.26 we see the SMIF model instances corresponding to the UML profile view in figure 1.25. The “Identification” “Association Type” has two association property types: “identifies” and “identified by” that each have a corresponding type: “Entity Type” and “Identifier”.

This association type is the type of an association (that has no name) with two bindings to an instance of person and an instance of Identifier, “Frank”. This association hold for the lifetime of both entities.

1.7.5 Relationships

Relationships, and the corresponding Relationship Type, are part of the foundation of the SMIF language. In fact, 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. Relationships augment situations in that they are first-class actual situations. 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, also known as the “Qua Entity” [Guizzardi]. 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.

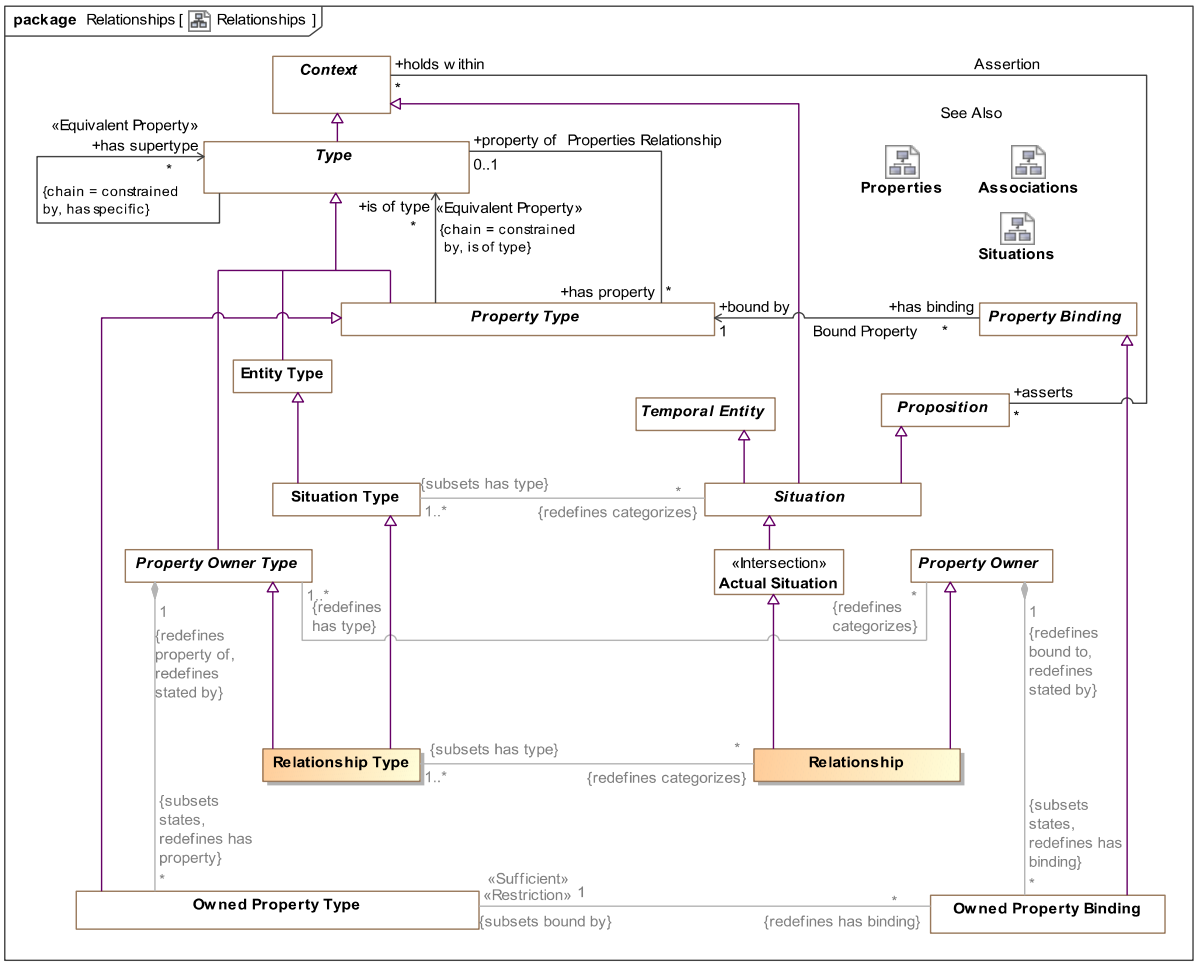


Figure 1.27: Defining Relationships

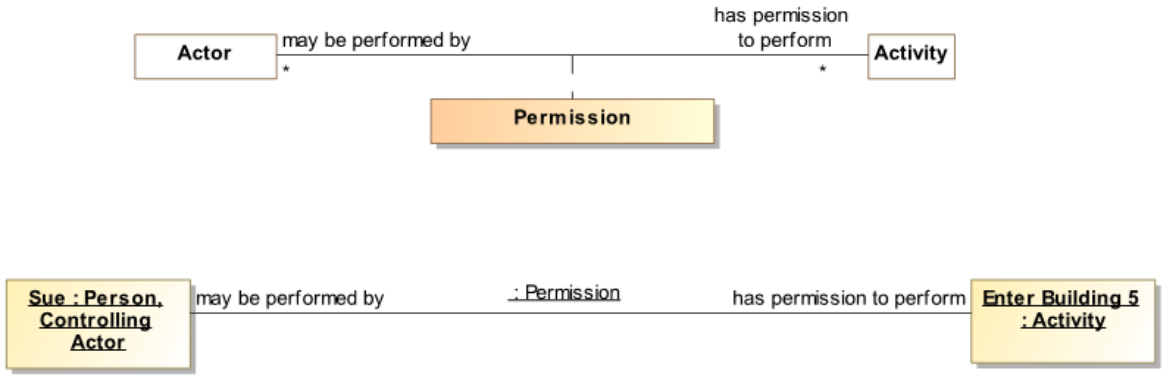
Figure 1.27 shows the SMIF conceptual model defining relationships, building on concepts we have already seen such as situations, associations and properties.

Relationship types build on “association” and “owned properties” as the ends of relationships. In the Characteristics section we saw that each characteristic is an independent situation. On the other hand, owned 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 “Owned Property Binding” is shown as being “owned” by the “association”.

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 owned property types (it is legal so share owned property types between relationship types).

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

Example 1



The “Permission” relationship example 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.

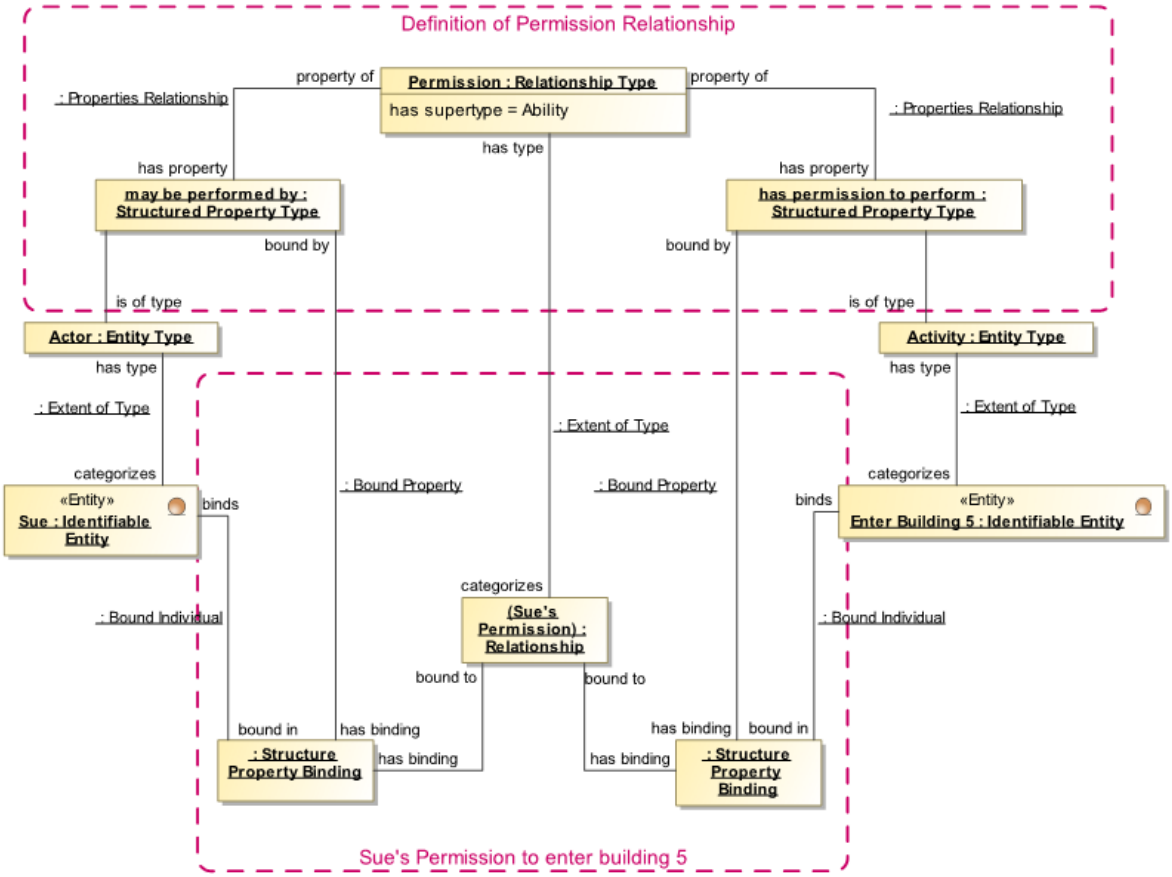


Figure 1.28: Defining and Using a Relationship

Assuming that “Actor” and “Activity” are already defined, we define a new “Relationship Type” with a name of “Permission”. Permission has two “Owned Property Types”: “may be performed by” an entity that <is of type> “Actor” and “has permission to perform” entity 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. - the actual Sue having the actual business permission to enter the actual building; we say this to emphasize that we are modeling the “real world”, not data about it. Sues’ permission relationship has two “Owned 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 can attest to a permission or other kind of ability.

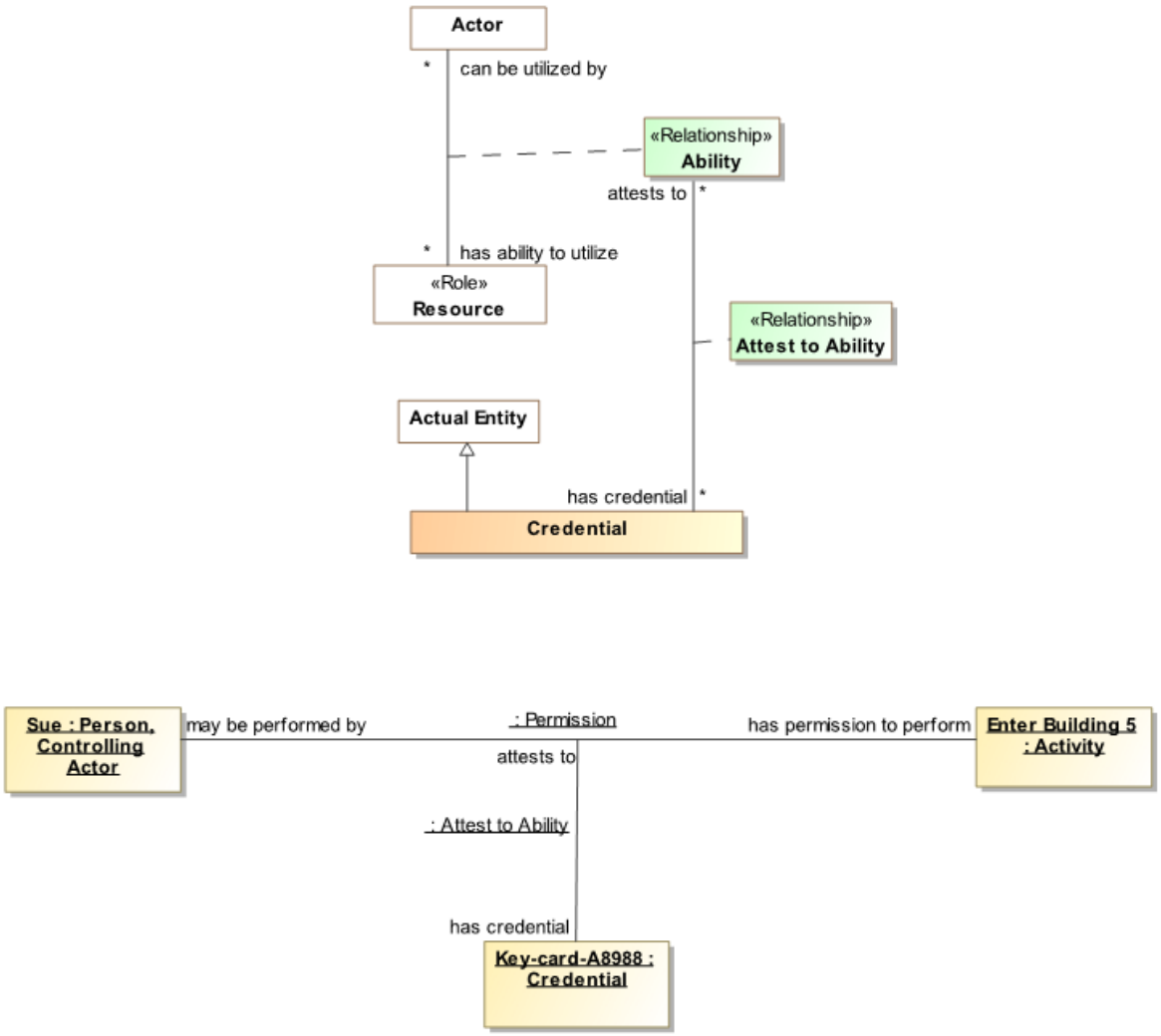


Figure 1.29: Relationship Involving Relationships

Permission is a subtype of "Ability" (we know it it not in the diagram, trust us). An ability is a relationship between an actor and some resource they can use – in the case above the ability was "Permission" to do something 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 – the "Permission" relationship is one end of the ""Attests to Ability" relationship.

