Date: December 2016

Semantic Modeling for Information Federation (SMIF)

Version 0.9

OMG Document Number adtf/2016-12-01

Normative Reference: http://www.omg.org/spec/SMIF

Associated Normative Machine Consumable Files:

adtf/2016-12-02

For latest version please see:
https://github.com/ModelDriven/SIMF/tree/master/NextSubmission

---

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents, Report a Bug/Issue (http://issues.omg.org/issues/create-new-issue).

Table of Contents

Table of Figures

Model diagrams not indexed for unknown reasons.

Preface

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable, and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language®); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel™); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at http://www.omg.org/.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from the OMG website at:

http://www.omg.org/spec

Specifications are organized by the following categories:

Business Modeling Specifications

Middleware Specifications

- CORBA/IIOP
- Data Distribution Services
- Specialized CORBA

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

- UML, MOF, CWM, XMI
- UML Profiles

Modernization Specifications

Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications

- CORBAServices
- CORBAFacilities

OMG Domain Specifications

CORBA Embedded Intelligence Specifications

CORBA Security Specifications

Signal and Image Processing Specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
109 Highland Avenue
Needham, MA 02494
USA

Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: pubs@omg.org

Certain OMG specifications are also available as ISO standards. Please consult http://www.iso.org

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman/Liberation Serif – 10 pt.: Standard body text

Helvetica/Arial – 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

Courier – 10 pt. Bold: Programming language elements.

Helvetica/Arial – 10 pt: Exceptions

NOTE: Terms that appear in italics are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification via the report form at:

http://issues.omg.org/issues/create-new-issue

This page intentionally left blank.

Submission-specific material

0 Submission Specific Material

0.1 Submission Introduction

The SMIF submission team is pleased to present a revised submission to the "Semantic Modeling for Information Federation" Request for Proposal ad/2011-12-10

The IPR mode for this submission is **Non-Assert**.

Clause 0 of this document contains information specific to the OMG submission process and is not part of the proposed specification. The proposed specification starts with Clause 1. All clauses are normative unless otherwise specified.

0.2 Submission Team

0.2.1 Submitters

The following companies submitted this specification:

- Data Access Technologies, Inc. (Model Driven Solutions Division)
  - Cory Casanave
- No Magic, Inc.
  - Jim Logan
- PNA-Group, Ltd.
  - Sjir Nijssen
- 88 Solutions
  - Manfred Koethe
- Thematix Partners LLC
  - Elisa Kendall

0.2.2 Contributors & Supporters

Contributors

- Tibco Software Inc.
  - Paul Brown

0.3 Proof of concept

No Magic has a released product implementing most of the SMIF profile and OWL mapping. Prototype efforts for mapping are expected but have not yet fully validated the model and mappings.

0.3.1 Resolution of Mandatory requirements

6.5.1.1 Proposals shall define the SMIF Conceptual Model as a model of the concepts required to model information and achieve federation using SMIF. This model shall be a conceptual domain model of SMIF itself, expressed in the SMIF Notation (see requirement 6.5.2.2).	The conceptual models are specified in clause Error: Reference source not found using UML/SMIF notation expressed using the SMIF conceptual modeling profile.

6.5.1.2 The SMIF Conceptual Model shall define the concepts necessary for creating conceptual domain models (CDMs), sufficiently general to express the semantics being represented by the information modeling constructs in the languages identified in requirement 6.5.3.1, including the following capabilities:	Due to the similarity of needs, the SMIF model is one language that may be used for different levels of abstraction. The level of abstraction and purpose of a specific model is specified.
a. General capabilities for modeling all relevant aspects (i.e., all rules, laws, etc.) of concepts, including (but not necessarily limited to) the definition of: individual things, relationships, classification of individual things (including multiple classification), sub-classification and inheritance (including multiple inheritance), roles (that describe how individual things are involved in various processes, compositions and relationships), composition and constraints.	Capabilities relevant to information federation are included. Modeling of "all rules and laws" is considered out of our information federation scope, but follow-on efforts could address additional concerns. Concepts applicable to rules and laws may be modeled like any other concept. All the following are included: • Individuals • Relationships • Classification (of anything) • Multiple classification • Sub-classification (Generalization), including multiple inheritance. • Roles • Constraints Specific models of compositions as patterns are intended to be included in SMIF models.
b. Definition of one or more names by which users refer to a concept, as well as one or more *separate* reference identifiers that would normally be hidden from users. (This is required to maintain the stability of concept references across multiple languages, communities and viewpoints.)	Concepts may have any number of names and identifiers. Names and identifiers may be scoped by a context.
c. Definition of the *context* of concepts, allowing for the grouping of concepts such that no single dominant decomposition is required (that is, in addition to just a hierarchical grouping, allow for a multi-dimensional separation of concerns [Ossher1999] delineated by multiple contexts).	Context is a first-class SMIF concept that relates a set of things to applicable assertions that hold for them. Something may be in any number of context across multiple contextual dimensions.
d. Definition of *patterns* of reusable, parameterized conceptual structures and the use of such patterns within a context.	Patterns are a first-class concept.
e. Definition of *units* that describe what can be measured about various conceptual *quantities* and asserting that some conceptual quantity is measured in specific units.	Units are represented using unit types bound to quantity kinds.
f. Ability for *federated definition* of concepts; that is, allowance for the definition of a concept in a CDM such that it can be modified and/or extended across multiple contexts and models.	SMIF uses an open world assumption that may be closed in a specific context. As such definitions may be federated and extended.

6.5.1.3 The SMIF Conceptual Model shall define the concepts necessary for creating logical information models (LIMs), capable of representing information context, information structures, integrity rules, derivation rules, views and viewpoints as may be found in the languages referenced in requirement 6.5.3.1, but not be bound to any particular data representation or schema language, including the following capabilities:	Due to the similarity of needs, the SMIF model is one language that may be used for different levels of abstraction. The level of abstraction and purpose of a specific model is specified.
a. Usage of one or more terms and/or concepts defined in a CDM, as identified by MBRs between a LIM and the CDM, to define the semantics of information elements in one or more LIMs.	A CDM concept may be represented by any number of LIM concepts.
b. Identification of concepts from a CDM ("what can be known about a subject domain") as being required or optional in a LIM ("what may or must be included in a particular information structure"), with appropriate cardinalities.	A CDM specifies the semantics of the domain, not the data. Data cardinalities may be different from real-world cardinalities. What we can know, must know and do know may be independent. This is accomplished by using the "represents" relation and mappings.
c. Ability for different LIMs related to the same CDM to represent different (and possibly incompatible) subsets of information about conceptually the same things (as semantic precision does not imply universal agreement).	See response to (b).
d. Ability for a LIM to *close* the definition of a concept that has a federated definition in the related CDM, fixing it relative to a specific context in the CDM relevant to the LIM. (Once a definition is closed, it can then be assumed that no further statements will be made about that concept within the context relevant to a particular LIM thus allowing for the application of defaults and constraints impacting that concept.)	SMIF rules operate on a closed set of models based on their context whereas the models may be extended or refined in other contexts. Context is the foundation for closing the world.
e. Ability to define *viewpoints* that specify *views* on a CDM or LIM that act as effective contexts for a particular purpose relevant to one or more other LIMs, including formation of views from composite concepts.	Each LIM is effectively a viewpoint that is mapped to the underlying CDM using MBRs. See Error: Reference source not found.
6.5.1.4 The SMIF Conceptual Model shall define the concepts necessary for creating model bridging relations (MBRs), sufficient to enable independently conceived models at all levels (CDM, LIM, PDS) to be federated, such that the similarities and differences between elements defined in each can be expressed, including the following capabilities:	Due to the similarity of needs, the SMIF model is one language that may be used for different levels of abstraction. The level of abstraction and purpose of a specific model is specified.
a. Ability to relate identical and similar information concepts that have been independently conceived and represented in information models using the same or different information modeling languages or physical schema.	Identical and similar concepts are mapped using representation and mapping rules. The mapped models may or may not be independently conceived.
b. Ability to handle differences in name, structure, representation, property sets and underlying semantic theories.	Mapping rules provide for differences in naming and structure. Mapping rules may be defined between compatible semantic theories. SMIF

c. Ability to relate the same information across views that share the same underlying concepts and to specify one view of a model from another (*projection*).	See the response (b).
d. Ability to state the *purpose* for an information structure in one model relative to the related structure in another model. (Examples of purposes include creating, reading, updating and deleting recorded information and providing a snapshot in time, measurement, expected value or required value of a property of or association between information records.)	The purpose of a mapping may be specified in the textual documentation of a mapping; no other support is deemed necessary.
6.5.1.5 Proposals shall define a Kernel as a subset of the SMIF Conceptual Model with the minimum set of foundational concepts necessary in order to precisely define all other concepts within the SMIF Conceptual Model. Proposals shall provide a formal logic interpretation of the semantics of the SMIF Kernel, expressed in a formal logic such as Common Logic as defined in ISO standard 24707.	The kernel is defined as a subset of the SMIF model expressed in the diagrams under the package "Kernel". The kernel is defined as a mapping to the fUML subset of UML, which provides a formal logical interpretation of the semantics in Common Logic. As the kernel is also specified in UML, no specific mapping is required. Like MOF, fUML does not comprehend SMIF subsets and redefines restrictions, as such these restrictions are not enforced by the fUML kernel.
6.5.2.1 Proposals shall define a SMIF Metamodel as a MOF or SMOF model of the abstract syntax of a modeling notation sufficient for completely defining any conceptual data model (CDM), logical information models (LIM) or model bridging relation (MBR).	A MOF meta model of SMIF is included. It is directly derived from the SMIF conceptual model by removing SMIF extensions not valid in MOF. MOF does not comprehend SMIF subsets and redefines restrictions. As such, these restrictions are removed in the MOF model and must be enforced by other means. They remain restrictions on the model structure.
6.5.2.2 Proposals shall define at least one graphical concrete and at least one textual concrete syntax for the SMIF Metamodel. The graphical notations shall be specified using the OMG diagram definition standard based on the abstract syntax.	The SMIF graphical notation included leverages UML and the SMIF profile for UML. While various "fact modeling" textual notations have been evaluated in creating SMIF, no text notation is included at this time. It is anticipated that other notations will be defined for the SMIF model in later efforts.
6.5.2.3 To the greatest extent practical, the SMIF Metamodel and notations shall be based on reuse or adaptation of existing modeling and logic languages. Proposals shall provide justification when this is not considered to be the best solution.	The SMIF graphical notation utilizes UML . In keeping with the philosophy of SMIF, the relationship to other models is expressed as mappings.
6.5.2.4 The content of models expressed using the SMIF Metamodel shall be Web addressable resources, each having a unique Web identity in support of Linked Open Data.	As a MOF meta model SMIF models are web addressable. The OWL/RDF mapping of SMIF also produces web addressable model content.
6.5.2.5 Proposals shall provide an MBR model bridging from the SMIF Conceptual Model to the SMIF Metamodel, specifying how CDMs, LIMs and MBRs based on concepts defined in the SMIF Conceptual Model may be represented using the SMIF Metamodel and so expressed in SMIF notations. Conversely, all statements made as part of any model represented using the SMIF Metamodel shall have a precise and well-defined semantic mapping to the SMIF Conceptual Model.	The SMIF meta model is a minor transformation from the SMIF conceptual model using the same semantics, terms, constructs and model identity. For this reason, no mapping was deemed necessary.

Proposals shall define normative MBR models, in the SMIF Language, that bridge the SMIF Conceptual Model to metamodels for the following existing languages, in order support the federation of information defined in these languages. a. Entity-relationship (ER) modeling, with a metamodel such as that proposed for IMM b. SQL Data Definition Language (DDL), with a metamodel such as that proposed for IMM c. XML schema definitions (XSDs), with a metamodel such as that proposed for IMM d. Unified Modeling Language (UML) e. Semantics of Business Vocabularies and Rules (SBVR) f. OWL web ontology language, with the metamodel as given in ODM g. RDF Schema (RDF/S), with the metamodel as given in ODM	Mappings are specified for UML and OWL. Additional mappings may be included as user demand indicates. Experience indicates that mappings should be independent of the foundation specification such that they can be developed and maintained independently. This helps to avoid monolithic specifications.
6.5.3.2 Proposals shall provide a minimum of four non-normative examples drawn from different domains, demonstrating the overall applicability of the proposed SMIF Language to the definition, extension, validation, federation and integration of information models and their physical schema representations.	Extensive examples are provided in OMG submissions based on SMIF. These include "threat/risk" (OMG document SYSA/2016-0-02) and draft versions of FIBO. Other multiple other small examples are included in this document. Numerous examples are provided in this specification.

0.3.2 Non-mandatory features

6.6.1 Proposals may provide a direct mapping from the SMIF Metamodel to RDF, RDF/S and/or OWL, as an exchange format beyond that provided by XMI based on the SMIF Metamodel abstract syntax.	A mapping to OWL-2 is included.
6.6.2 UML Profile for SMIF 6.6.2.1 Proposals may define a profile of UML that represents all or part of SMIF using UML stereotypes, tagged values and OCL constraints. 6.6.2.2 If a UML Profile is included, an MBR shall be defined between the profile and the SMIF Metamodel. 6.6.2.3 If a UML Profile is included, proposals shall describe the fidelity of the profile and any information loss between the profile and corresponding models expressed in SMIF notation.	A UML profile for SMIF (at all levels) is included and defines the graphical syntax for SMIF. Other notations may be added in the future.