At the bottom of figure 1.29 we see an instance of the relationship that is at the top of figure 1.29, 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, a 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 SMIF model instances instead of the UML profile.

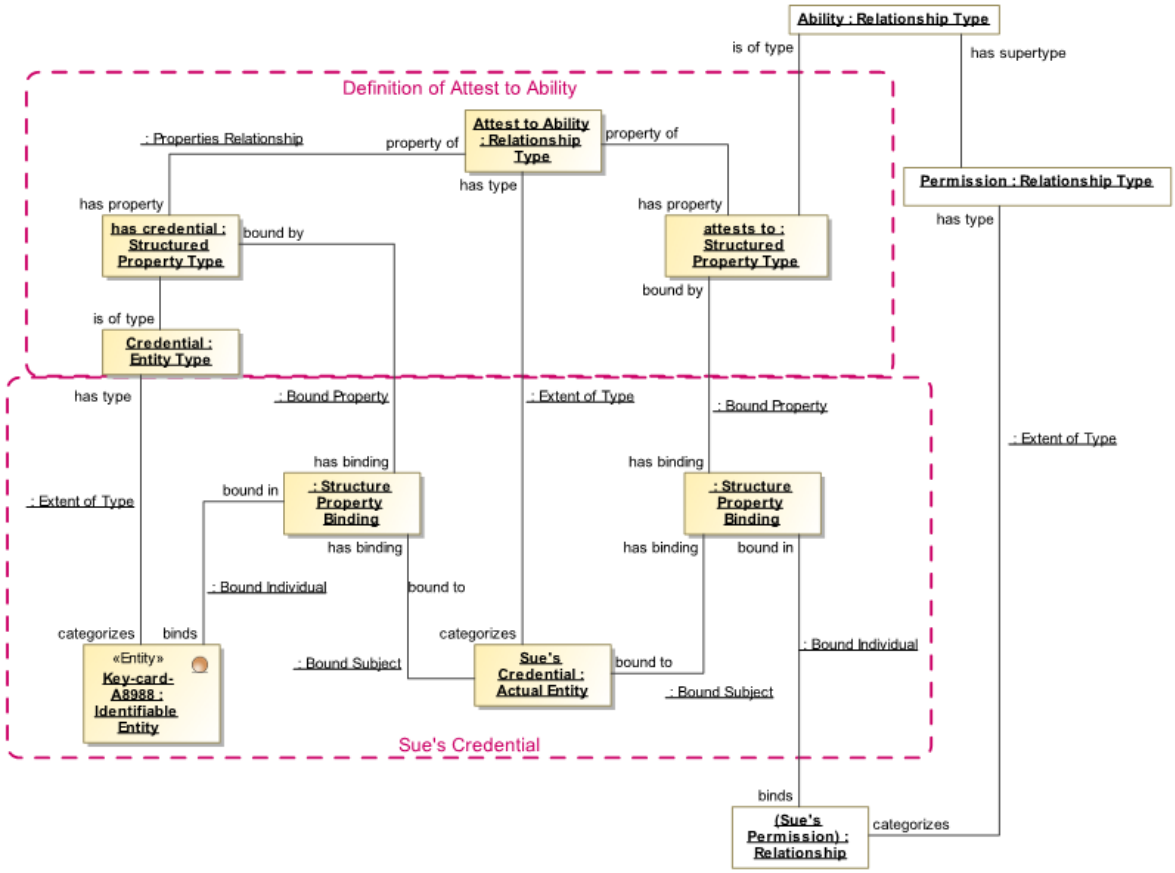


Figure 1.30: Relationship Involving Relationships – instance model

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 represent and evaluate various threat scenarios relating to such a stolen credential.
- We could model mediating actions and their result.

1.8 Composition and Sequencing of Actual Situations

In section 1.4 we discussed situations from the "outside", treating a situation like any other identifiable entity. Situations are a composition of other entities and relationships – the elements that make up a situation are "asserted" by the situation.

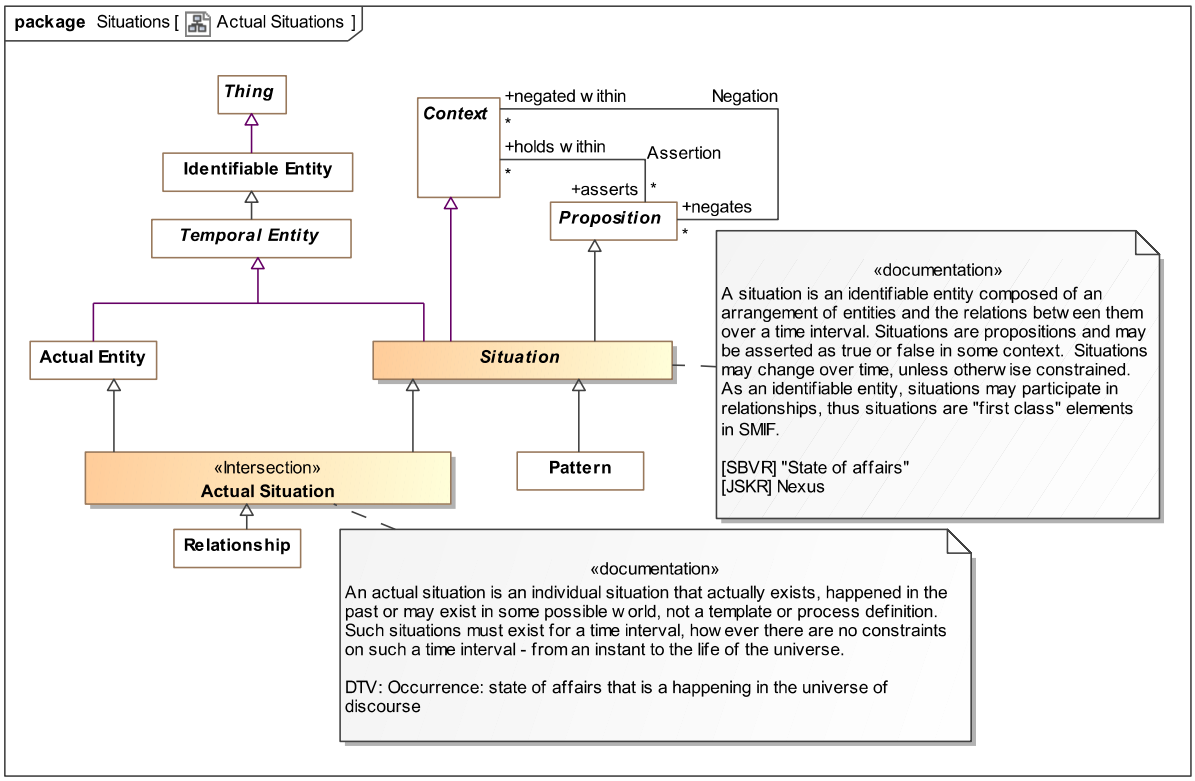


Figure 1.31: Actual Situations

Subtypes of "Situation" are "Actual Situation" or a "Pattern". This section deals with actual situation composition, we will look at patterns in a later section. Actual situations are complete, where as patterns may have variables.

As shown in figure 1.31 we see that a situation and an actual situation is a *Context* that <asserts> or <negates> Propositions. Propositions can be rules, relationships, characteristics, patterns or other situations. Each of these asserted/negated propositions becomes an element of (something true/false within) the subject situation context.

We have already seen some examples of atomic situations; relationships and characteristics. Each relationship and characteristic, such as those seen in section 1.7, is an atomic situation. A relationship is a configuration of the set of things in bound together immutably for a time period. If we had a set of such relationships, all true "together" it would make up an actual situation. Since situations are temporal entities, they exist for a certain time interval *but the elements within them may change*.

Building on the example of Sue and her "key card", we could have the situation that Sue has a permission, the key card attests to that permission and that she is in possession of it. We are using some additional types and relationships defined in [ThreatRisk] and are assuming these are sufficiently intuitive to be shown without definition. The [ThreatRisk] specification is available for review.

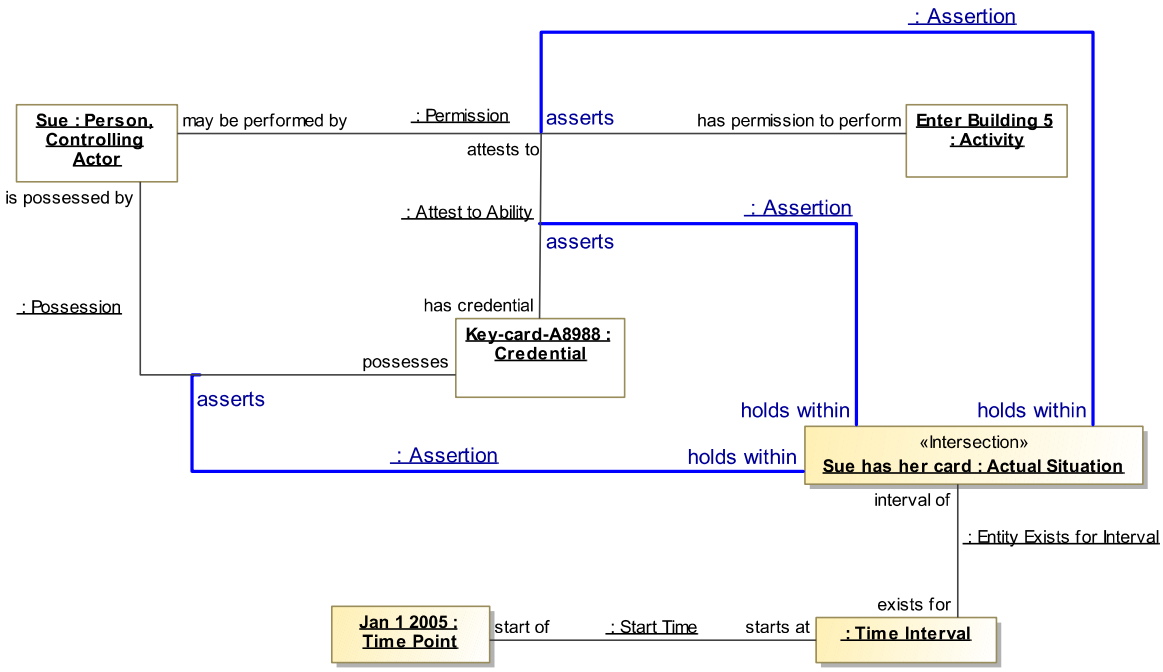


Figure 1.32: Initial Situation Example

Figure 1.32 shows the addition of the "Possession" relationship – Sue <possesses> Key-card-A8988. We also introduce the "Sue has her card" actual situation which started "Jan 1 2005". This situation <asserts> the possession, the permission and the attest to ability relationships. These relationships were all "true" starting on this date, based on this context of the "sue has her card" situation. Note that it doesn't say anything else about these relationships, this does not imply that any of these situations did or did not exist at any other time – that could be said, but it is not here. *Lack of an assertion does not imply the opposite* – this is the "open world assumption" in action.

But what if Sue's card got stolen?

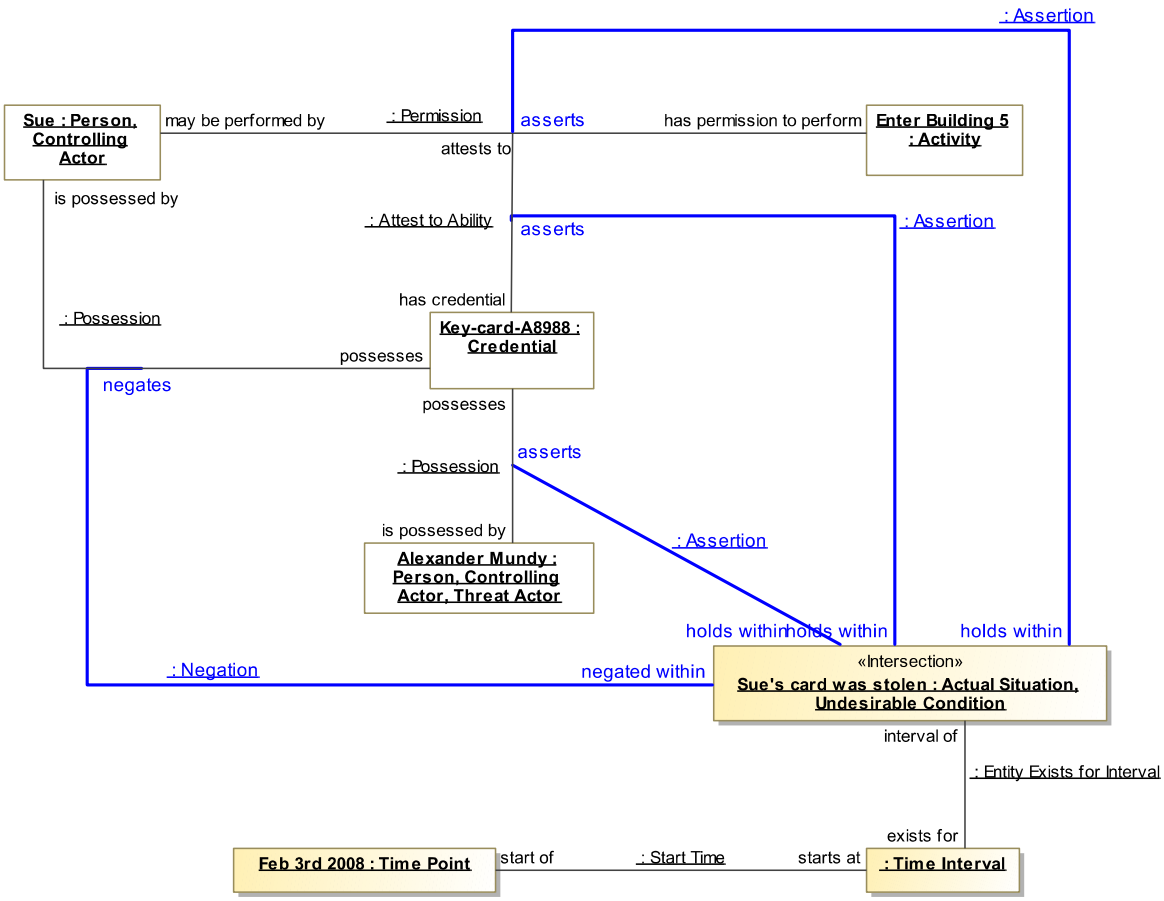


Figure 1.33: Example Situation After Theft

In an attack (not shown), Sue's card was stolen by "Alexander Mundy[1]", so now we have a new situation - "Sue's card was stolen". There is a new "Possession" relationship – Alexander Mundy <possesses> "Key-card-A8988". In this new situation starting on Feb 3rd 2008, it is asserted that Alexander possesses the card and that the card still attests to the permission of Sue to enter building 5. This is also classified as an "Undesirable Situation" (A classification from [ThreatRisk]). Note that in this situation Sue's possession of "Key-card-A8988" is "negated"; that it is stated to not being true. We are saying Alexander has it and Sue doesn't.