0.4 Resolution of Discussion Issues

References to and naming of individuals.	All SMIF entities may have multiple names and identifiers. This includes individuals as well as types and metadata. Each term is a first-class entity that may be defined in a context independent of the original definition. Context may also be used to cope human languages.

SMIF

1 Scope

1.1 Business Need

Our ability to share, manage, analyze, communicate and act upon information is at the foundation of the modern enterprise and open, collaborative government. Information sharing is essential for an integrated approach to enterprise supply chains, fighting terrorism, business and government intelligence, inter-organizational collaboration and integrating enterprise applications. Yet, this essential capability has remained difficult and expensive to achieve in information systems which are frequently isolated, stove piped, and difficult to integrate. The inability of our systems to share information hampers the ability of our organizations to collaborate and for our processes, services, and information resources to work together. Much of our information technology budgets are consumed by attempts to overcome this "semantic friction" in our systems and organizations are currently spending more on application integration than on building new applications [Gartner2011]. The overall human and financial cost to society from our failure to share and reuse information is many times the cost of the systems' operation and maintenance.

In general, information sharing can be understood at a number of different levels.

- *Infrastructure* is the technology used to maintain data and move it from one place to another.
- *Format* is the way data are structured, its syntax.
- *Semantics* deals with how data is interpreted as meaningful information. For an information system, this interpretation is reflected in how the data is processed in order to carry out the business purpose of the system.

We are effective at dealing with data infrastructures today, and we are somewhat effective at handling multiple data formats, albeit via manual and point-to-point integrations. However, we are not very good at understanding how the semantics of data in independent data sources are related. Too often, how each system interprets shared data is implicit in the specific design and operation of the systems. Differences in structure, terminology, viewpoint, and notations make system-specific data structures hard to integrate, negatively impacting the capability to federate these systems or the information they contain.

Full semantic integration requires information systems to all properly and consistently interpret the data exchanged among the systems. This, in turn, requires that there be an explicit understanding of what the desired semantic interpretation *is* at a business level. A semantic *model* can be used to express this understanding in a way that can be validated by the business stakeholders of the systems being integrated. And, given a computational underpinning for such a model, it can then also be used for supporting analyses and deductions necessary to carry out the necessary integration.

Unfortunately, for most existing information systems, the desired semantics have not been effectively modeled. The following are some scenarios in which semantic integration is, nevertheless, critical. Diverse and disparate efforts are currently being made to address these scenarios, examples of which are included with the scenario descriptions below. But, as of today, there is no consistent way to address modeling for semantic integration in general across all these areas.

- *Data integration between business systems.* Many large businesses have a critical need to better integrate systems in support of complex products. Not only may their business area have suffered financial distress, but there may be a need for new government reporting or new analytics and integration due to acquisitions. Such organizations typically have multiple layers of existing data bases, middleware specifications and XML schemas for use in web services, event brokers, etc. Most, if not all, of the existing systems and technologies still need to be supported. There may be dozens or even hundreds of enterprise systems involved and hundreds or thousands of small applications and spreadsheets.

  Example. A common approach chosen for integrating major business systems is to create a "canonical model" of the data within a domain and then map data into and out of that model using data mapping tools. Unfortunately, while there are various proprietary tools to support such an effort, there is no widely available standard-based tooling for the job. For instance, while UML can be and is used for the modeling part of the job, a general modeling notation such as UML is not for the conceptual level of modeling required, and there is currently no standard profile to adapt it to the task nor for mapping data into and out of a canonical model in general. (The Model Driven Message Interoperability specification provides some support for the latter, but only limited to message format transformation for the financial services domain.)

- *Data federation across multi-disciplinary teams.* Developing complex systems often involves many parties who are widely distributed in location and time. Such development therefore requires efficient and effective information exchange during the complete development and operations lifecycle of the system. This can only be achieved by realizing semantic integration between all involved parties.

  Example. The European Cooperation for Space Standardization (ECSS)[1] addresses this issue by introducing the concept of *a global conceptual model.* This model is used in the implementation of "space system data repositories" as federations of physical databases. These databases are geographically dispersed and change over time but are logically integrated in an interoperable architecture, so that data can be exchanged effectively and reliably. Such data repositories need to be stable over a long period of time, so modeling must be at the semantic level *independent* of technology and tools. This modeling allows for upgrading the implementation technology without changing the model and data itself. The primary aim of this is to substantially reduce the system development and operation costs while achieving greater precision and federation.

- *Information federation across an industry.* Major industries, such as finance and telecommunications, need to deal with the representation of information relative to multiple contexts, taking into account different business processes, specific modeling goals and needs, visualization and implementation requirements or the existence of overlapping modeling domains. These differing contexts and conditions may require emphasizing different aspects and characteristics of essentially the same information. The representation of a concept in one view may be different from the representation of the same concept in another view as the context-specific details that are relevant differ from view to view. Information can be described using different yet compatible paradigms (e.g., domain-specific languages vs. UML and profiles) yet the meaning and semantics of the information should stay the same regardless of the format and viewpoint. This, again, highlights the need to focus on a common core model of shared semantic concepts.

  Examples. Some examples of efforts to deal with industry-level information federation are the Shared Information and Data (SID) Model, developed by the TM Forum [TMForum], the Common Information Model (CIM) developed by the Distributed Management Task Force [DMTF] and the Reference Information Model (RIM)developed by Health Level Seven [HL7].

- *Information sharing and federation of threat and risk information,* Threats and risks are increasingly multi-dimensional in nature – spanning physical space and cyber space. Threat actors understand and exploit our stove piped approach to sharing and analyzing information which leads to ineffective collaboration and mediation. Only by federating information across multiple domains such as cyber, physical, critical infrastructure, criminal, intelligence and defense, irrespective of technical and political boundaries, can we effectively counter multi-dimensional intentional threats, natural events and system failures.

  Examples. Attacks on our critical infrastructure have and will combine cyber attacks with physical attacks. This has been seen in exploits of our electric power grid where physical weaknesses are combined with Cyber to harm our physical infrastructure. By combining Cyber, criminal and terrorist information we will be better able to deal with these critical threats.

[1] ECSS is an initiative established to develop a coherent, single set of user-friendly standards for use in all European space activities [ECSS].

- *Data federation across government organizations.* Information sharing has been recognized by governments as a key enabler for purposes as diverse as fighting terrorism to financial transactions. There has been some progress in standardizing exchange schemas, which is a big step ahead of no standards at all, but the need exists to ensure that there is no ambiguity in the semantics of the exchanged data in order to safely enable the reuse of that data. In addition, any such standard must accept that there are and will be other such standards and that these also need to be federated.

  Example. The U.S. Information Sharing Environment (ISE) "provides analysts, operators, and investigators with integrated and synthesized terrorism, weapons of mass destruction, and homeland security information needed to enhance national security and help keep our people safe" [ISE]. ISE depends on fixed schemas for information sharing, i.e., the National Information Exchange Model (NIEM) and the Universal Core (UCORE). These schemas provide XML Schema definitions that are claimed to be sufficiently common and universally understood by relevant stakeholders regardless of the IT systems being used within their intended domains. Even within NIEM, though, hundreds of overlapping schemas have been defined.

- *Model federation across different modeling metamodels.* The OMG itself has multiple standards related to modeling. These standards were originally created independently, resulting in difficulties when users try to use them together to share information embodied in models using the different standards. A conceptual domain model, distilled from the existing OMG modeling standards, would facilitate their comparison, acknowledging the commonality (or lack thereof) between the different concepts and definitions and bridging those concepts.

  Example. OMG specifications related to just process modeling include BPMN, UML Activities, BPDM, and SPEM. A case in point in the difficulty this has caused relates to the *UML Profile for DODAF and MoDAF* (UPDM), a wide ranging profile supporting US Department of Defense (DOD) and UK Ministry of Defence (MOD) architecture frameworks. The UPDM community wishes, for example, to be able to use BPMN process models in the context of their UML Profile. A stopgap tactic has been to define an additional *UML Profile for BPMN*, which allows BPMN-looking diagrams to be drawn in UML, but it is clear this is not a strategic approach. A better approach would be to create a "process modeling" conceptual domain model that would then permit model bridging relations between BPMN, UML, BPDM and SPEM models, allowing sharing across users' process models

- *Schema Evolution*. As information systems evolve to support changing enterprise needs, the datasets they use need to evolve as well. While some changes are additive and readily accommodated, others involve factoring and evolving concepts. At their core, such changes require the evolution of the dataset schema underlying the system and the migration of the data from the old to new schemas. Such changes also impact the logic that interacts with the dataset and every external interface and related data structure. While there is some tooling available for schema migration, there is little available to aid in the evolution of the logic and external interfaces. The absence of semantic understanding of the relationship between the schema and external interface data structures makes tooling to aid in the evolution problematic.

  Example. It is common for an enterprise to represent the concept of *customer* as a composite of information about the person and the role that person plays with respect to the enterprise. Evolving needs, including regulatory requirements, require many enterprises to now factor this concept so that they can represent that the same person may play other roles as well, such as employee. Such semantic understanding is required to enforce constraints such as a prohibition against the same individual playing both the customer and employee role in a transaction. The absence of semantics-based tooling makes such changes labor intensive and error prone.

Current standards for information and data modeling may be effective at defining a particular data model for a particular application using a particular technology to solve a particular problem. But, as highlighted by the above examples, the methodology for using these standards at a higher level of abstraction – namely for cross-domain and crossorganizational semantic modeling – is not as well or as widely understood. As a consequence, the models available within a given organizational context are often not well suited to use across multiple dimensions or technologies, and so poorly support the needs for sharing and federation.

1.1 Scope

1.1.1 Semantic federation and integration

A semantic information federation approach is the one leveraged in this specification. A semantic approach focuses on concepts and their meaning, not how they are represented in any particular schema, syntax, vocabulary, or technology. Mappings then define how various data formats and vocabularies *represent* those concepts. Concepts are well defined in a conceptual reference model – a more precise way to define a vocabulary or taxonomy. Conceptual reference models may be called "ontologies", or "abstract data models" but some ontologies or abstract data models are essentially programs and not conceptual.

The essential difference between a conceptual reference model and a concrete application model is that it describes real world things and their relationships as understood by stakeholders. It is a *model of the world*[2], not a model of data or a system. When we have a concept like "Incident" in our model, "instances" of incidents are real things that happen – not a Java object or stream of XML. However, we may also have concepts of actual things, such as a specific incident.

- A conceptual reference model is conceptual in that it is an expression and formalization of how a community conceives of their domain, problem area, business or environment. It is not a model of the solution or a technology. This could also be called a "conceptual domain reference model".
- A conceptual reference model is a reference model in that it is intended to supply reference concepts for what information in various systems means, to "connect the dots" between application or data models. It is not intended as a concrete application or solution model in and of itself.

These real-world reference concepts are the pivot points between different ways to name, describe or talk about the things we deal with every day. This "world of things" is what we understand – of course there can be many names for and descriptions of the same thing.

How do we know it's the same thing? In some cases we can describe something so precisely and mathematically that we can be sure, in many cases it is just a shared concept based on a definition and how that concept relates to other concepts. We allow for both precise and pragmatic definition of things.

In real-world scenarios such as finance and risk risk management we also have to be fully aware of how much we trust various information sources. It is common, if not the general rule, that different information sources will have conflicting information about the same things. How do we know what to trust? This specification provides the basis for trust, in capturing the provenance of information, but it leaves the evaluation of trust to the capabilities that utilize or analyze the information – or to the stakeholders who must make decisions based on it. This is a common pattern in this approach, providing the basis for decisions but not the specifics for how to make those choices.

A conceptual reference model has some similarities to a canonical data format in that it attempts to capture cross-stakeholder information needs – but it abstracts above the data format, technology, terminology and even the specific use case and structure for that information.

[2] Or more generally, real or possible worlds.

1.1.2 Expressing conceptual reference models

As stated in 7.4.3, conceptual reference models are models of the world – or at least how communities conceive of the world. This is differentiated from models of data (e.g., an E/R model or XML Schema) or models of software (e.g., a Java program). In their pure definition, *Ontologies*[3] are conceptual models, however not all ontologies or ontologies are conceptual and many are intended for building semantic applications using specific "reasoning engines", not as reference models. Ontology languages are typically optimized and restricted for their intended class of reasoning engine, not to capture domain concepts in general.

Of course, human natural languages are the most common way to express concepts. Natural language is used in the definition of our concepts but those definitions are augmented with more formal assertions.

There can be confusion between the language used to express a model and what it models. For example, while Entity-Relational (E/R) was designed for SQL data models it can be used conceptually. At the other end of the spectrum many ontology languages have been used to express data models or to support specific forms of inference based computation. *The language does not make a model conceptual* (or an ontology or a data model), *what is being modeled does*. Of course some languages are better than others for conceptual reference modeling and mapping than others, [SMIF] is used because it is designed for expressing conceptual reference models and mappings to various forms of data.