Assuming Sue reported the theft there should be some mediation action taken!

[1] "It takes a thief" Television Series

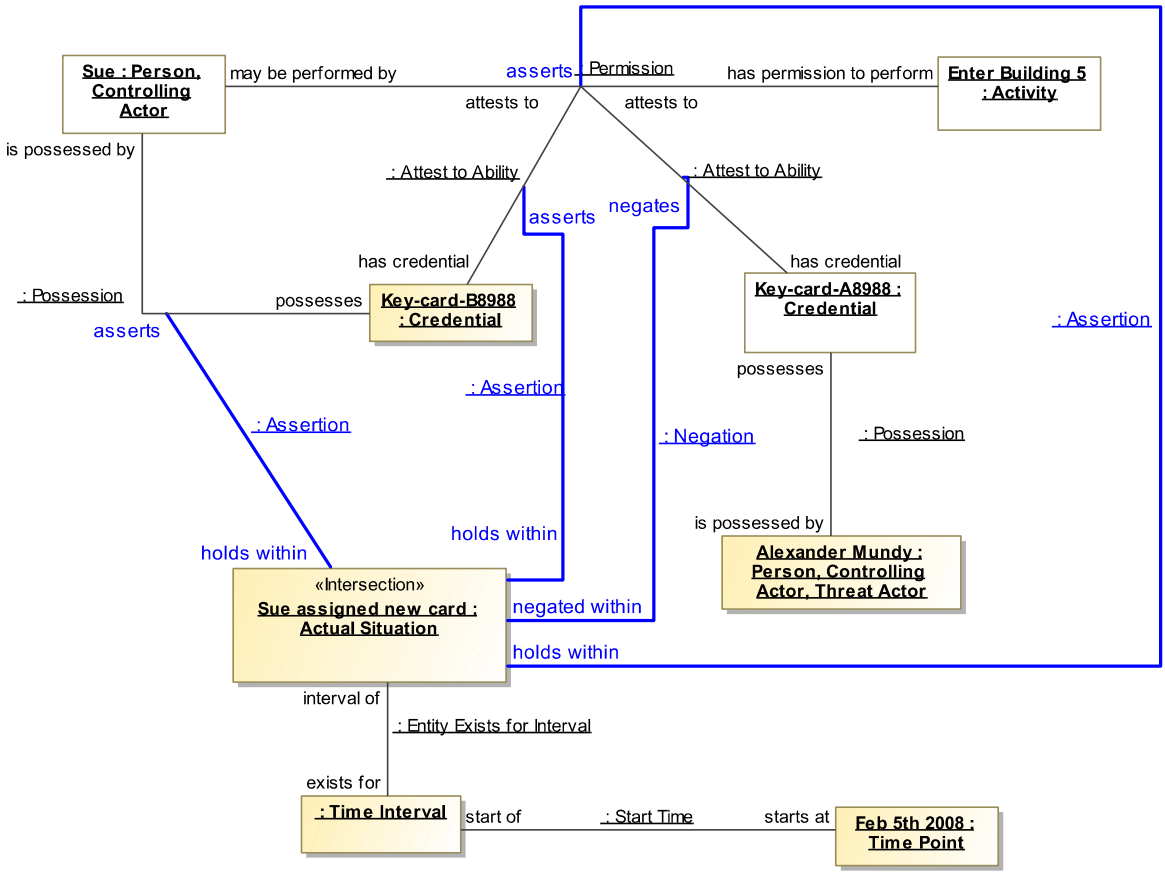


Figure 1.34: Example Situation After Mediation

The situation after a mediation (mediation activity not shown) is that card “Key-card-A8988” was revoked and a new card, “Key-card-B8988” was assigned to Sue. The post-mediation situation called “Sue assigned new card” shows that Sue’s permission is still in tact (the permission at a business level never changed), that Sue has the new card. But, the “attest to ability” relationship of “Key-card-A8988” has been negated and “Key-card-B8988” asserted (how this happens it outside of this model). In the final situation we don’t know if Alexander still has the old card or not – but we don’t care. Building 5 is safe!

What this has shown is a sequence of situations, happening in time, relating things that exist across all these situations.

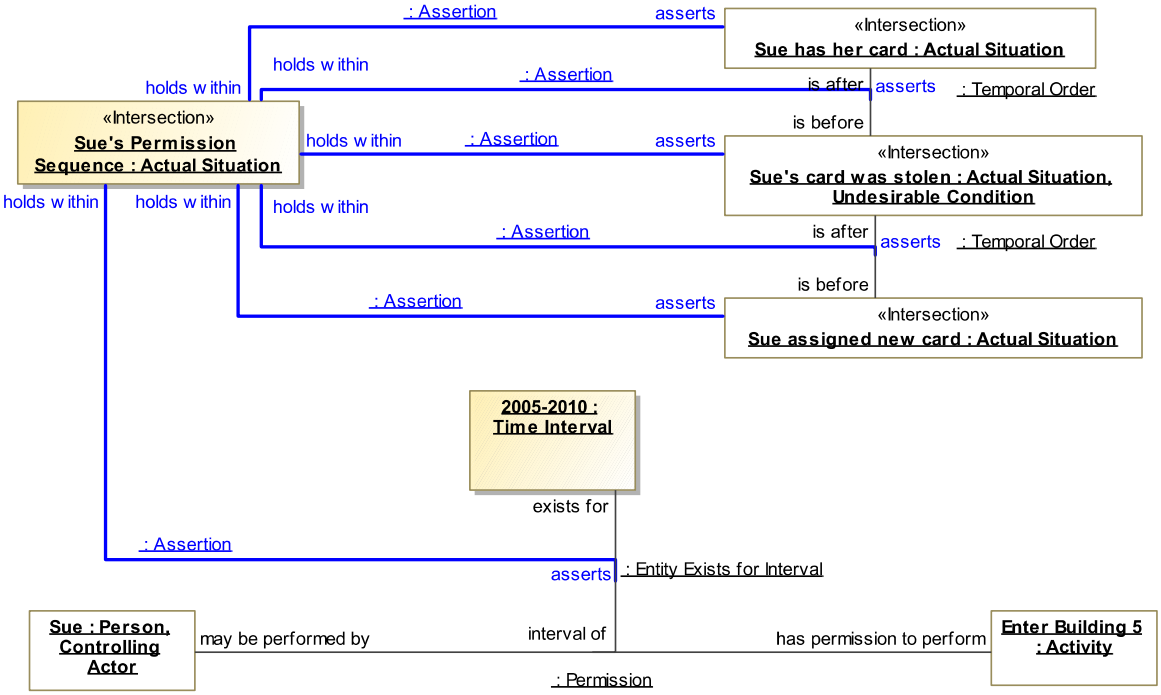


Figure 1.35: Sequence of Situations Example

In that each situation is its "own thing" happening in its own timeframe they can all "co-exist" in the same repository and be analyzed together. We can have an overall situation "Sue's Permission Sequence" that asserts them all and defines some ordering. We could also add additional relationships, perhaps to say Sue's "business permission" is valid in 2005-2010. Note that Sue, the permission, the key cards and their relationships are not "created and deleted", but retain their lifetimes through these various situations. What changes is the situations they are asserted in and the termination dates of the time intervals. This "4D" capability allows the federation of information across different timeframes such that we can analyze actual and possible cause, effect, correlation and mediation.

"Actual Situations" are the glue that binds together these timeframes and related entities that exist across time. Also note that we are showing each assertion individually, but it is possible to bunch a set of elements together under a package and assert them together.

1.9 Patterns

Patterns are similar to actual situations in that they are configurations of entities and the relationships between them, however patterns represent a *set* of real or possible actual situations. Patterns have *variables* that are are placeholders for elements in actual patterns.

For example; Sue having possession of a credential for entering building 5 is “actual”. People possessing a credential to enter some building is a pattern. The pattern <classifies> actual situations that meet the constraints of the pattern.

While patterns may contain variables, they may also include actual entities. For example, we could describe the pattern of people that have permission to enter building 5 but do not have the credential in their possession – a pattern to worry about. “building 5” is an actual entity where as the person and their credential are variables.

Patterns may be used for information mapping, to express rules, to query or to define projections of viewpoints for specific kinds of stakeholders.

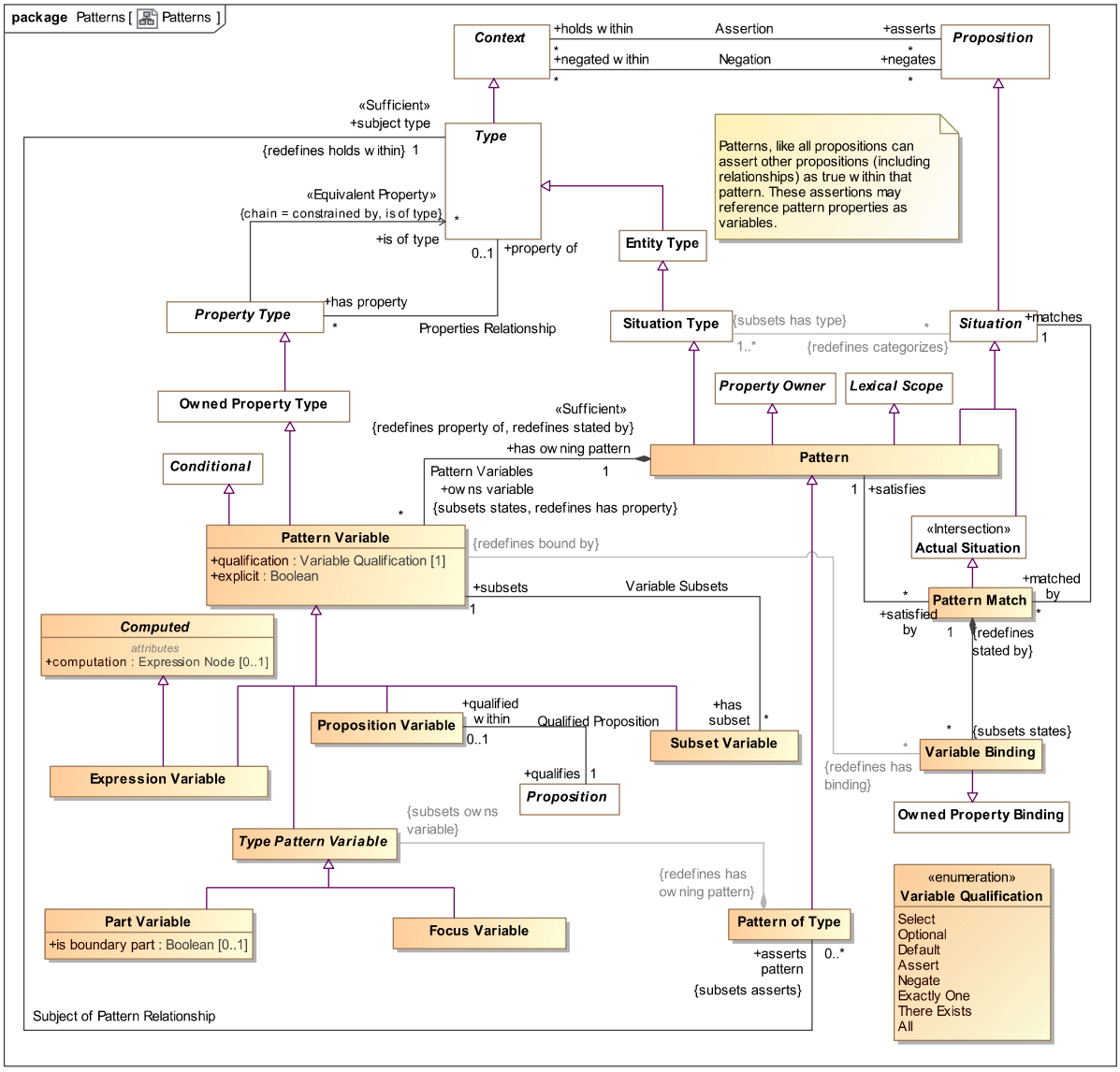


Figure 1.36: Full Pattern Model

For context, figure 1.36 presents the full pattern model without further comment. We will “build up” these concepts one step at a time, below.

1.9.1 Patterns – top level

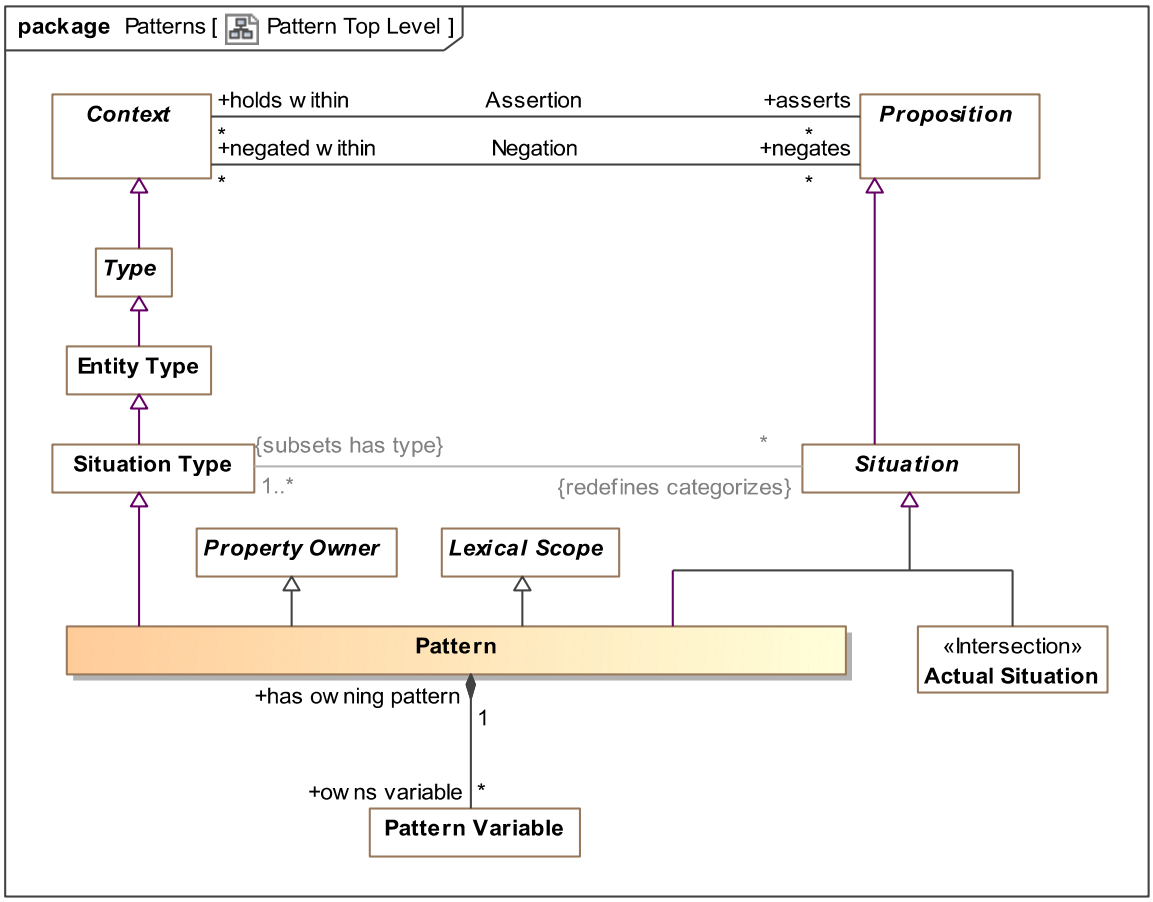


Figure 1.37: Patterns - Top Level Model

Figure 1.37 illustrates how patterns fit into both types and actual situations. The "internals" of complex patterns using "Pattern Variables" will be discussed below.

Patterns are a situations in that they describe how other entities are related and combined, just like an "actual situation". Patterns are a "Situation Type" in that they <categorize> other situations that could be other patterns or actual situations. Patterns are a "Lexical Scope" in that they "own" specific assertions and variables that describe the pattern. Patterns are property holders in that they may have pattern variables.

1.9.2 Repeated Patterns

At the simplest level, patterns can be just like actual situations except that they may happen over and over. For example, if there is the situation of "My coffee cup is on my desk", that situation may occur almost every morning, except Sundays – making it a pattern. Each "actual situation" that the pattern <categorizes> has a specific time – the cup was on my desk: Monday, the cup was on my desk, Tuesday, Etc. For such simple patterns, they look just like "actual situations" with some detail missing – in this case the timeframe. Such a "repeated pattern" is the simplest kind of pattern – it has no variables other than the time the pattern instance occurs.

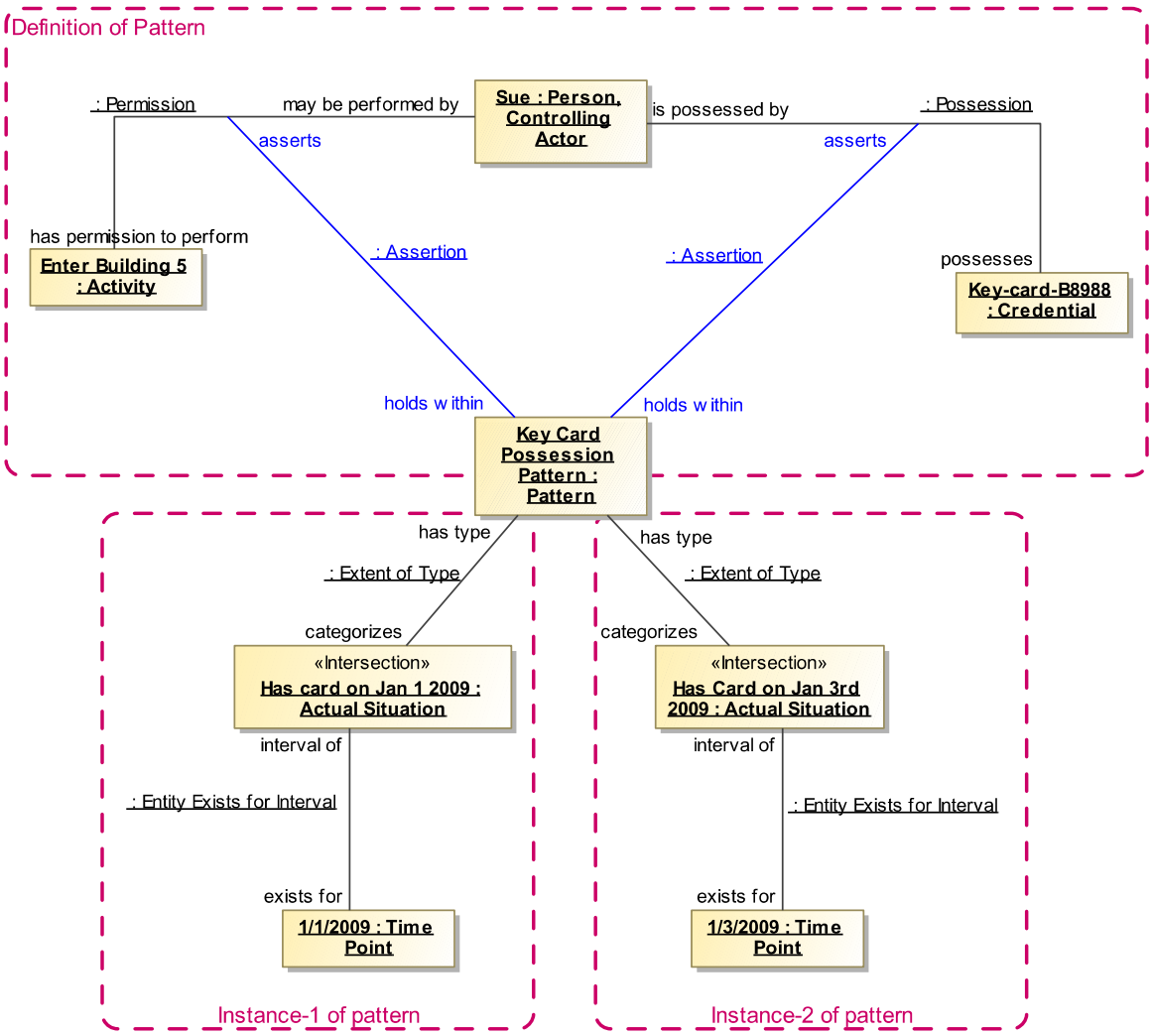


Figure 1.38: Repeated Pattern Example

In figure 1.38 we revisit Sue and her key card. We define a pattern “Key Card Possession” that asserts two things: Sue has permission to enter building 5 and Sue has possession of Key-card B8988. But, what if Sue had her card on Jan-1, lost it on Jan-2 and found it again on Jan-3rd? We have two “repetitions” of the same pattern “Key Card Possession Pattern”, each at a different time. Note we don’t know what happened on the 2nd or the 4th – those would be additional assertions. Remember, lack of an assertion does not make it false (open world assumption)

In the top box “Definition of Pattern” we see that defining this simple pattern is not that different than defining an “actual situation”, but there are no time parameters. We are declaring it as a “Pattern”.

In each of the lower boxes we see an “instance” of this pattern: e.g. “Has card on Jan 1 2009” <has type> “Key Card Possession Pattern”. This actual situation exits for the time period “1/1/2009”, since it is an instance of “Key Card Possession Pattern” we know that all assertions made for the pattern (the possession and permission) hold for what the pattern <categorizes> We know this because a type, like all context, applies its assertions to each thing it <contextualizes>. So the pattern assertions are carried forward to all its instances; both “Has card

on Jan 1 2009" and "Has card on Jan 3rd 2009" assert both the permission and the possession. In short; everything said about the pattern applies to all the pattern instances.

The pattern instances will be valid, <exist for> each of their indicated time periods.

1.9.3 Pattern Variables and Bindings

Pattern variables provide for variability of pattern contents. For each thing that may change (including relationships!) there is a pattern property.

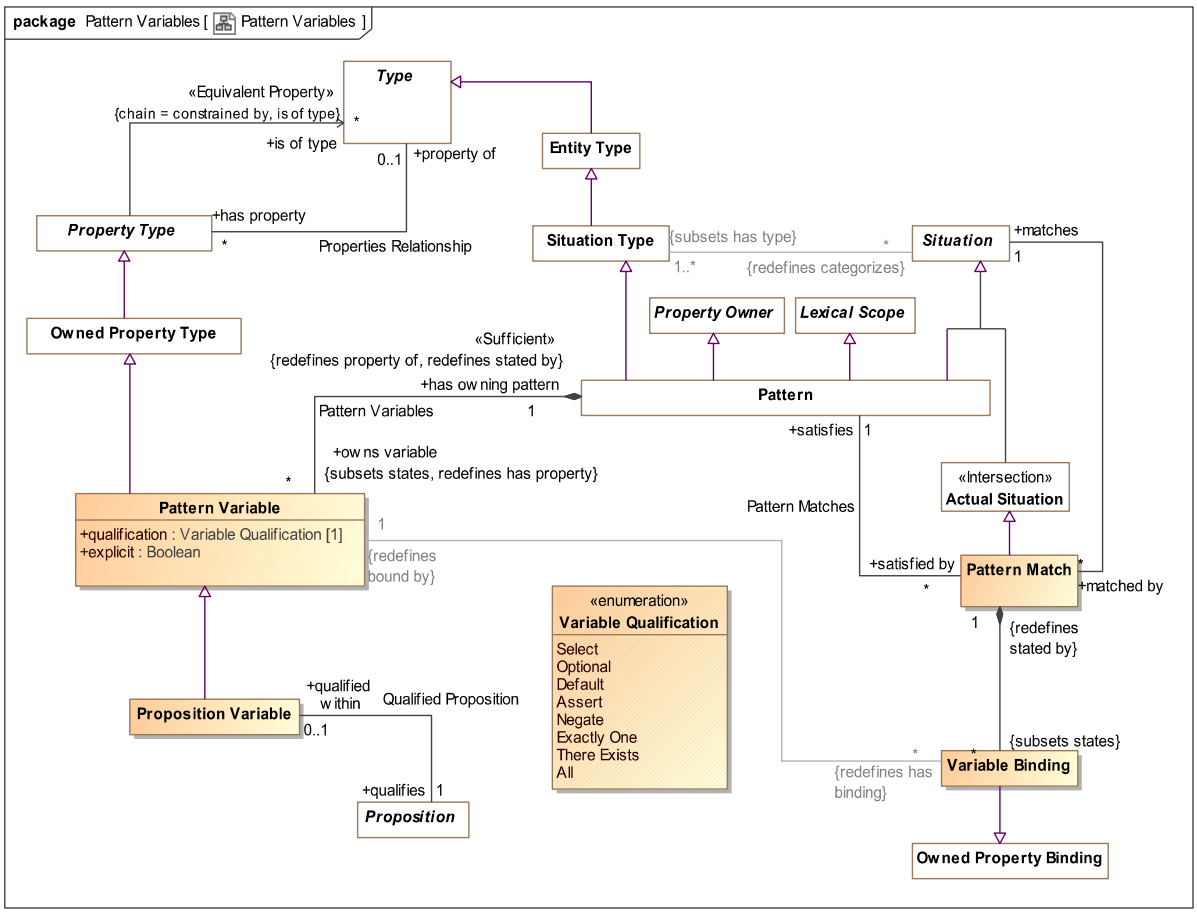


Figure 1.39: Pattern Variables

Figure 1.39 shows the definition of "Pattern Variable", the "Variable Qualification" enumeration and "Qualified "Proposition" to help define patterns and "Pattern Match" using "Variable Binding" to match patterns to situations. We will initial focus on "Pattern Variable.