Our goal in this specification is to utilize a set of conceptual reference models as the pivot point between different data models and syntaxes for expressing information about real-world threats and risks. While using a conceptual reference model in this way is not new, there has not been a well-accepted standard for doing so. None of the well accepted modeling languages are specifically designed for conceptual reference modeling and mapping – most are designed for software modeling (data, procedural computation, or inference).

The Unified Modeling Language (UML) was originally designed for modeling object-oriented software, but is also used for other purposes and is easily extended with *profiles*. We are using a profile of UML based on Semantic Modeling for Information Federation (SMIF). UML is a well-accepted modeling language with widely available resources – SMIF provides a standard way to use UML for the purpose of conceptual reference modeling and mapping. The combination of UML and the SMIF profile provides an expressive, and automatable way to express the conceptual reference models and mappings. Any standard conformant UML tool can import and manage the profile and the conceptual reference model but special tooling is required to automate mappings.

The intent of the conceptual reference model and mappings is that a tool or infrastructure developer can take that model and interpret it and transform it as appropriate for their own technology stack and data formats. They may then use that technology stack to implement the information sharing and federation capabilities described conceptually. However*, this specification makes no assumption about what that implementing technology stack may be or how it is implemented*. In addition, this specification makes *no assumption about a new "intermediate data format"* based on the conceptual reference model- the conceptual reference model has no *normative* data format – it maps to multiple possible data formats that already exist. Keeping the "middle" conceptual and virtual is a way to help resolve the "data format wars" that plague many attempts to federate where yet another data format may be unwelcome.

Mapped data formats must, of course, be used in any implementation – ultimately you need an explicit data (or language) syntax to communicate and process data. Each of the mapped data formats such as STIX or NIEM may be used to express threat & risk data within their domains. There is also growing interest in the "Semantic Web[4] " which uses the "Resource Description Framework Schema" (RDFS) language as well as the "Web Ontology Language"

---

[3]*Ontology*: 1 : a branch of metaphysics concerned with the nature and relations of being. 2 : a particular theory about the nature of being or the kinds of things that have existence.[www.merriam-webster.com]. However, ontologies have become associated with a particular branch of formal languages such as OWL and Common Logic that support logical inference.

[4]The term "Semantic Web" refers to W3C's vision of the Web of linked data. Semantic Web technologies enable people to create data stores on the Web, build vocabularies, and write rules for handling data. Linked data are empowered by technologies such as RDF, SPARQL, OWL, and SKOS. [http://www.w3.org/standards/semanticweb/]

(OWL) or the Simple Knowledge Organization System (SKOS) to describe the web of data on the internet. The semantic web technologies are well suited to data federation. The conceptual reference model can be mapped to semantic web technologies generated from the operational threat and risk (OTR) conceptual reference model, using the SMIF specification. Conceive

1.1.3 Pivoting through conceptual reference models

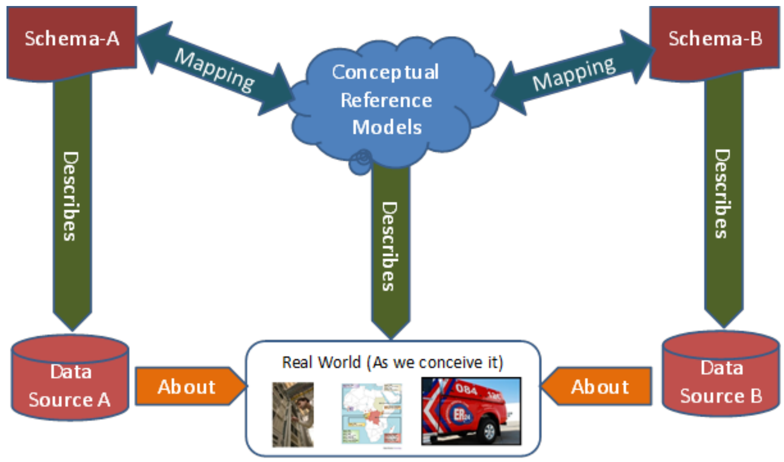


Figure 5 Illustration of pivoting through a conceptual reference model

The illistration above shows how conceptual reference models provide the “pivot point” between different schema for various data sources. The conceptual reference model describes the world (or a possible world) as we conceve it. Schema describe data, that data is about the same “real world”. Where schema elements are mapped to the same concepts their data can also be mapped or federated. Any number of schema (or other data descriptions) can pivot through the same concepts and thus provide for mappings between any combination of data sources.

1.1.4 Mapping to information and data models

Conceptual reference models are not intended to define data schema for specific applications, but to define the semantics behind those schema by mapping them to concepts. Each data schema to be mapped is imported into a model and a “**mapping model**” defines how the data structures in a concrete schema represent the common reference concepts. Only those concepts that need be shared or federated with other data schema need be mapped. An implementation of this specification is then able to map between and federate information in these different schema.

2 Conformance

The Conformance clause identifies which clauses of the specification are mandatory (or conditionally mandatory) and which are optional in order for an implementation to claim conformance to the specification.

Note: For conditionally mandatory clauses, the conditions must, of course, be specified.

There are five distinct types of conformance. These are listed below. Unless otherwise stated these types of conformance are independent.

1. *Abstract syntax conformance.* A tool demonstrating abstract syntax conformance provides a user interface and/or API that enables instances of concrete SMIF metaclasses to be created, read, updated, and deleted. The tool must also provide a way to validate the well-formedness of models that corresponds to the constraints defined in the SMIF metamodel. Abstract syntax may be further refined as either:

    a. Conceptual model conformance – corresponding to all elements not included in the packages “Rules” & “Mapping rules”.

    b. Pattern abstract syntax conformance – corresponding all conceptual model packages.

2. *UML Profile conformance.* A tool demonstrating UML Profile conformance provides a user interface and/or API that enables instances of SMIF UML notation to be created, read, updated, and deleted. Note that a conforming tool may provide the ability to create, read, update and delete additional diagrams and notational elements that are not defined in SMIF. UML Profile conformance may be further refined as either:
    a. Conceptual Modeling Profile Conformance – All elements defined for the conceptual modeling profile, clause Error: Reference source not found
    b. SMIF Rules profile conformance - All elements defined for the SMIF Rules profile, clause Error: Reference source not found.

3. *Model interchange conformance.* A tool demonstrating model interchange conformance can import and export conformant XMI for all valid SMIF models, including models with profiles defined and/or applied. Model interchange conformance implies abstract syntax conformance. A conforming SMIF tool shall be able to load and save XMI as a SMIF MOF meta model.

4. *Semantic conformance.* A tool demonstrating semantic conformance provides a demonstrable way to interpret SMIF semantics, e.g., data transformers, code generation, model execution, or semantic model analysis.

Where the SMIF specification provides options for a conforming tool, these are explicitly stated in the specification. In a number of other cases, certain aspects of the semantics are listed as "undefined" or “intentionally not specified” or “not specified”, allowing for domain- or application-specific customizations. Only customizations that do not contradict the provisions of this specification will be deemed to conform to it. However, models whose meaning is based on such customizations can only be interchanged without loss with tools that support the same or compatible customizations.

This specification comprises this document together with XMI serialization contained in machine-consumable files as listed on the cover page. If there are any conflicts between this document and the machine-consumable files, the machine-consumable files take precedence.

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these do not apply.

List of normative references. (specific reference to be included)

[UML]	OMG Unified Modeling Language (UML) v2.5 http://www.omg.org/spec/UML/2.5/
[MOF]	OMG Specification ptc/2013-08-20, Meta Object Facility (MOF) Core, v2.5
[ODM]	ODM Ontology Definition Metamodel, 2 September 2014. http://www.omg.org/spec/ODM/1.1/
[OWL-2]	W3C/TR REC-owl2-syntax:2009 OWL 2 Web Ontology Language: Structural Specication and Functional-Style Syntax. W3C Recommendation, 27 October 2009. http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/
[NIEM]	http://reference.niem.gov/
[WGS-84]	http://earth-info.nga.mil/GandG/wgs84/
[JCGM 200:2008]	http://www.iso.org/sites/JCGM/VIM/JCGM_200e_FILES/MAIN_JCGM_200e/01_e.html
[NIST-SI]	http://physics.nist.gov/cuu/pdf/sp811.pdf
[NIST-800]	http://csrc.nist.gov/publications/PubsSPs.html
[BFO]	http://ifomis.uni-saarland.de/bfo/
[MathWorld]	From *MathWorld*--A Wolfram Web Resource. http://mathworld.wolfram.com
[CL]	ISO Common Logic, ISO/IEC 24707:2007(E)
[ISO1087]	Terminology work — Vocabulary, 2000, ISO 1087-1
[SOWA1999]	John F. Sowa, *Knowledge Representation: Logical, Philosophical, and Computational Foundations*, 1999, ISBN 0-534-94965-7

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

4 Terms and Definitions

For the purposes of this specification, the following terms and definitions apply.

Terms defined in other sections

- See section Error: Reference source not found for definitions of:
  - *Conceptual Domain Model (CDM).*
  - *Logical Information Model (LIM).*
  - *Physical Data Schema (PDS).*
  - *Model Bridging Relation (MBR).*
- All terms defined in the model, clause Error: Reference source not found, are defined in SMIF.

Other Terms

- **Instance**: An "instance" designates something categorized by a type (including meta types). For example; "Fido is an instance of Dog" means that Fido <has type> Dog and that Dog <categorizes> Fido. Note: Instance does not imply (or prevent) any implementation or other restrictions such as "Factories" or "Single Classification" as do some programming languages.
- **Fact**: Facts are something that someone or something asserts to be true. The class of things that can be asserted are called "propositions" as they can be true or false. Once asserted to be true, these propositions are facts. Of course the relevance, trust or belief in facts is open to interpretation.

- **Concept**: Everything we describe in a SMIF model is considered a *concept*. A concept is anything conceived. For something to be in a model there must be a conception of it. Concepts are inclusive of types, categories, values and individuals.
- **Identity**: That which makes something differentiated from something else. Identity is an abstraction that should not be confused with identifiers, which are symbols, adopted by convention, used to identify a particular thing having identity.

1.5

5 SMIF Model 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.

5.1 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 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 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 parties must, ultimately, share 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 or communication.

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.

5.1.1 Thing

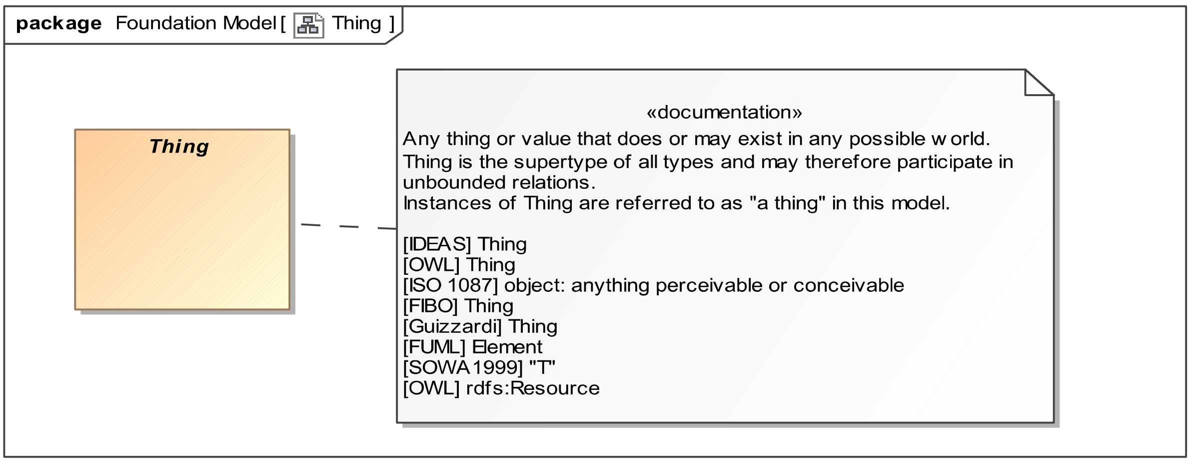


Figure 5.1: Thing

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 world” 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)

5.1.2 Type

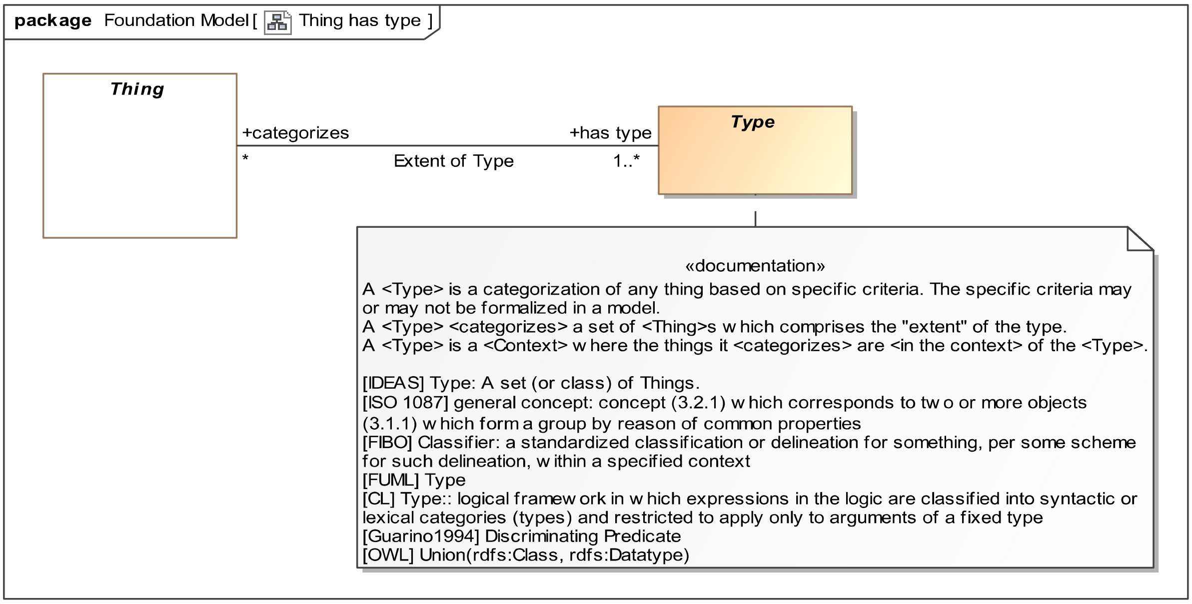


Figure 5.2: Thing has type

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 "intension" (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 intension based on our observation of and understanding of the world we live in.