Pattern variables provide a placeholder for the "real" elements in actual situations. Pattern variables specialize "Owned "Property Type" so they have a type, <is of type> and <has owning pattern> of the pattern in which the variable is defined. Pattern variables have a "quantification" that defines the semantics of the variable within the context of the pattern. We will see how these are used in the examples.

The intent of patterns is to ultimately classify actual situations; either by finding them or asserting them. Elements within the actual situations are bound to pattern variables using variable bindings. This proves that a particular pattern <classifies> an actual situation. Said the other way, the pattern is a type of the actual situation it matches based on the variable bindings in a pattern match.

Qualified Proposition provides the ability to reference some proposition, such as a Relationship, as a variable within a pattern. These propositions typically involve other pattern variables. For example, a relationship between a Coffey cup and a table is sits on is <qualified within> a qualified proposition as part of a pattern. When there is an actual Coffey cup on an actual table the Coffey cup, the table and the relation between them are abound to their respective variables.

“Pattern Match” and “Variable Binding” are used to connect a pattern and its variables with actual situations instances and the elements that constitute them as will be seen in section 1.9.6.

1.9.4 Example pattern definition in UML Profile

The SMIF UML profile provides for the expression of patterns using “structured classifiers”. Each pattern variable is either a property or connector owned by a structured classifier. Connectors are used to define “Qualified Propositions” based on associations.

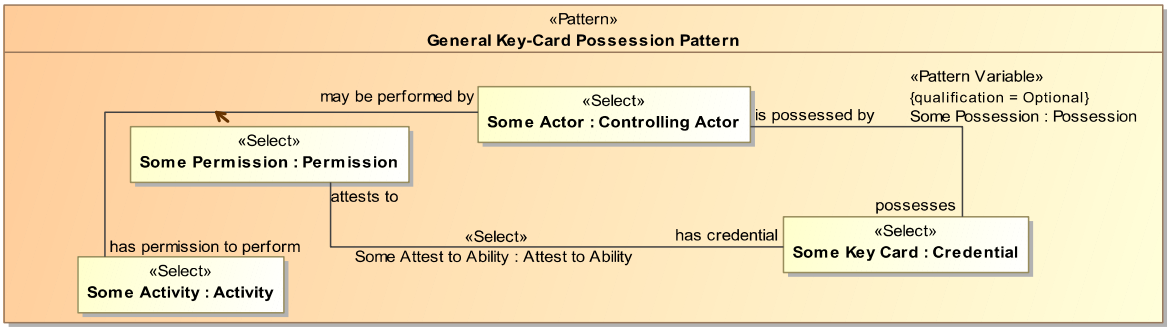


Figure 1.40: Patterns in UML Profile Example

Keeping with our example based on Sue and her key-card, figure 1.40 defines the “General Key-Card Possession Pattern”. This pattern defines “variables” for the person having permission, the activity they have permission for and the key-card. These are called “Some Actor”, “Some Activity” and “Some Key Card”, respectively. The connection between these kinds of entities are relationships; “first class” situations in their own right. For this reason we can define variables for them as well. These are called “Some Permission”, “Some Possession” and “Some attest to ability”.

Note the “box” for “Some permission” with an “Equivalent to” dependency to the permission connector line. This is required due to UML’s inability to represent connectors as association classes. To mediate this the association class is made a part that is equivalent to the connector of the same association class. This allows association classes to have connected parts as shown. This separation is not required in the SMIF model.

An implementation of SMIF will be able to “match” information about real people and permissions to these patterns.

1.9.5 Example pattern definition in SMIF model

As with all UML representations of SMIF there is a SMIF model counterpart. The model instances that correspond to the above UML example are as follows:

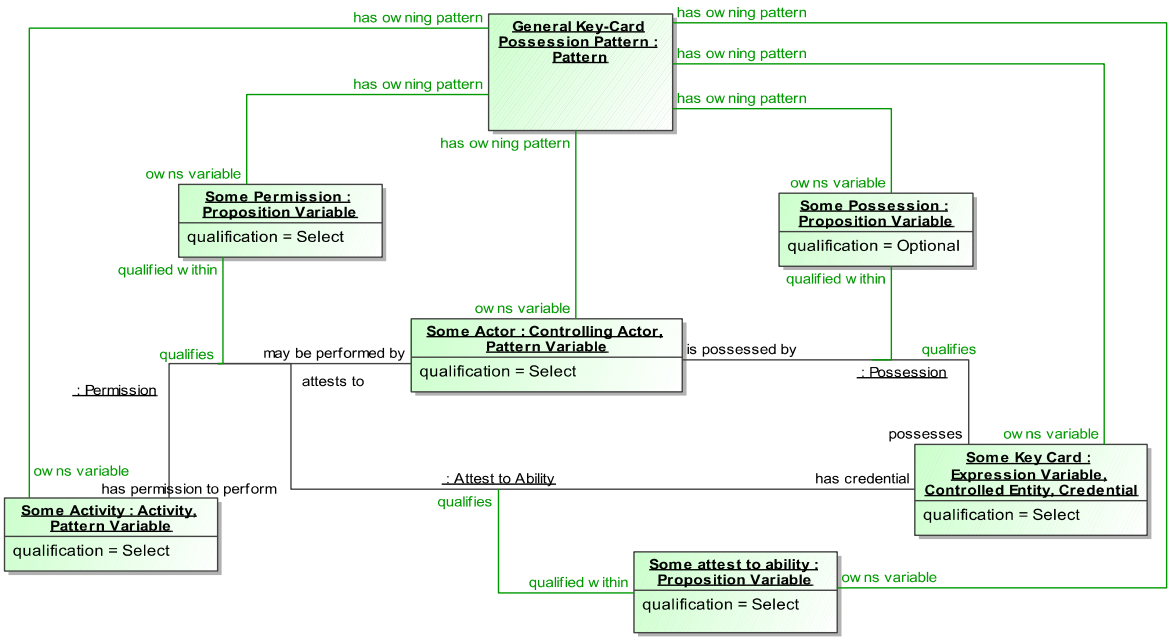


Figure 1.41: SMIF Model Example of Pattern Variables

The above model of instances of the SMIF metamodel define the pattern illustrated using the UML profile. Note that the names of the elements in the meta model instances correspond directly to those in the UML profile view. We note again that this would not be a notation used in any application.

The pattern “owns” the pattern variables, including the “qualified propositions” that provide variables for relationships. Each of the pattern variable instances: “Some Actor”,. “Some Activity” & “Some Key Card” will be “bound” to actual entities filling those placeholders.

The variables for relationships between those entities: “Some Permission”, “Some Possession” and “Some attest to ability” will be filled by actual relationships that match the form of the relationships those variables <qualifies>. Remember that relationships may be “first class” entities with their own identity, context and time frame – so it is just as important to have placeholders for them as for the more “noun oriented” entities the usually connect. The relationships referenced in a pattern provide a template for the actual relationships that fill the slots. In this way each relationship essential acts as a “sub pattern”.

As associations (not shown in this example) are not temporal it is generally not required to have an explicit variable for each one – each association or other proposition defined within a pattern will be “replicated” in each instance with its own identity.

This pattern also shown how “qualification” is used. Most of the pattern variables have a qualification of “Select”. Select will enable the pattern to “match” any configuration of elements that can fill *all* the select variables. In this example it would be any actor that has permission to perform an activity and there is a credential attesting to that permission. What is *not* required for the pattern to match is that that credential is in the possession of the controlling actor. Alone this may not make much sense, but when used for a query or as an indicator for mediation it could become meaningful. We could also mark possession as “Negate” in which case it would match all permissions where the actor *did not* have possession of the credential. Various combinations of “qualification” provide for the real capabilities of patterns.

Comparing patterns to an SQL query, “select” is like the columns in the “where clause” and “Optional” would be like the columns listed after the “Select” statement information to be returned.

1.9.6 Pattern Matching

“Pattern Match” connects a pattern with an actual situation it matches or another pattern it matches. Focusing on pattern match:

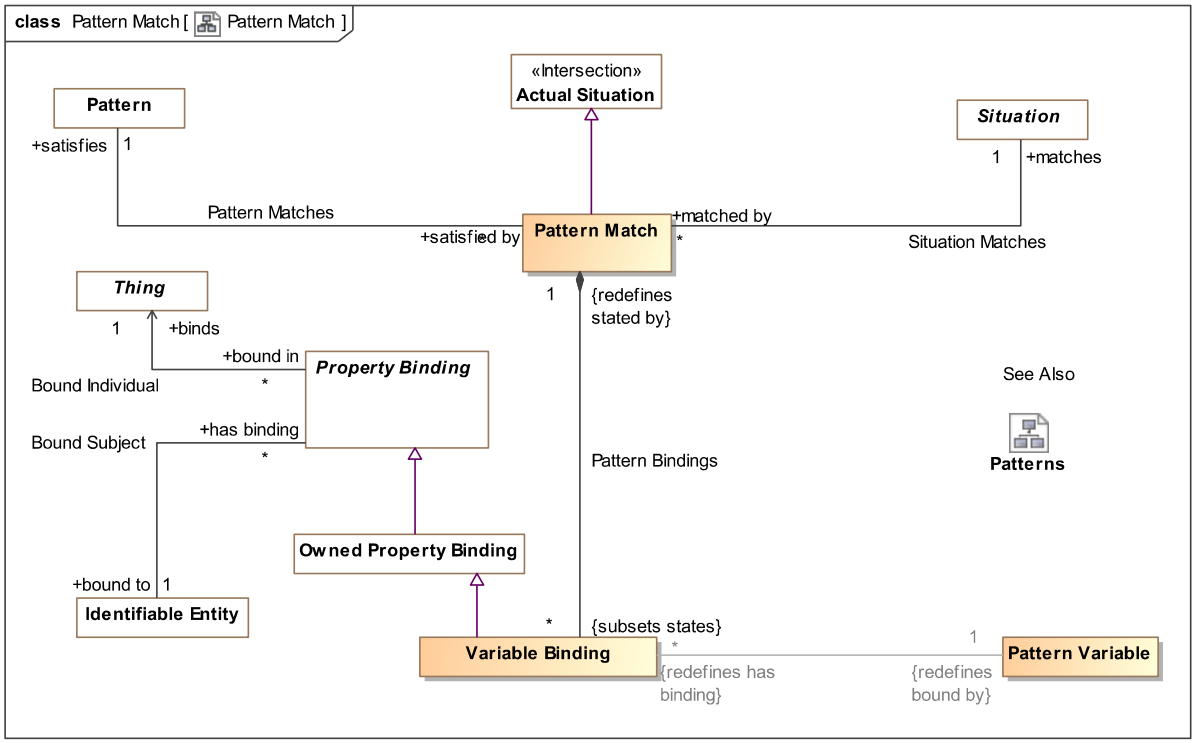


Figure 1.42: Pattern Matching Model

We see that a “Pattern Match” <satisfies> a pattern that <matches> a situation that is the instance of the pattern. This is supported by a set of “Variable Bindings” to “Pattern Variables” that the pattern match <states>.

Variable Binding builds on the general concept of a “Property Binding” in that it <binds> something. What it <binds> is <bound to> some entity based on the the property it is <bound by>. This is similar to the RDF/OWL concept of a “triple” with the exception that bindings are directly or indirectly identifiable. Variable Bindings are bound within the context of a pattern match (which could be represented as a named graph in RDF, but there are other approaches to structures in RDF).

So within a Pattern Match, each variable is bound to one or more individuals that satisfy the constraints of that variable.

1.9.7 Pattern Matching Example

Consider a situation we have already seen: Sue and her Key-card. We will consider if it could be an instance of the pattern defined in section 1.9.5.

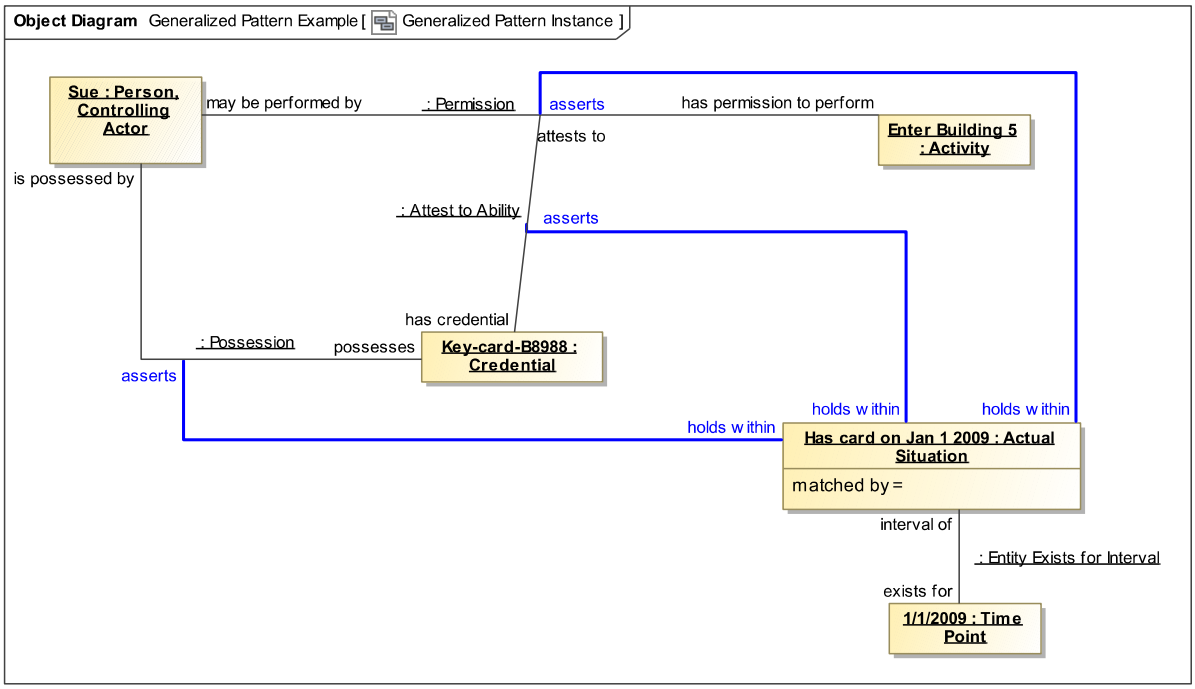


Figure 1.43: Potential instance of a pattern

The above defines 3 “entities” and three relationships.

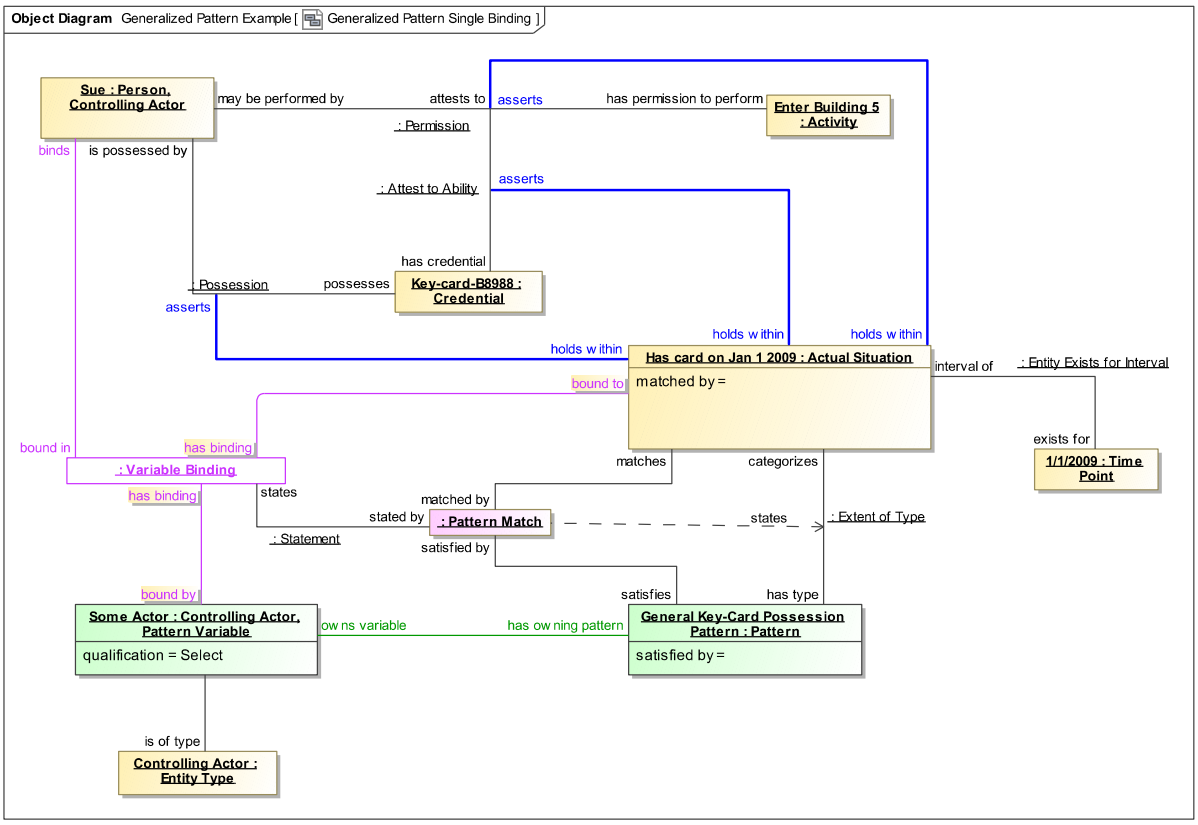


Figure 1.44: Binding of a single variable

Figure 1.44 shows a pattern match and the binding of just one pattern variable (Some Actor) to one actual person (Sue). Just one is shown because all the elements for a pattern binding become somewhat messy to diagram – it is not something most people need to look at other than to understand the concept. At this point we are just showing one of the variable bindings, all will be required to match the pattern.

The “Pattern Match” is shown as it <matches> the “Has card on Jan 1 2009” actual situation and this situation <satisfies> the “General Key-Card Possession Pattern”. The “Pattern Match” <states> a “Variable Binding” that <binds> “Sue” <bound by> the “Some Actor” Pattern Variable which is <bound in> the “General Key-Card Possession Pattern”.

Note also that as a convention of showing these instance diagrams in UML the type of the pattern variable is shown as a one of the types of the variable – this is required to satisfy UML’s rules for instance diagrams. In the actual model the variable <is of type> the type of the variable.

The above shows just one of the six variables being bound.

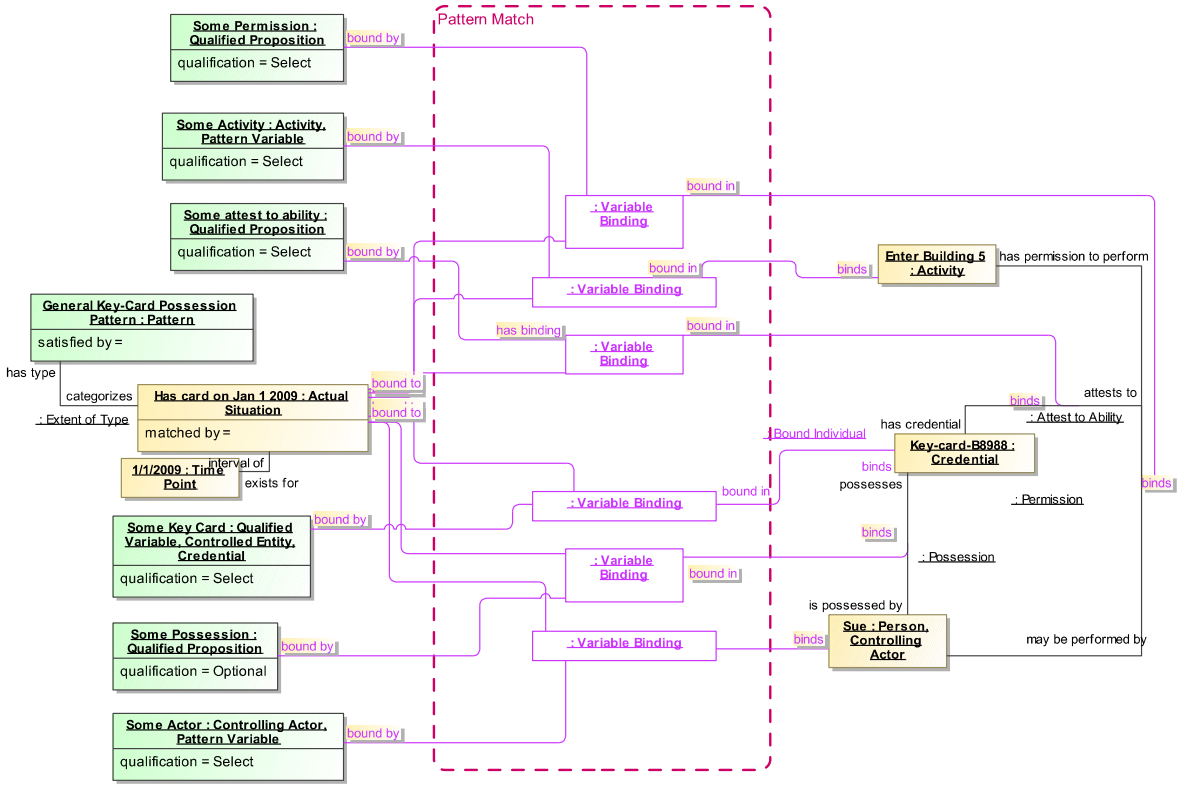


Figure 1.45: Full Binding of Pattern

Figure 1.45 shows all the variable bindings owned by the pattern match. As we said, it is not that readable but provided for reference. The set of all the bindings "proves" that the actual situation matches the pattern.

1.9.8 Computed Variables

Some variables in a pattern are computed based on other variables. Variables may be computed using either Expression Variables or Subset Variables.

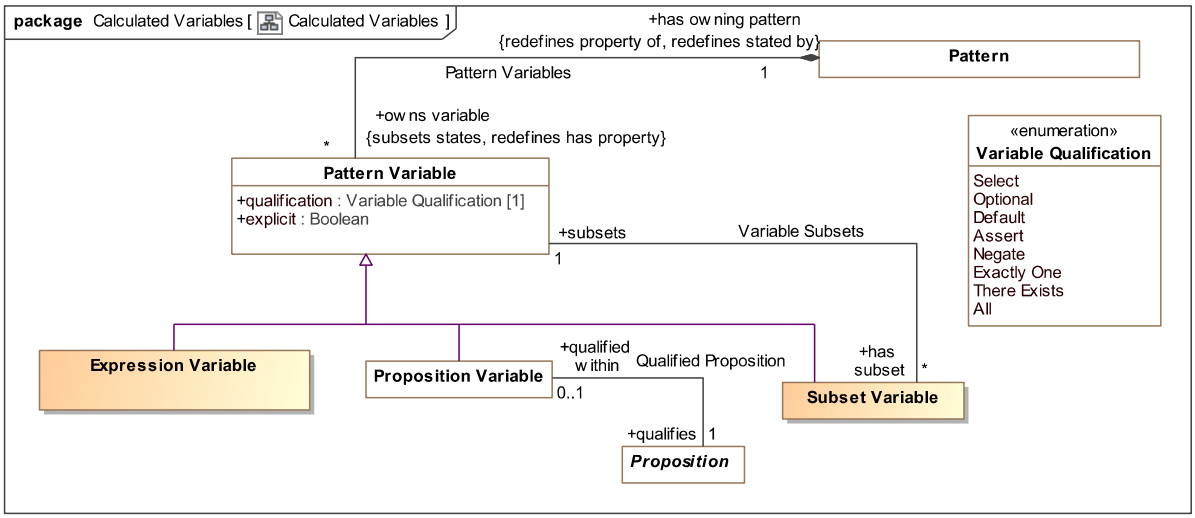


Figure 1.46: Subset and Expression Variables

Figure 1.46 shows the definition of subset and expression variables. Both of these types compute the value(s) of a variable based on other variables.

Subset Variable uses a base variable it <subsets> and applies additional constraints to the base such that the subset variable holds only those values that conform to these constraints. The most common constraint is probably the type of the subset variable as defined by <is of type>. The subset may also have required (<select>) characteristics and relationships as well as a general <condition> expression.

Expression Variable defines a <computation> expression that provides the value(s) for the variable. Note that as expressions may not be “reversible”, it may not be possible to assert an expression variable. The ability to assert ot map to an expression variable is implementation specific.

1.9.9 Subset Variable Example

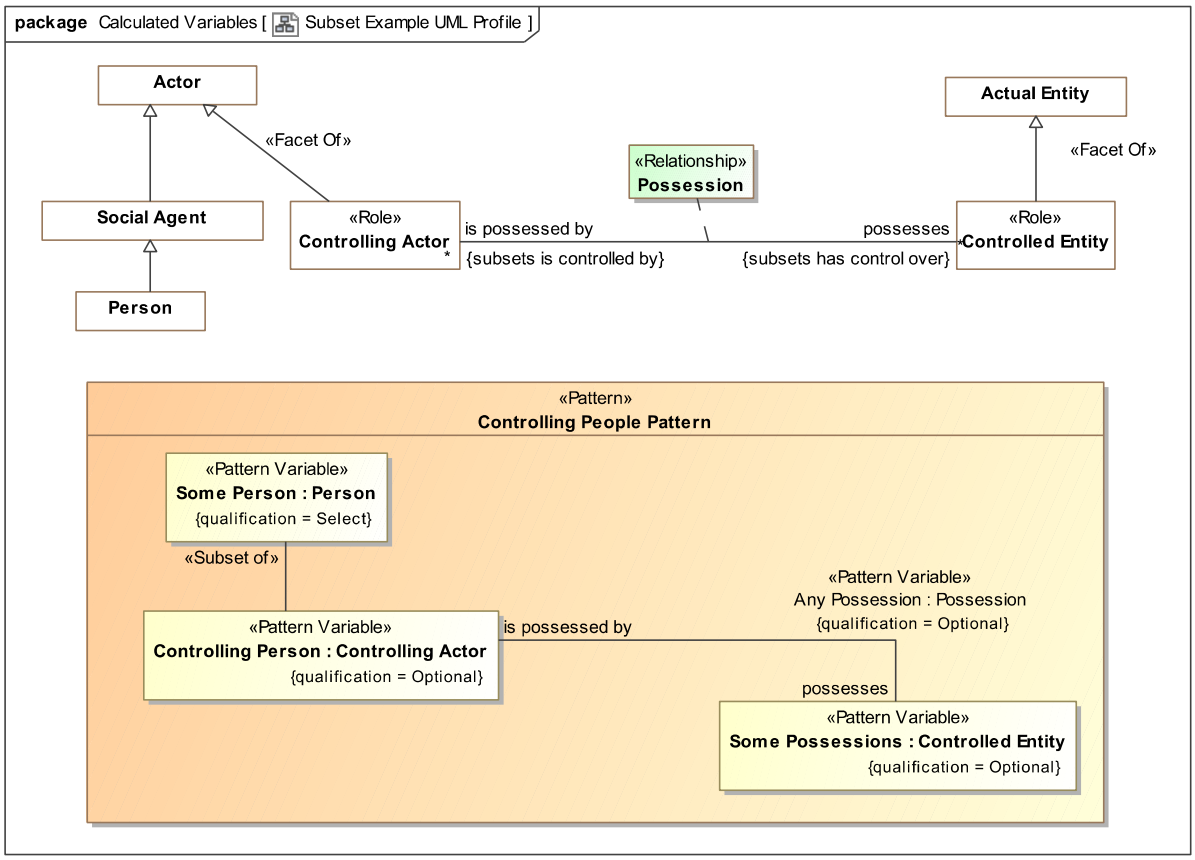


Figure 1.47: Subset Variable Example in UML Profile

Figure 1.47 shows a little more context for the "Possession" relationship we have been using as well as the "Controlling People" pattern that uses this relationship.