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

Example 1

Figure 5.3: Fido is a dog example

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

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

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

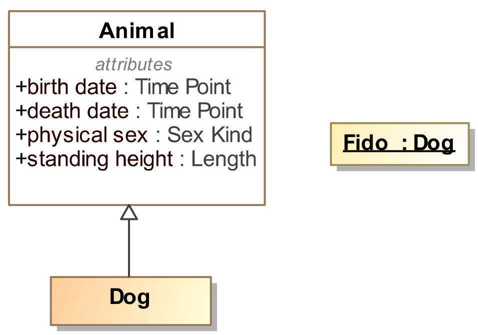


Figure 5.4: UML Type and Instance Example

We should understand that whenever we see an instance specification with a name followed by a colon (“:”) and a type, it means that there is an Extent of Type relationship instance between the instance (i.e., “Fido”) and its type (i.e., “Dog”). There could be multiple types listed, separated by commas (e.g., “Fido : Dog, Pet”), which means that there is an Extent of Type relationship instance between the instance (i.e., “Fido”) and each of its types (i.e., “Dog” and “Pet”).

Semantics

For all things X, where X <has type> T, X shall conform to 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, 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.

5.1.3 Identifiable Entities and Values

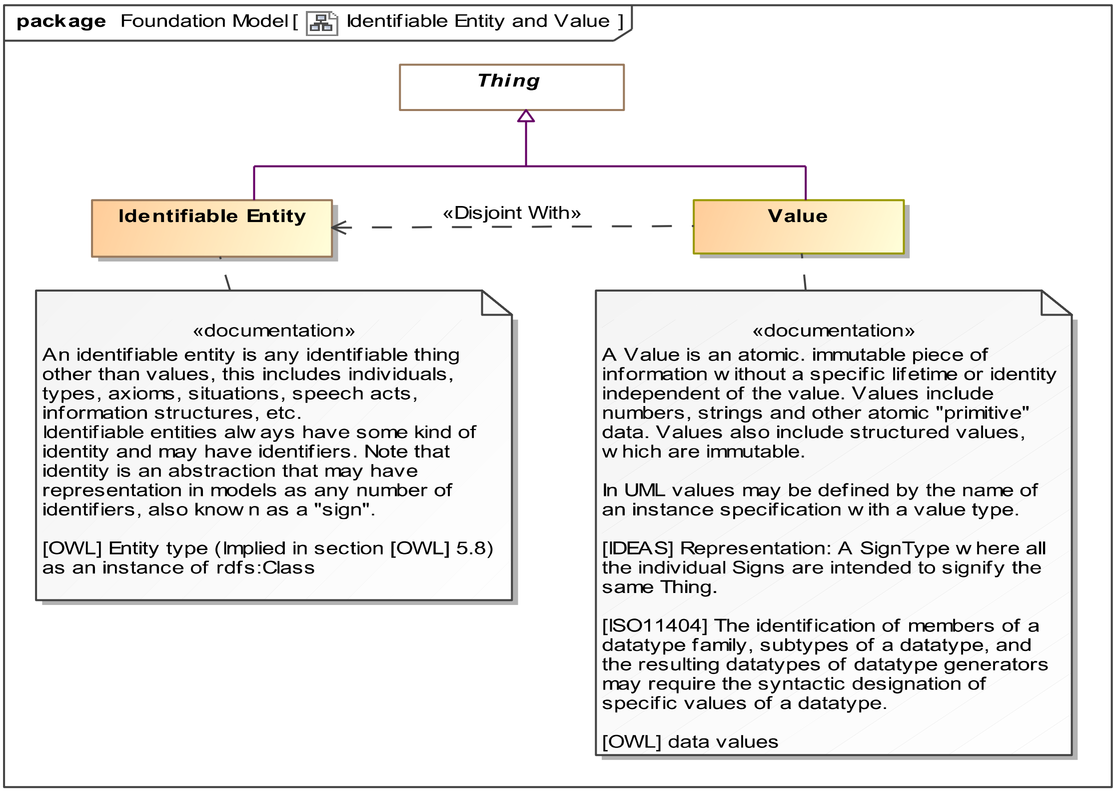


Figure 5.5: Identifiable Entities and Values

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 itself – which can never change and is 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 different 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

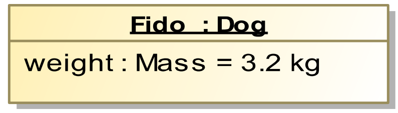


Figure 5.6: Identifiable Entity with Value Characteristic

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 a Characteristic always has a values as its 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.

5.2 Identifiers

5.2.1 Basic Identifiers

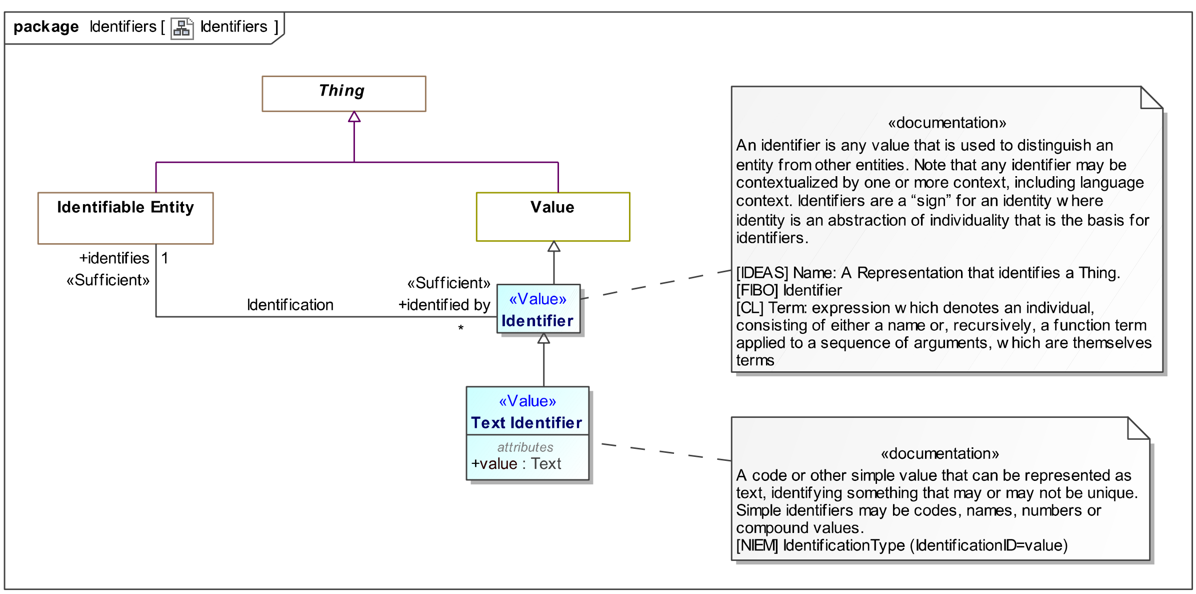


Figure 5.7: Basic Identifiers

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 (signs) that identify it*. A thing is also different from a data record that provides information about it. 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:

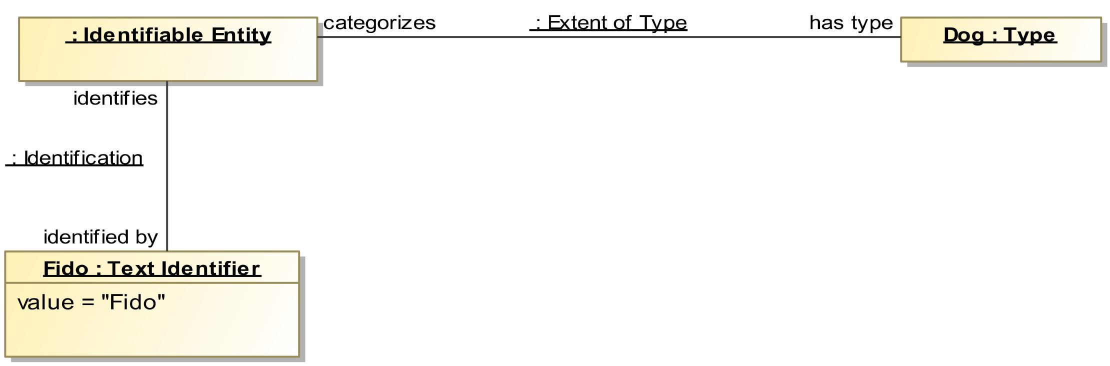


Figure 5.8: Fully expanded type and identifier instance model

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 identifier’s 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.

5.2.2 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 relate many forms of identifiers 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.

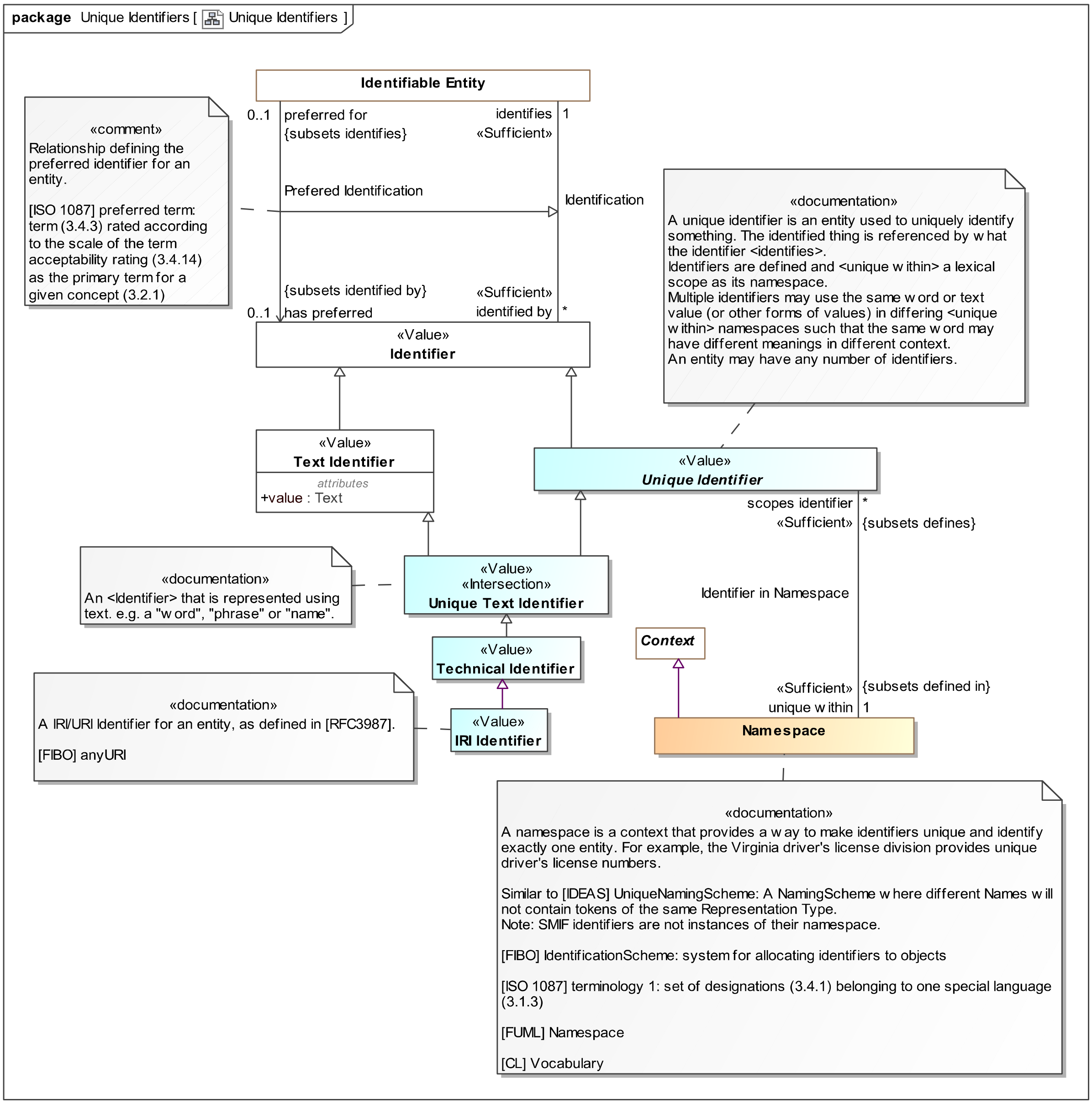


Figure 5.9: Unique Identifiers

In figure 5.9 we have shown some additional concepts to handle uniqueness and preference.

Note the “Preferred Identification” 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 “Preferred Identification” relationship specializes Identification, so that the <has preferred> identifier is the Identification that has the most intuitive meaning for the majority of people. Preference is merely 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 an identifier preferred in one context (e.g., a domain, language, or vernacular) may not be preferred in another. Later we will see how context can be used to impact what relationships are valid in any particular circumstance.

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

Names and Terms

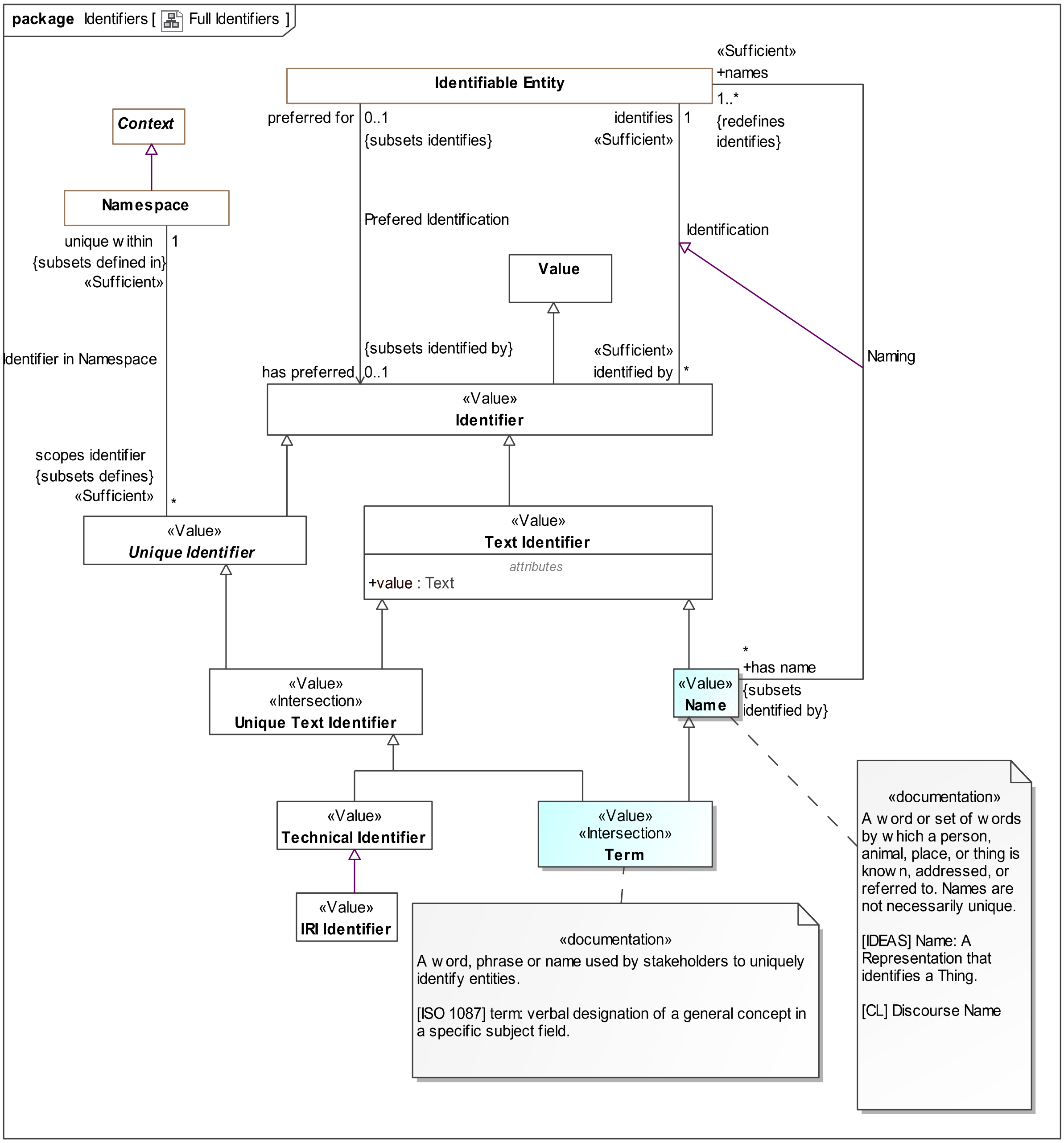


Figure 5.10: Full Identifiers Package

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.

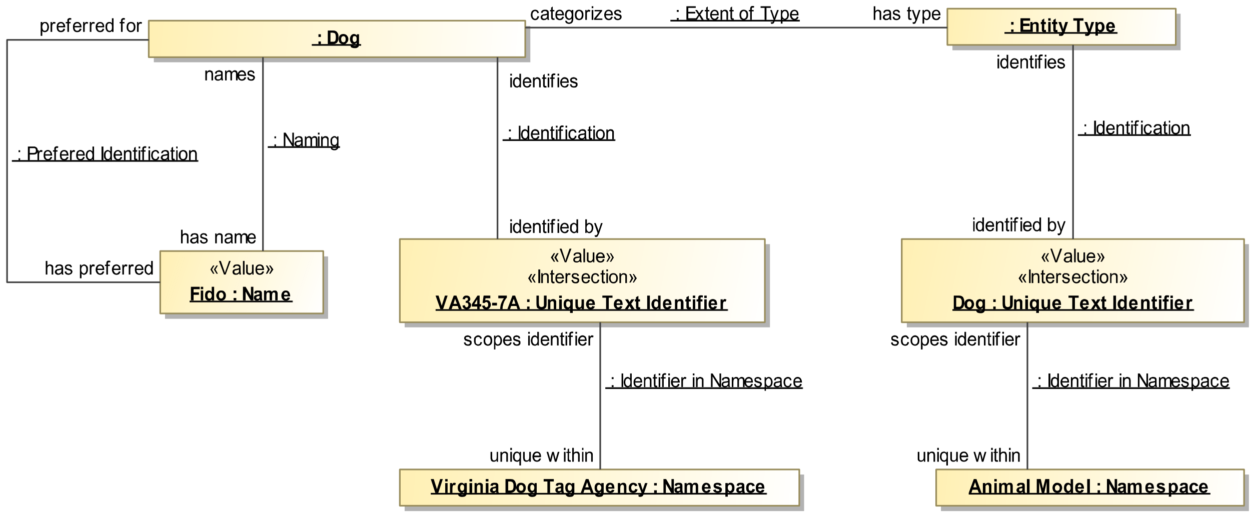


Figure 5.11: Full Identifier Example

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.

5.3 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.

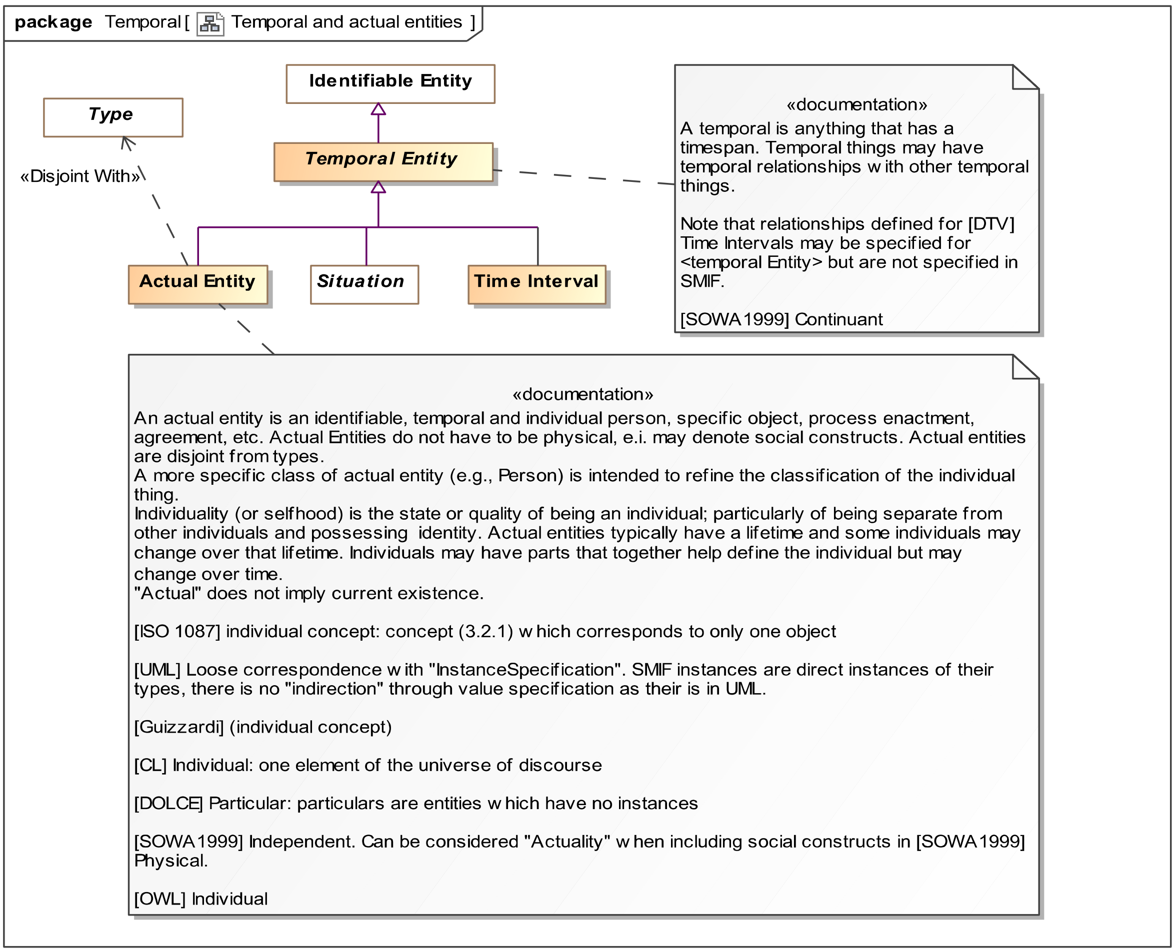


Figure 5.12: Temporal and Actual Entities

Temporal entity is primarily an abstract extension point in the SMIF model. It has no differentiating characteristics or relationships. 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, Situation, and Time Interval are temporal assists in more precisely defining their semantics.

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

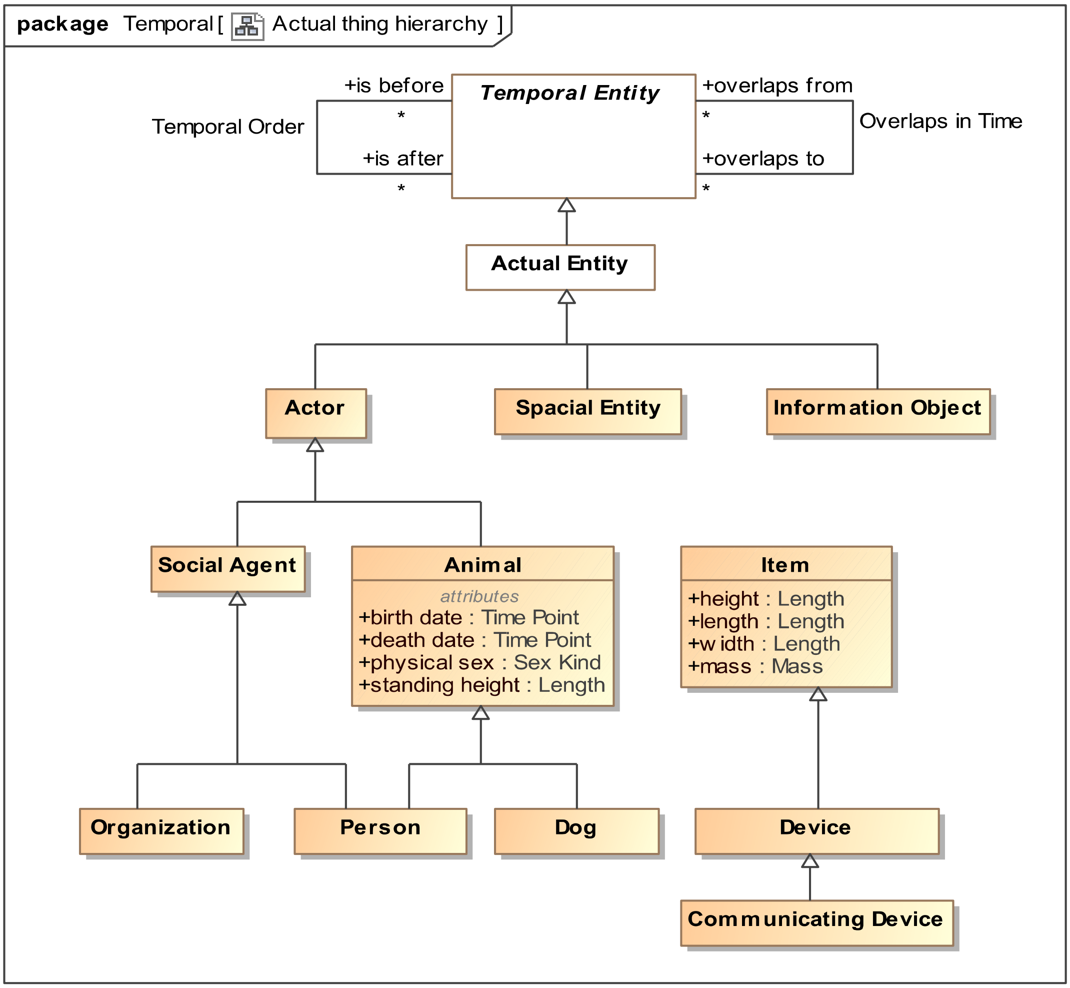


Figure ,5.13: Actual Thing Hierarchy Example

Figure 5.13 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.Here 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 Entity *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 ignored in any other model. Since a concept can be mapped to other models that may say the same or related things using different names, different structures, or more or fewer 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:

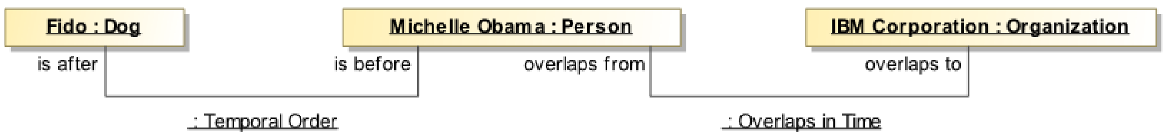


Figure 5.14: Temporal Instance Example

This example shows three instances of "actual entities": Fido, Michelle Obama and IBM Corporation. It further shows 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.

5.4 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.

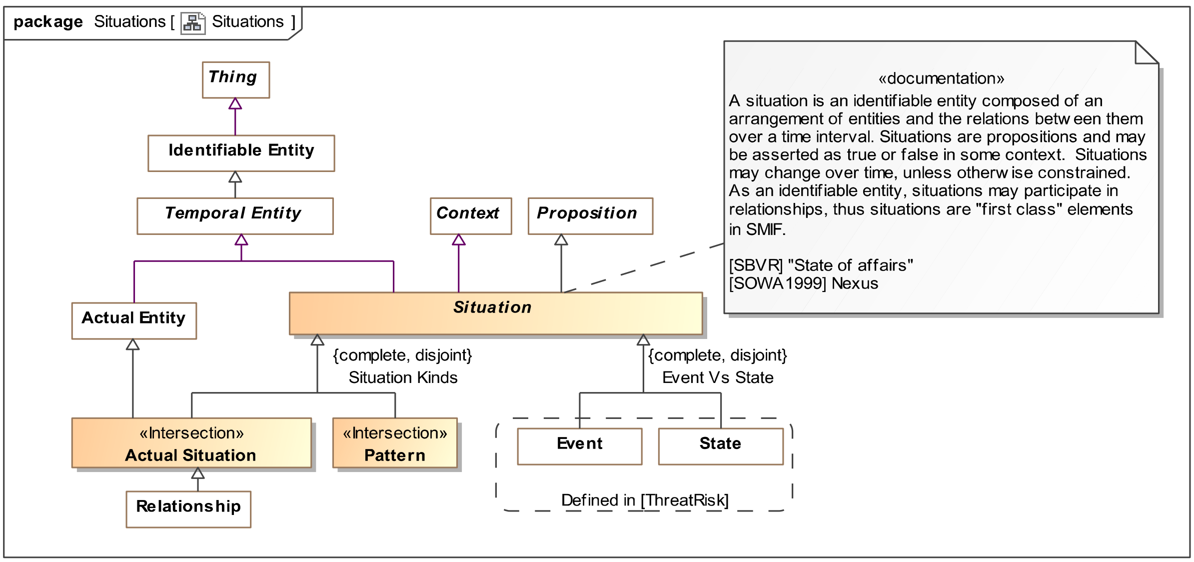


Figure 5.15: Situations - Top Level

What are actual situations? A particular terrorist entering a particular airport. A policeman who was at a concert. A particular rock falling, a particular cup full of water. Even relationships are actual situations – the actual situation of one thing being related to another, in some specific way, for some period of time, such as a particular cup holding water, or a particular person in a particular house. In the SMIF language, relationships and characteristic bindings are some of the primary kinds of actual 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 things happening (called events in this example) as well as static conditions (called states), such as a cup sitting on a desk.[5] As they are not needed for the definition of the SMIF language Event and State are not defined directly in the SMIF model – we merely show these 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.

[5] In [ThreatRisk], an instance of an Event or a State can also be an instance of either an Actual Situation or a Pattern. That provides the distinction of an event specification vs an event occurrence, as well as the distinction of a state specification vs a current state.

5.5 Kinds of Types (Metatypes)

5.5.1 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.

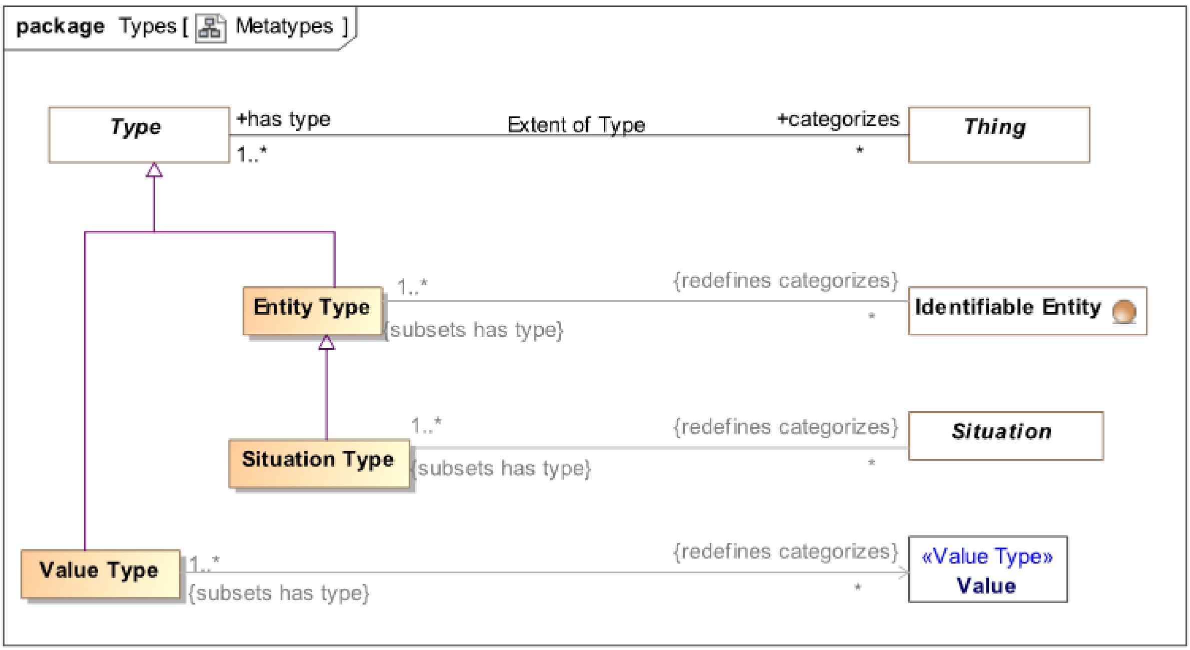


Figure 5.16: Metatypes

For something to be represented in SMIF, it must ultimately be an instance of "Thing". Every "Thing" is in turn an instance of the *metatype* called "Type", or one of its specializations, which categorizes what kinds of thing it is. Every specialization of "Type" provides increasingly-precise descriptions to which the things it categorizes must conform. We call "Type" and its the specializations *metatypes*, and we call "Thing" and its specializations *types*. We use this naming scheme because SMIF can represent multiple levels of typing. For example, "Fido" is an example of a general concept called a "Dog". This is the "natural" level of typing, and we do not usually append the word "Type" to those. "Fido" is an example of a "Dog", not of a "Dog Type". A dog type may be something like "English Sheep Dog".

Providing a metatype allows more expressability for the kinds of types in SMIF, and provides rules for defining kinds of types and rules about when and where each type can be used. The left side of Figure 5.16 shows the metatypes defined in SMIF for the types called Identifiable Entity, Situation, and Value, shown on the right side. SMIF generally uses a pattern where there is one special instance of a metatype that is the supertype of all other instances of that metatype. For example, There is one special instance of Entity Type called "Identifiable Entity". It would be the supertype of "Dog", and "Dog" would be another instance of Entity Type. For another example, the class called Value (shown on the bottom right) is an instance of Value Type (shown on the bottom left). Every specialization of Value, such as "Mass" is also an instance of the metatype Value Type.

Sometimes it is desirable to give these fundamental SMIF types distinctive styles and icons in UML. For this purpose, SMIF provides namesake stereotypes for each of these fundamental types in its UML profile. For example, a specialization of "Value" called "Mass" can have the stereotype «Value» applied to it to give that class a distinctive color and icon on UML diagrams. Applying such a namesake stereotype to a class *implies* it is both an instance of a metatype and a specialization of its namesake type, so creating the generalization arrow is unnecessary.

Note that each specialization of Type further constrains the kind of thing that the metatype can <categorize>. For example, all Value Types categorize only Values. In addition, the metatypes are a required type of the type they categorize. For example, each instance of a Value must have at least one type that is a “Value Type”, possibly among other types.

Each of the SMIF concrete 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.

5.5.2 Full Meta-Type Hierarchy

The following shows the complete hierarchy of metatypes.

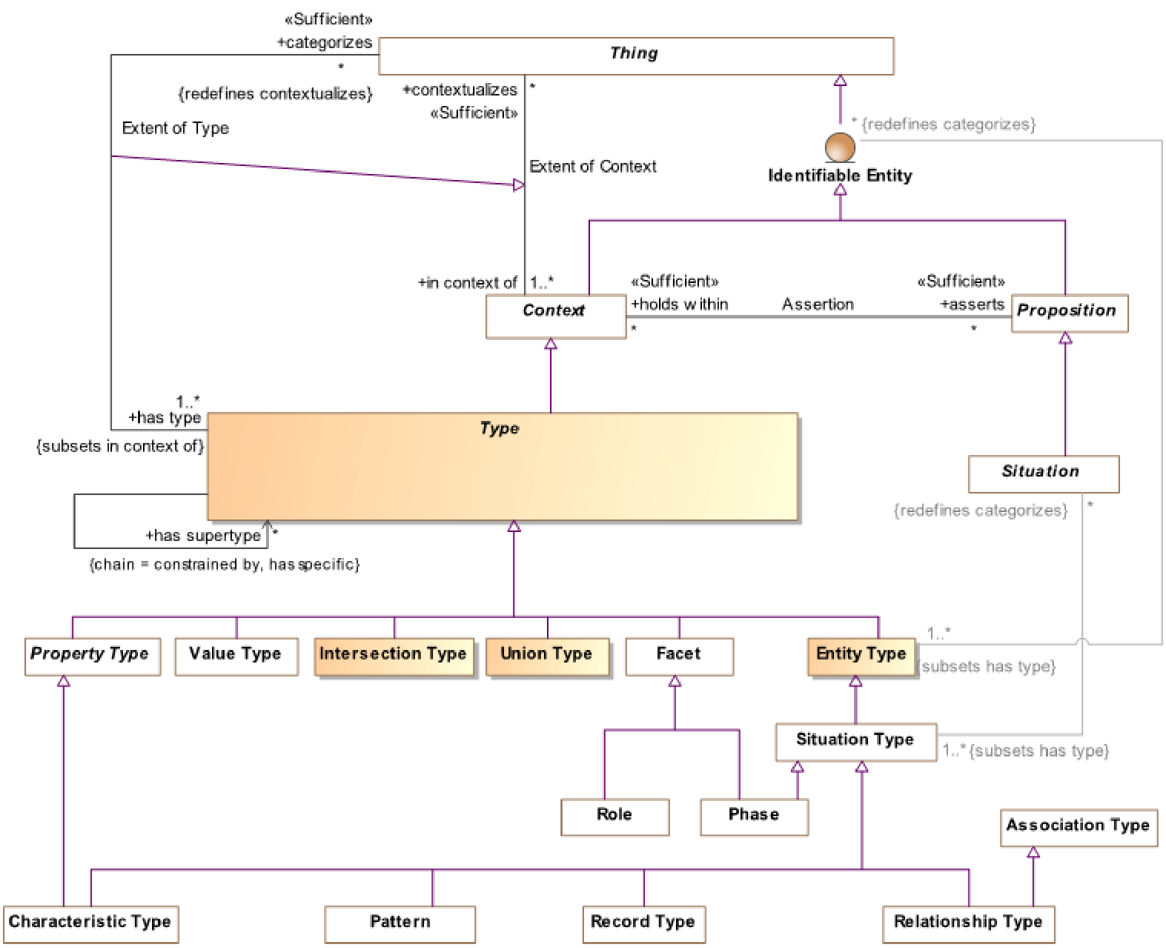


Figure 5.17: Metatypes

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

5.5.3 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

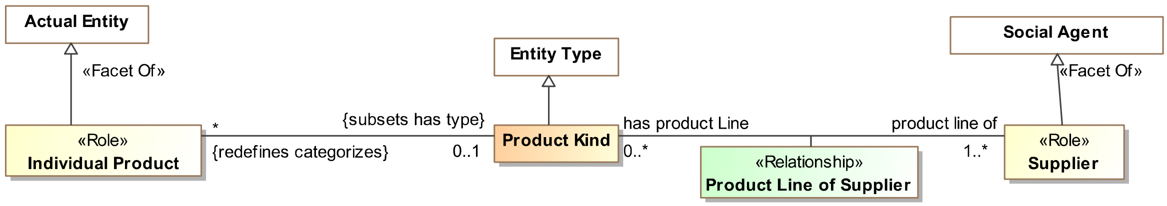


Figure 5.18: Domain Specific Metatype Example

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

Figure 5.18 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.