Controlling actor as a role

Note that "Controlling Actor" is defined as a "Role" and that this role is a <<Facet Of>> an Actor. Note also that "Person" in an indirect subtype of "Actor". A role is a dependent classification of some entity. What this model says is that any particular actor may be a "Controlling Actor" in various context or timeframes but that an actor is not necessarily a controlling actor. The controlling actor role may come and go with regard to any particular actor, such as "Sue". By saying that "Controlling Actor" is a <<Facet Of>> an actor, we are saying that only an actor may play this role. You must first be an Actor, then you can be a controlling actor. Since Actor is not a role (or other kind of facet, we will explore this in another section), an actor is always an actor – it is essential to their nature. Since "Person" is an indirect subtype of actor, a person may be a controlling actor. Since "Possession" is related to "Controlling Actor", any person that possesses something is implicitly a controlling actor – they control what they possess.

Not shown in this diagram is that possession is a subtype of "Control" which relates the same roles. So there may be some controlling actors that don't possess anything (this could be added to the pattern but we are trying to keep it simple).

Likewise, playing the role of a "Controlled Entity" can be played by any "Actual Entity".

The “Controlling People Pattern”

“Controlling People Pattern” is a pattern that starts with a “Select” of a “Person”. All people will “Match” this pattern. Besides matching people we want the pattern to tell us if each person is a controlling actor and, if so, anything they possess. We may want to use this pattern for a query or some kind of data mapping.

For each matched pattern there will be exactly one value bound to “Some Person”. If that person “plays the role” of a “Controlling Actor” then the “Controlling Person” variable will be populated *with the same person*. Said another way, the set of all “Controlling Persons” is a subset of all “Some Persons” based on “Some Persons” playing the role “Controlling Actor”.

All “Possession” relationships from “Controlling Person” will be bound to the “Any Possession” relationship and the “Some Possessions” variable. Note that “Any Possession” and “Some Possessions” may be bound to multiple values.

1.9.10 Controlling Person Pattern in the SMIF Model

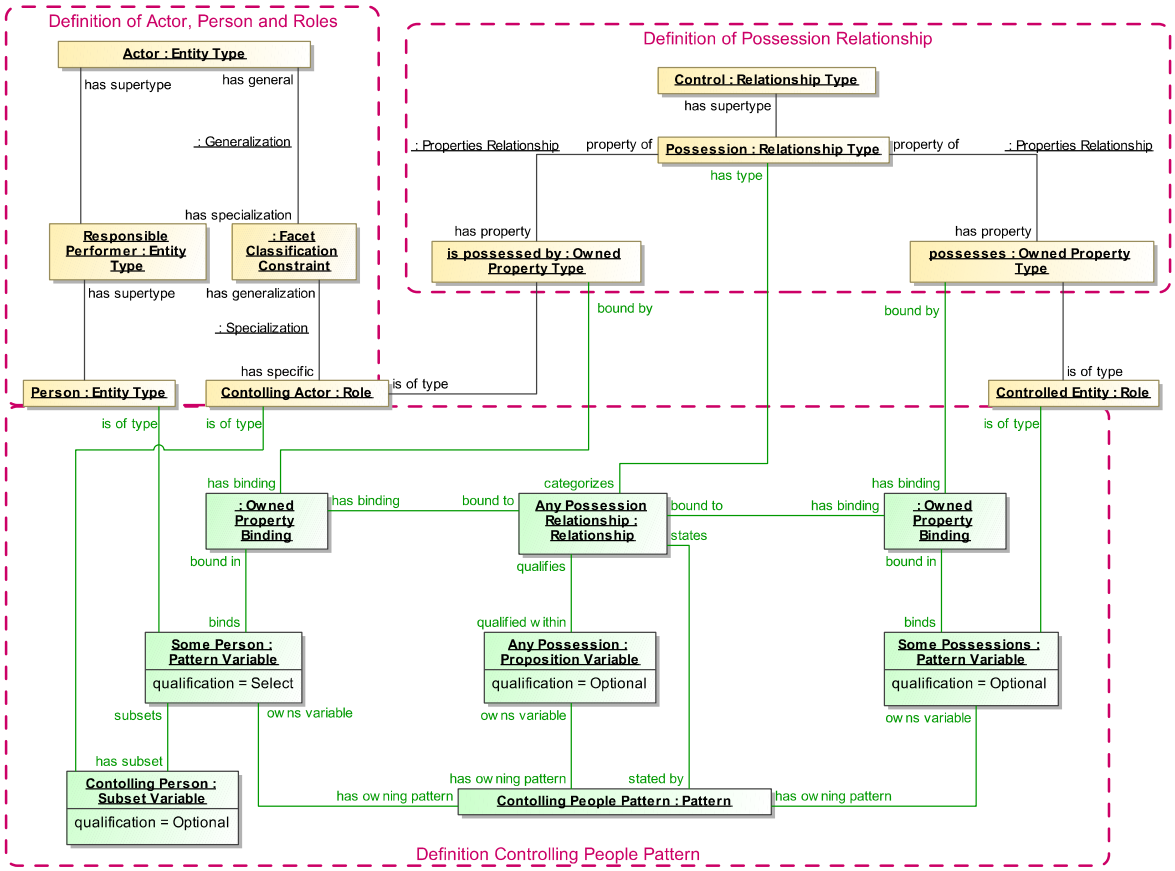


Figure 1.48: SMIF Model Instances of the Controlling Actor Pattern

The UML Profile diagram in figure 1.47 has a corresponding model as instances of the SMIF model as shown in figure 1.48. Notice that “Controlling Actor” is a “Role” and that it is constrained, using a “Facet Classification Constraint” to be a role of an “Actor”. Person is an indirect subtype of Actor.

The “Controlling People Pattern” owns a “Select” property “Some Person”, so all persons will be matched by this pattern. The optional “Controlling Person” variable will have a value only if the person matched in “Some Person” is also a “Controlling Actor”. That Controlling Actor Person may then have “Some Possession” relationships with “Some Possessions”. Note that as “Controlling Persons” are selected there may be multiple values bound to “Any Possession” and “Some Possessions”.

1.10 Mapping

Mapping defines how different concrete or logical data structures represent the same or related concepts as defined in a conceptual reference model. By “grounding” the meaning of the data structures in common concepts SMIF provides a foundation for integrating information from multiple sources, translating between data representations and federating information for analytics.

The basis of federation is patterns (See section 1.9). A mapping is a pattern that defines a correspondence between two sub-patterns, one “Concrete” and one “Reference”. The concrete pattern shows how data structures represent reference concepts. The sub-patterns are typically derived from different, independent, models. The patterns can then operate bidirectionally (as ling as no functions are used) – if data in the concrete source changes a SMIF implementation can update instances of the reference model. If instances of the reference model change, a SMIF implementation can update the concrete data source.

When using the SMIF UML profile the both models are loaded into UML. The mapping patterns are then represented using UML composite structure diagrams augmented with profile support. We will start with an example expressed using the UML profile and then see how it is represented in the SMIF model.

1.10.1 Mapping Components Example

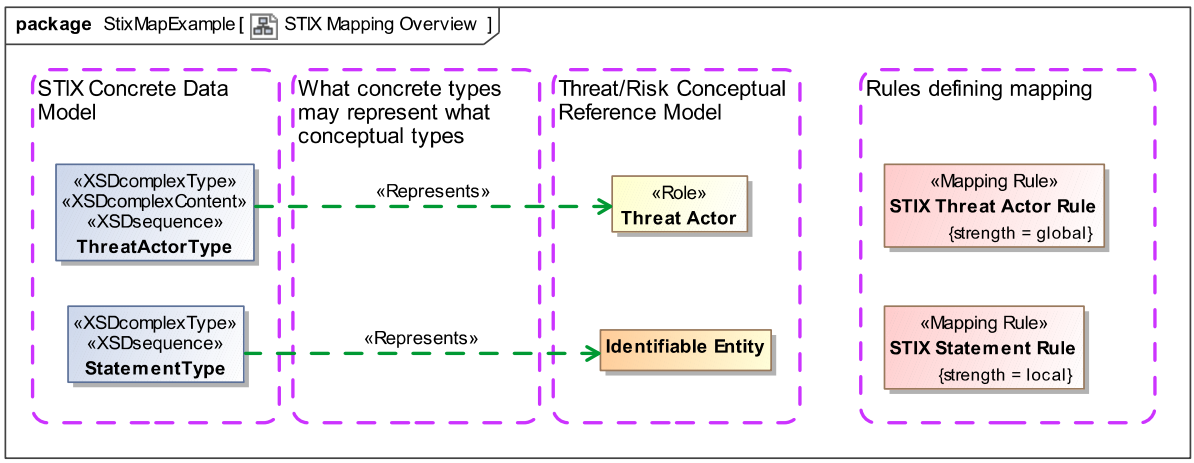


Figure 1.49: STIX Mapping Overview

Our example will focus on the mapping of “threat actors” and “statements” as defined in the “Structured Threat Information Expression” (STIX) XML schema. The STX schema and everything it references are imported into UML using off-the-shelf UML tool capabilities. This results in a UML model that directly reflects this XML schema and can then be used as the basis for mapping to the operational threat risk conceptual reference model (OTR).

Figure 1.49 shows the “top level” of this mapping and the basic components of any mapping.

- A mapping connects two models, one considered the “concrete” model and once considered the “reference” model. The concrete model is the one more specific to a technology, system or solution where as a reference model is more conceptual and less committed to concrete concerns. In this and many cases this choice is reasonably clear. There are cases where the models are at about the same level of abstraction, in which case the choice may be arbitrary – but making a choice of which to call the “concrete” model and which to call the “reference” model is required.
- The “STIX Concrete Data Model” is the concrete model, it is directly derived from an XML schema intended for information exchange between STIX enabled systems. Classes within this model are “ThreatActorType” and “StatementType”.
- The “Threat/Risk Conceptual Reference Model” (OTR Model) is the reference model in this example. This model has been constructed with other models (including STX), threat relevant literature and stakeholder input to capture the concepts represented by STIX and other data models. The goal has been to identify and define the concepts the data is representing and model them as best understood by stakeholders across multiple domains of interest impacting the analysis of threats and risks. The OTR model is not a data model, it is a model of the domain as stakeholders conceive it. This conceptual model is used to “pivot” between different data representations.
- A data element, such as ThreatActorType is intended to model data that represents something in the “real world” (or perhaps a world we imagine as possible, but based on the real world). For each “real world” concept that a data element represents, a <<Represents>> stereotype is defined. This says that t*he data structure contains some information about the represented concept*. <<Represents>> is not generally sufficient to fully map data, that is not its intent, it is a declaration that it can contain information about some concept which is then used to filter the more detailed mapping rules such that only things that represent a concept are mapped to it. We will see how this works, below. In the example we see that the STIX class “ThreatActorType” <<represents>> a real-world threat actor as defined in the OTR model. We also see that a STIX “StatementType” *can* represent information about any “Identifiable Entity” as defined by the OTR model.
- The rightmost column “Rules defining mapping” shows the specific mapping rules that define how and when the STIX data types represent the OTR concepts. This “insides” of these rules are UML structured classifiers that comprise the rule details. In an implementation of SMIF these rules help implement the translation and federation of data defined using the mapped data models.
- One other element that can be seen at this level is the “strength” of the rules. Strength defines when a rule is asserted (triggered). The “STIX Threat Actor Rule” has “strength-global” which means it always applies and can be triggered by any data source changing. The “STIX Statement Rule” has “strength=local” which means it is only triggered when required to fulfill another rule – such as the threat actor rule. A 3rd possibility is “strength=default” which defines a rule that is only triggered if no other rule has mapped the elements.

The above example should make clear these basic mapping elements; the concrete and reference models, how data elements may represent concepts and the top-level identification of mapping rules. In the following sections we will look at each of these in more detail.