5.6 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.

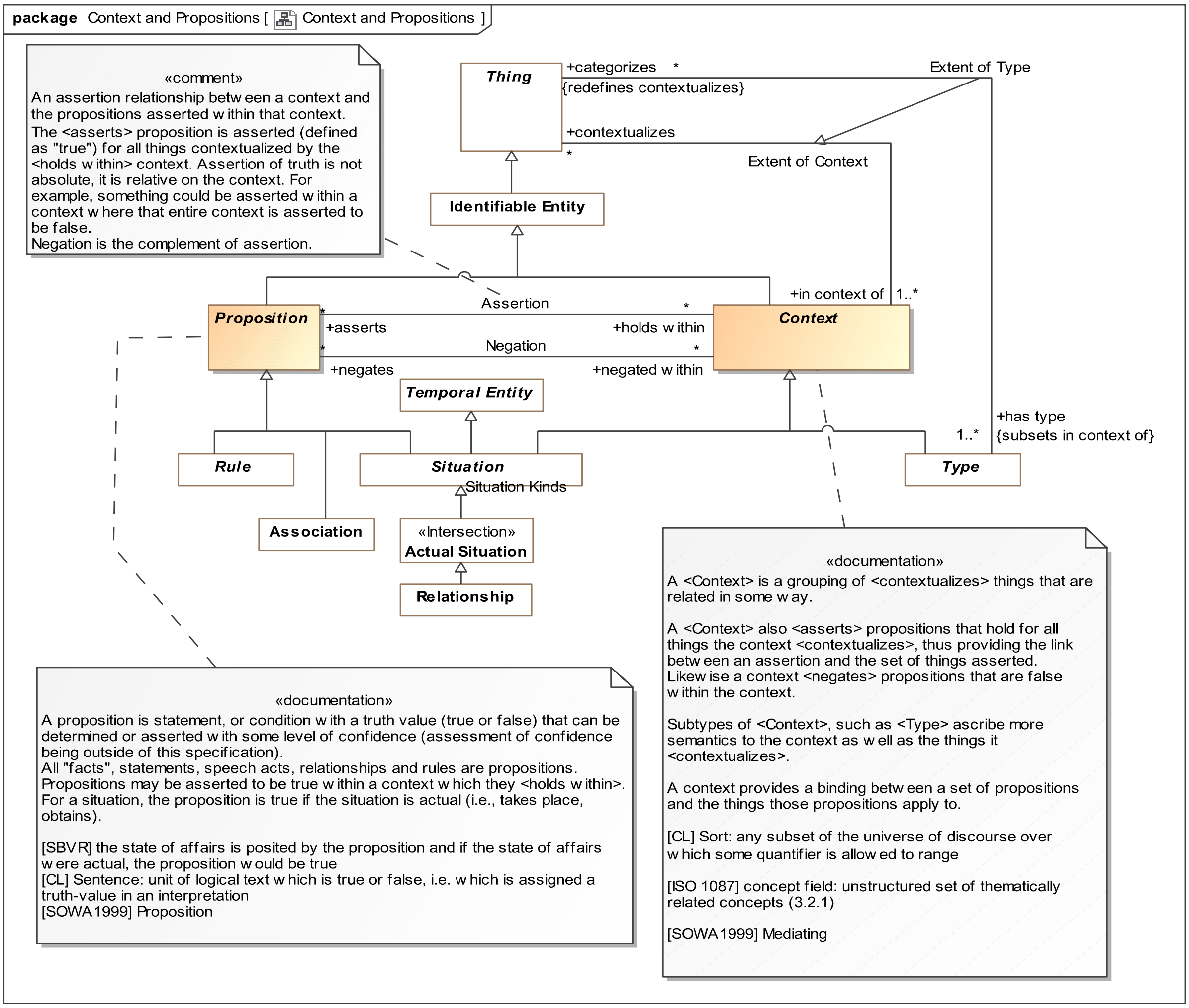


Figure 5.19: Context and Propositions

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 5.8 for examples of assertions negations in a context.

5.7 Properties, Characteristics and Relationships

5.7.1 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.

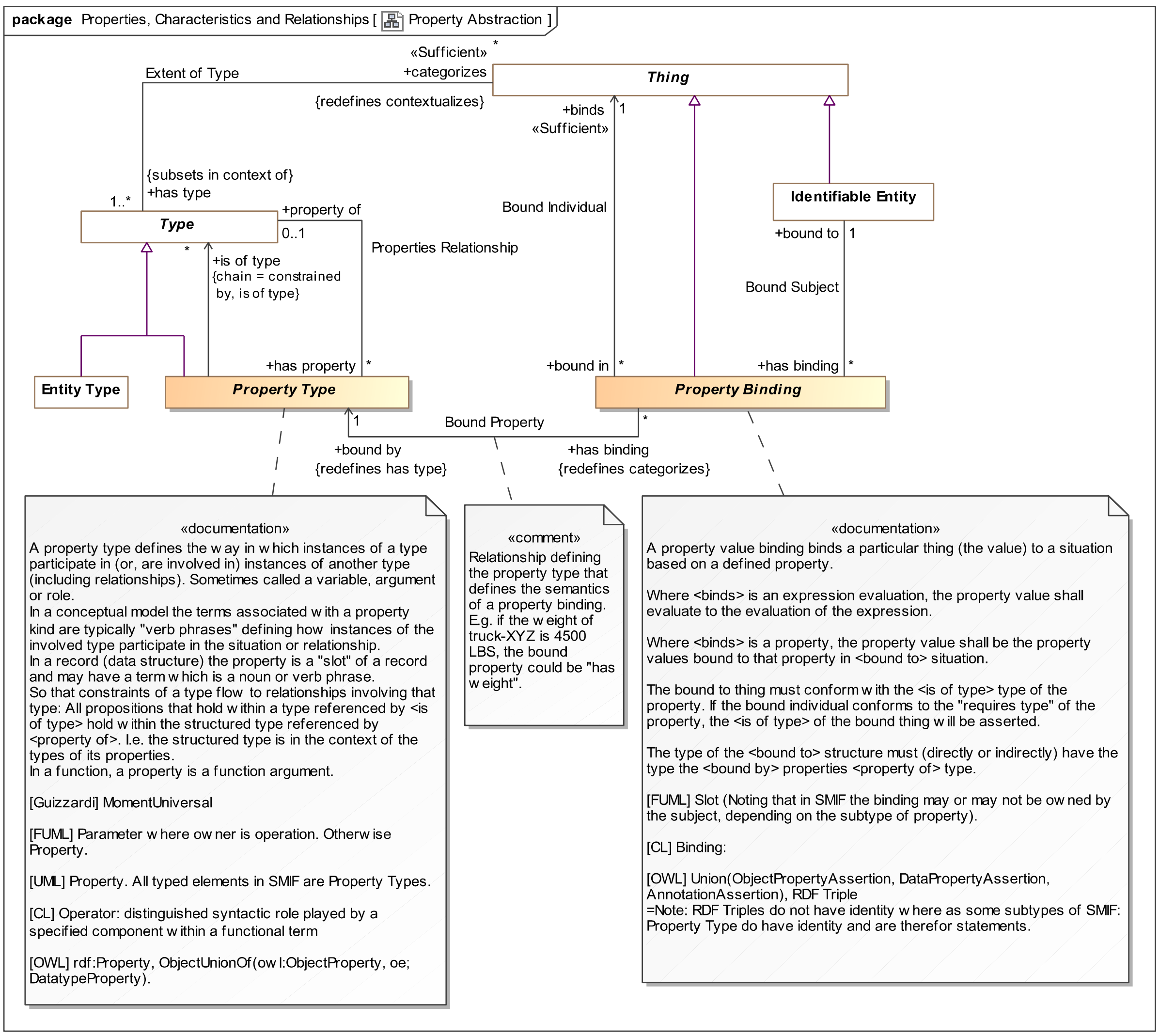


Figure 5.20: Definition of Property Type & Property Binding

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.

5.7.2 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 5.7.3. 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.

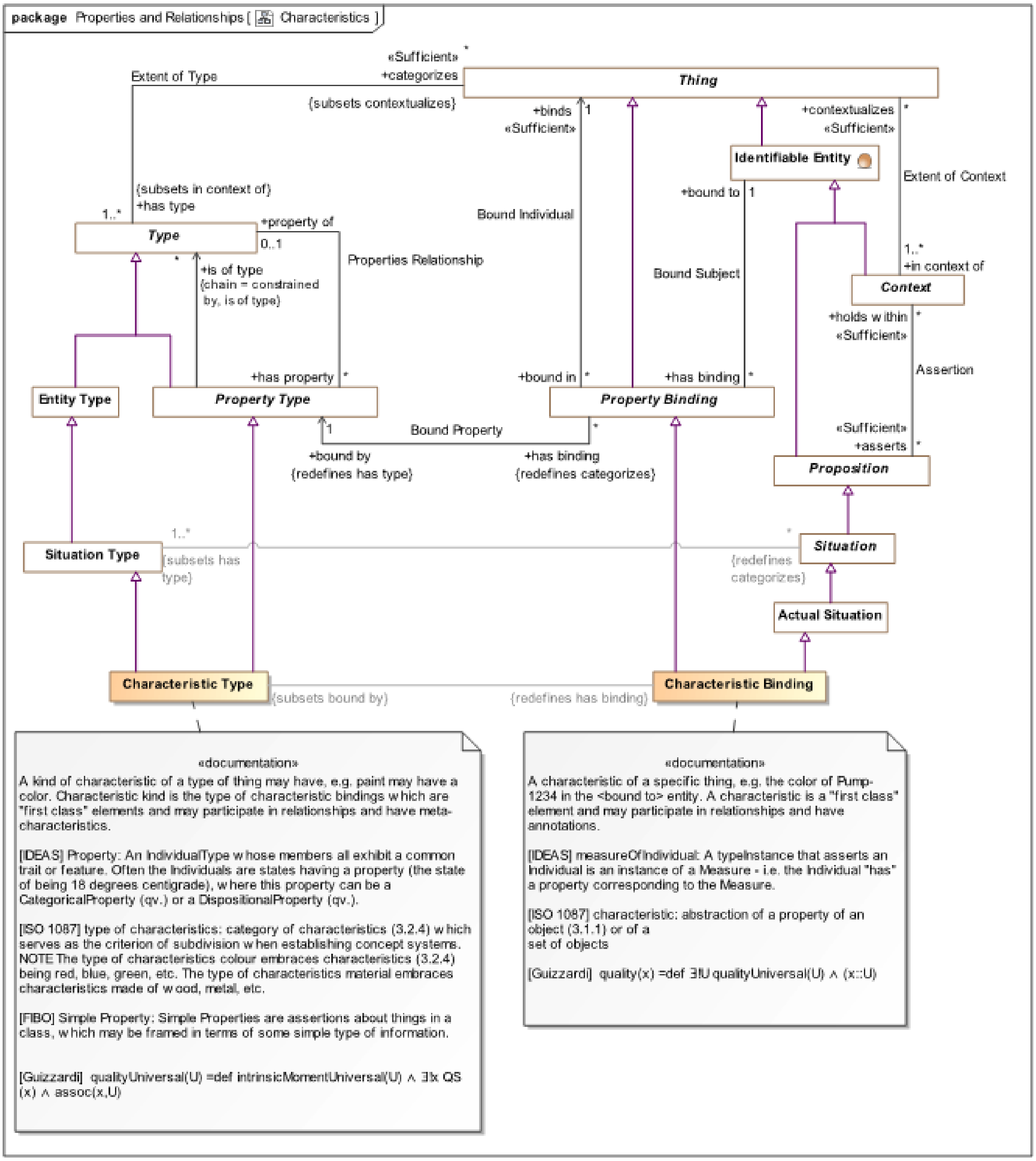


Figure 5.21: Definition of Characteristic & Characteristic Kind

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 same *characteristic type* at *different times or from different sources*. The expectation is that the type of characteristics will be a value type, but to allow for diversity in approaches, this is a recommendation, not a rule.