1.10.2 STIX Concrete Data Model

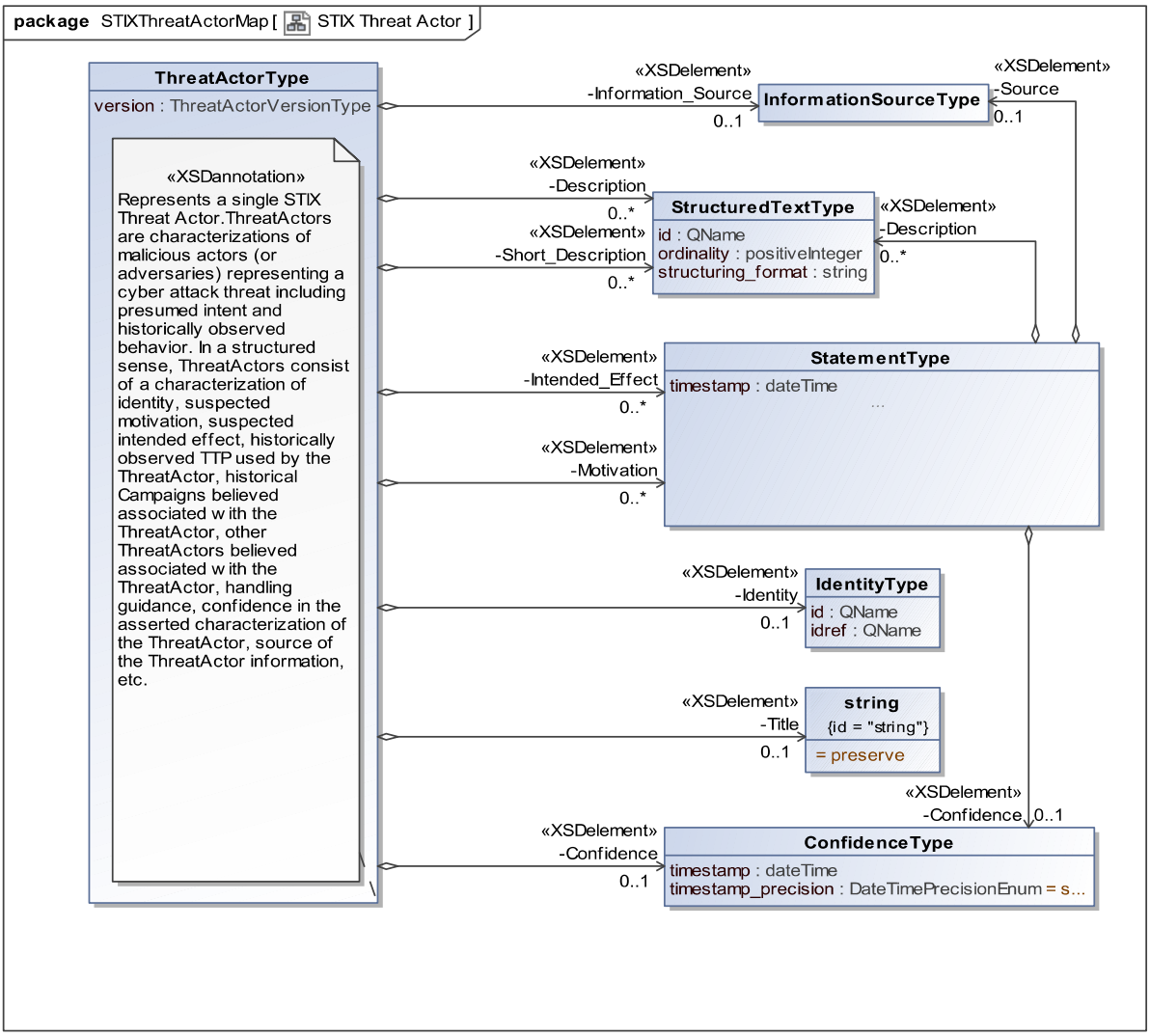


Figure 1.50: STIX Concrete Model Fragment

Figure 1.50 shows a fragment of the STIX model dealing with threat actors. As noted above, this is directly derived from the STIX XML Schema. For the example we will not delve to much into the specifics of the STIX model, it is presented so that the mapping example can be better understood.

There are a few things to note about this model. First, there is a good correspondence between "ThreatActorType" as defined here and the general concept of a threat actor. As is typical, the STIX definition makes threat actor specific to Cyber threat actors, for the purposes of STIX. This is more narrow than the general concept of a threat actor – which could cause many kinds of mayhem.

We also note that information about the threat actor (the real actor in the real world) is intermixed with metadata about the threat-actor information, such as confidence. Other than understanding the concepts, there is no deterministic way to know that "Motivation" is probably about the actor where as "Confidence" is about the information record. One of the challenges of mapping is untangling these mixed concerns.

Note that "StatementType" is used in the STIX model for "Intended effect" and "Motivation". In STIX "StatementType" is used where there are no explicitly modeled data elements for a concept, a StatementType just provides a definition and some metadata for these concepts. In other models these elements may be explicitly

modeled, a challenge for integration. Having a motivation or metadata about confidence is not at all specific to threat actors so the OTR model defines these concepts at a much higher level so that they can apply to anything that the would make sense for that concept. So having a name can apply to anything we can identify where as only a threat actor can perpetrate a disruptive action (based on how these concepts are defined in OTR). Conceptual reference models are define concepts free of the context of a particular application or data structure, reflecting their meaning to stakeholders.

As with all UML models, properties or relationships that are defined for a “supertype” (or superclass) apply to all subtypes. So from this diagram we can see that a “threat actor” (like any identifiable entity) may have a name but that if you perpetrate a disruptive action you must “play the role” of a threat actor that must be a “social agent” (person or organization). We will delve into roles, below.

1.10.3 OTR Conceptual Reference Model

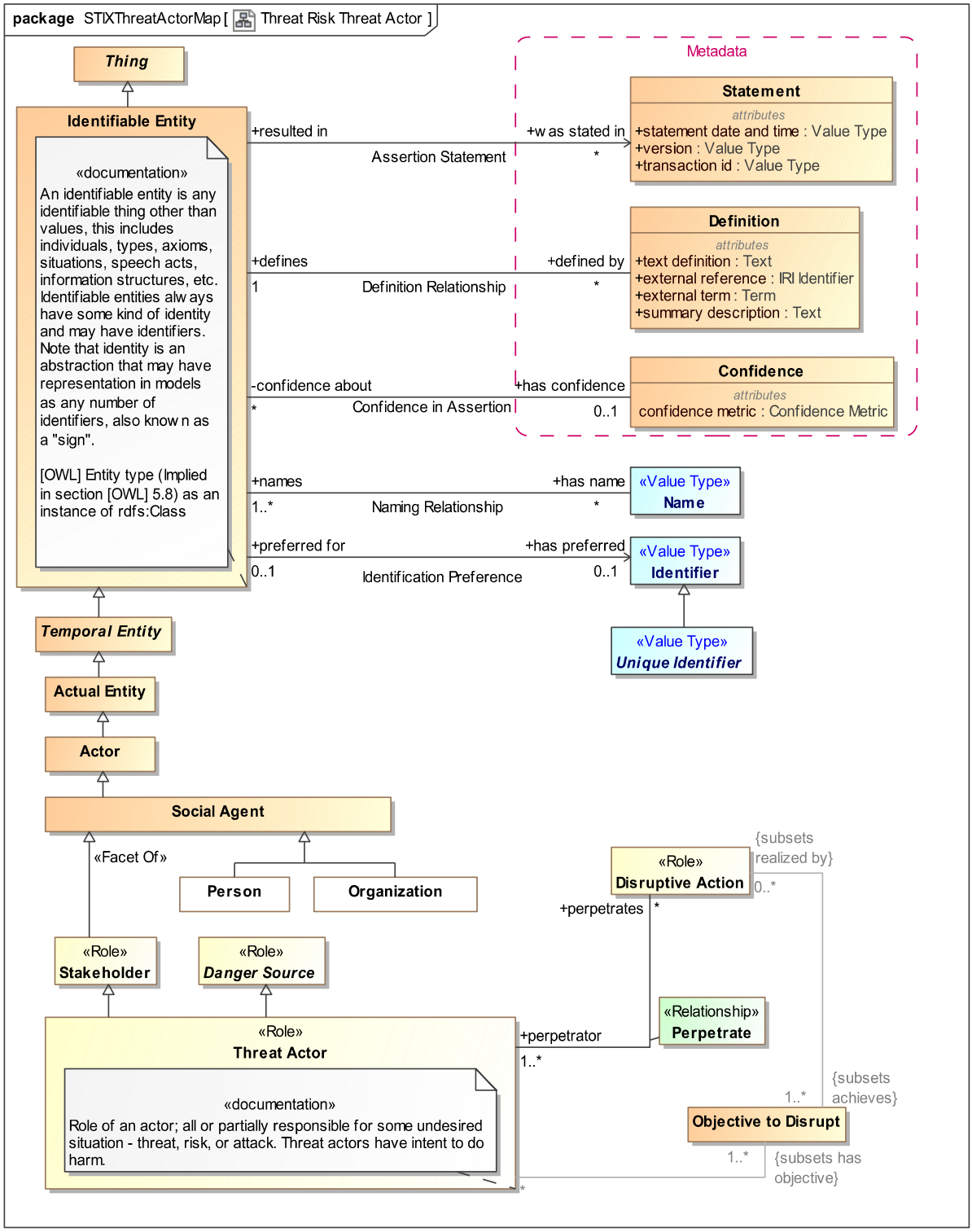


Figure 1.51: OTR Conceptual Reference Model Fragment

The OTR model fragment is figure 1.51 shows the properties and relationships that will be used to map concepts in the STIX model. Note that these two models are of a very different "shape", use the same or different terms but clearly have commonality. A fundamental difference is that in the OTR model concepts are defined for their most general interpretation – This is how OTR is structured, how general to define reference concepts is a decision left to the model authors..

Another thing to note is that not all the information in either model is "complete" relative to the other. The purpose of conceptual reference models is to capture *shared concepts* across domains and different data models. The concepts that are mapped between different data models can be mapped – the others are ignored or must be populated with data specific rules. It is simply not practical for every concept of every data model to be mapped – so we don't try.

It should also be noted that there is no "required" reference model, any number of reference models may be used to accomplish some mapping. OTR is a reference model for threats and risks, it does not claim to be the only one possible or useful.

As with the STIX model, we will not look to deeply into the specifics of the OTR model semantics as our purpose here is to use these model fragments as part of the mapping example.

1.10.4 STIX / OTR Mapping Rule

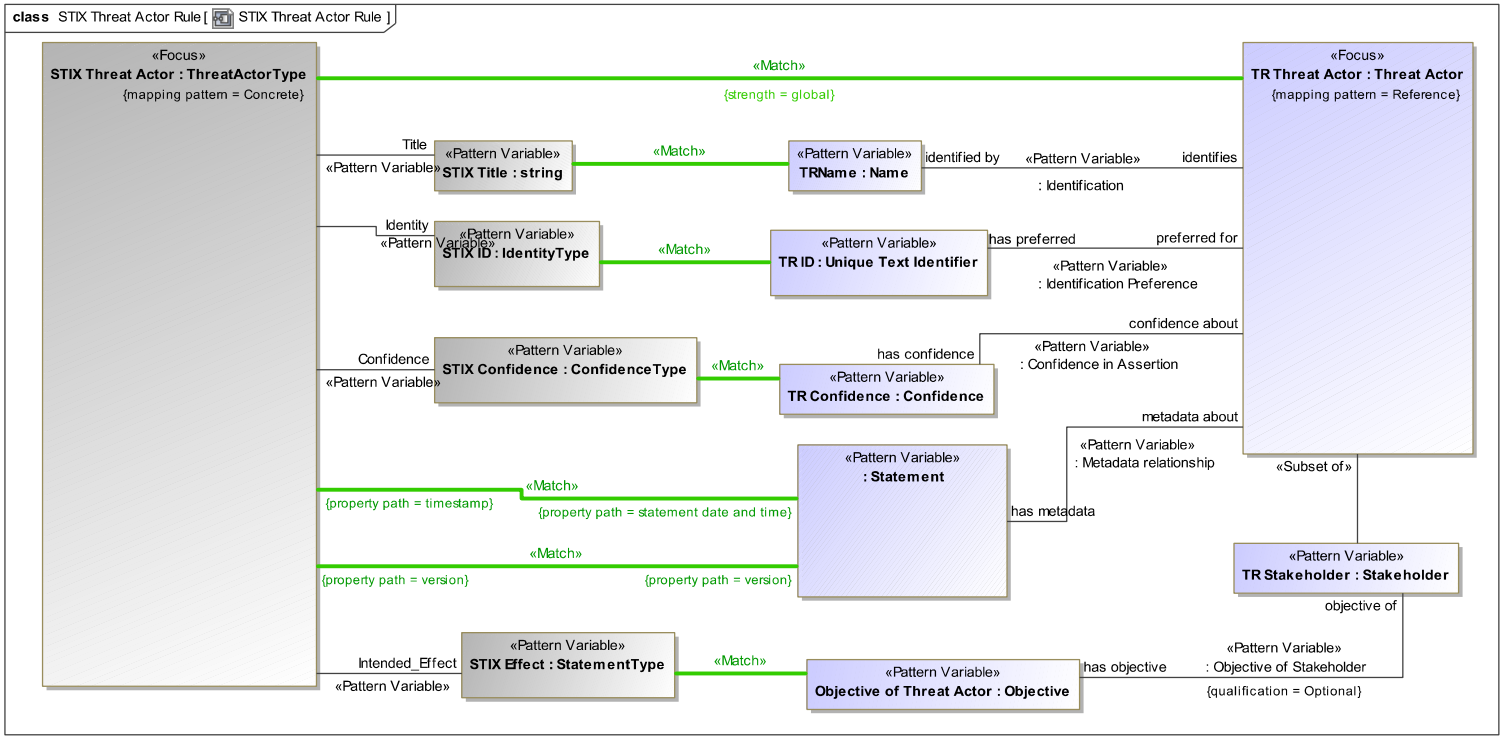


Figure 1.52: STIX - OTR Threat Actor Rule

Figure 1.52 shows the detailed mapping of STIX ThreatActorType to OTR Threat Actor as the "composite structure" of the "STIX Threat Actor Rule" we saw in the summary. It does this using pattern variables, relationships and "Match" rules.