Examples

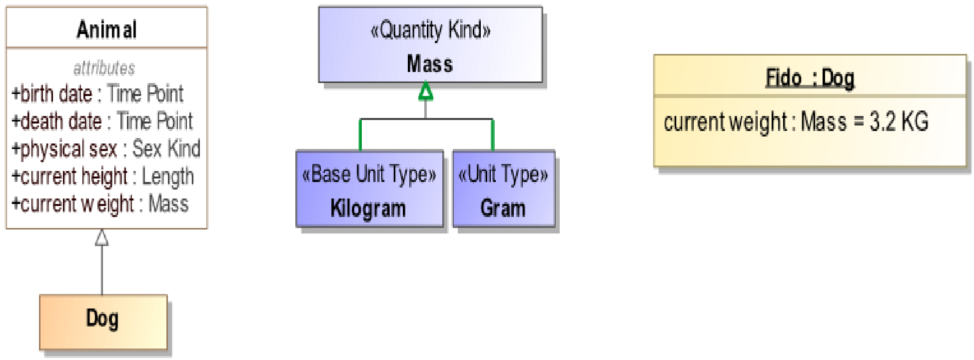


Figure 5.22: Defining and Using a Characteristic

Figure 5.22 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.

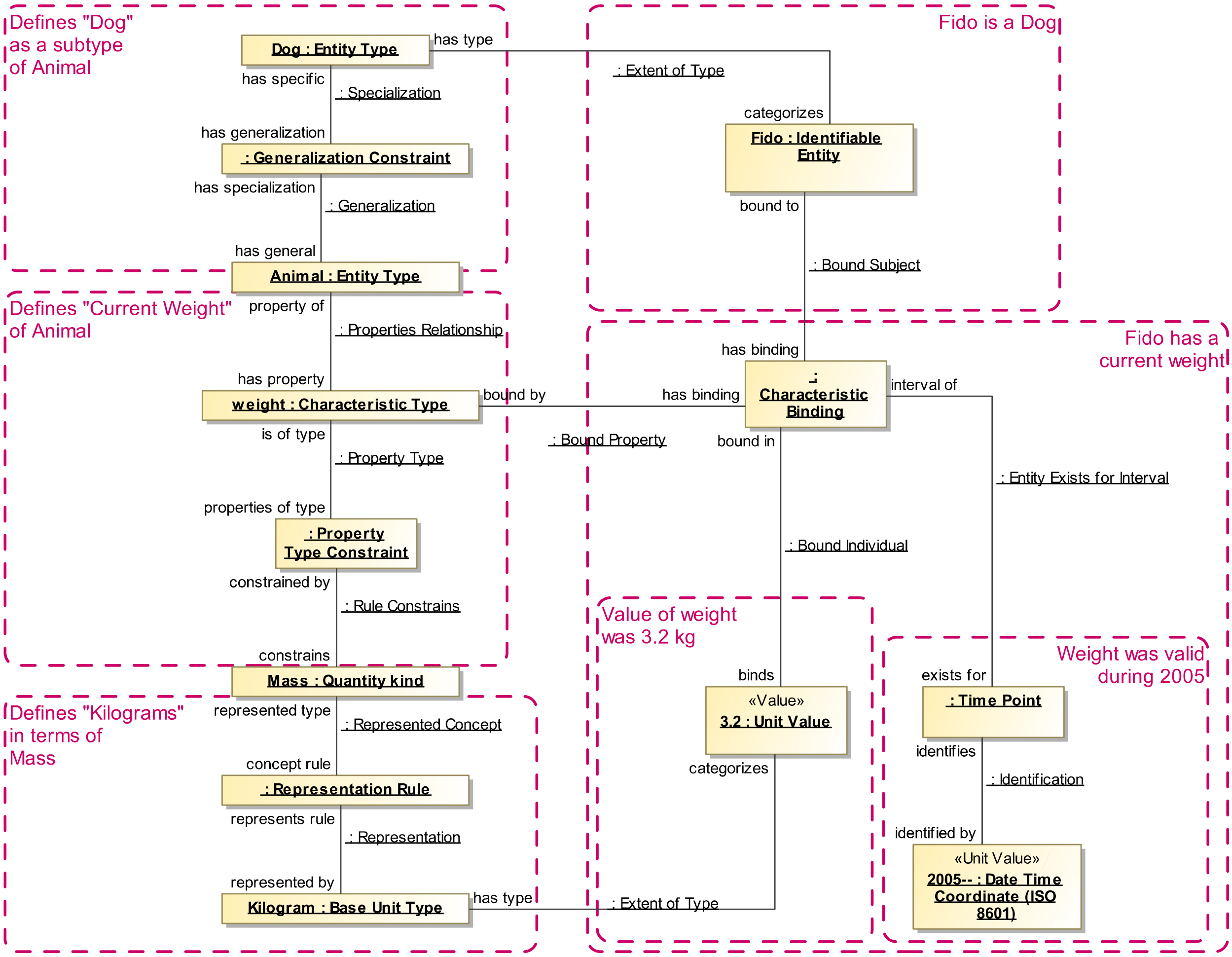


Figure 5.23: Instance Model Defining Characteristic & Value for an Entity

Figure 5.23 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.

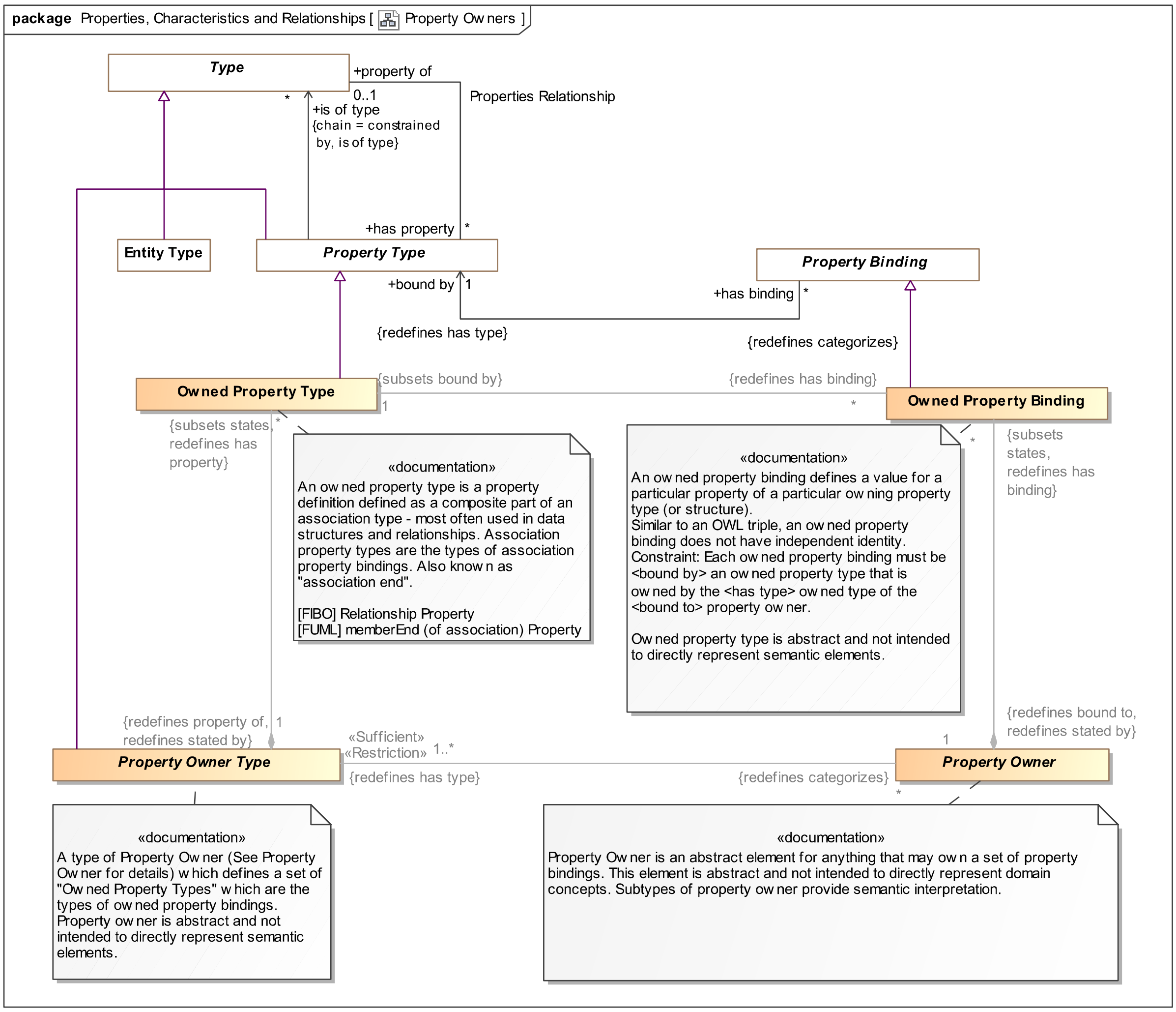
This model uses some rules not yet defined: Generalization Constraint, Property Type Constraint and Representation Rule – you may refer to the reference section for details on these rules. Time point and Date Time Coordinate come from the [ThreatRisk] example model. Rules are used rather than simple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

5.7.3 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.

5.7.4 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].

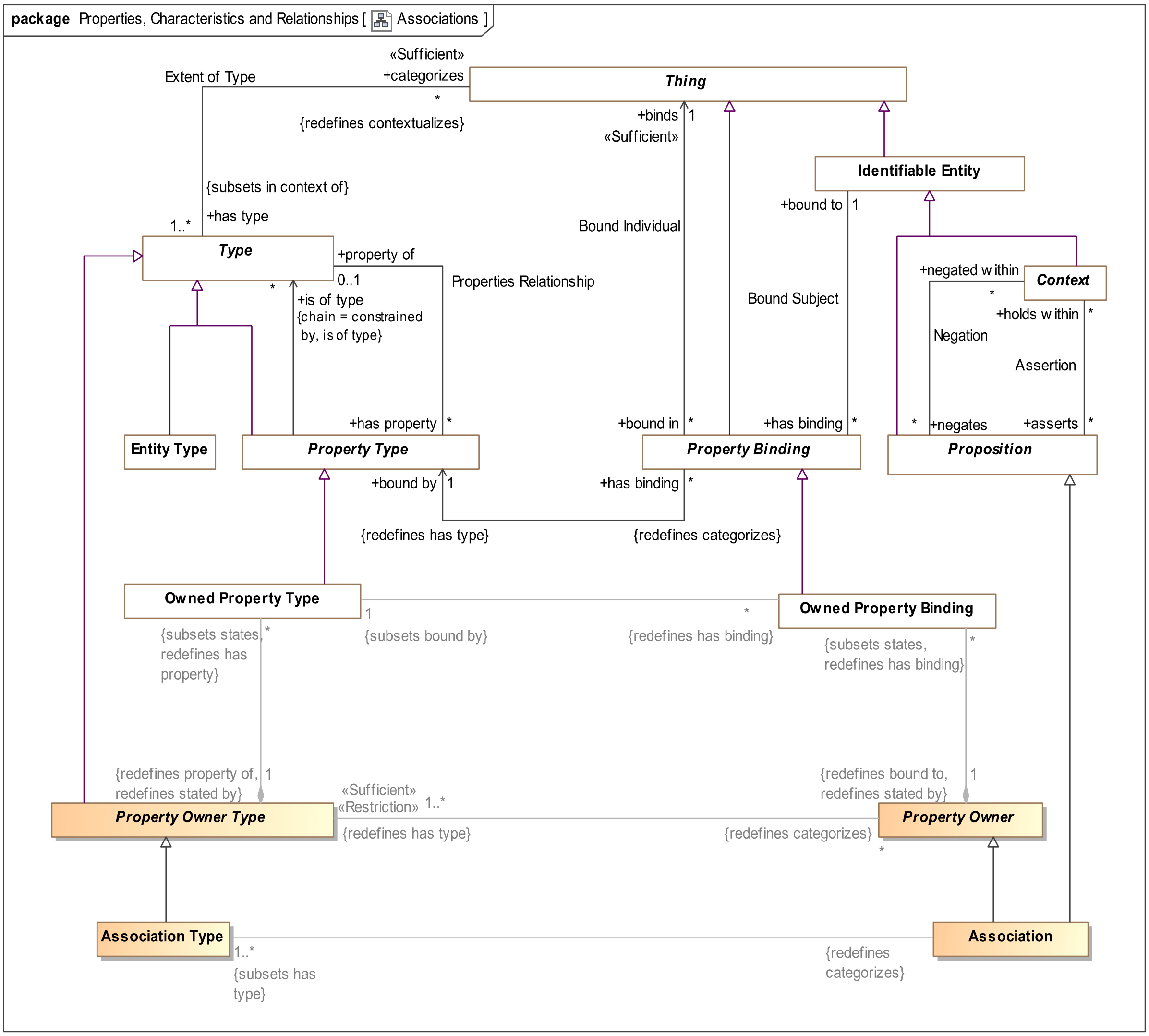


Figure 5.24: Associations

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

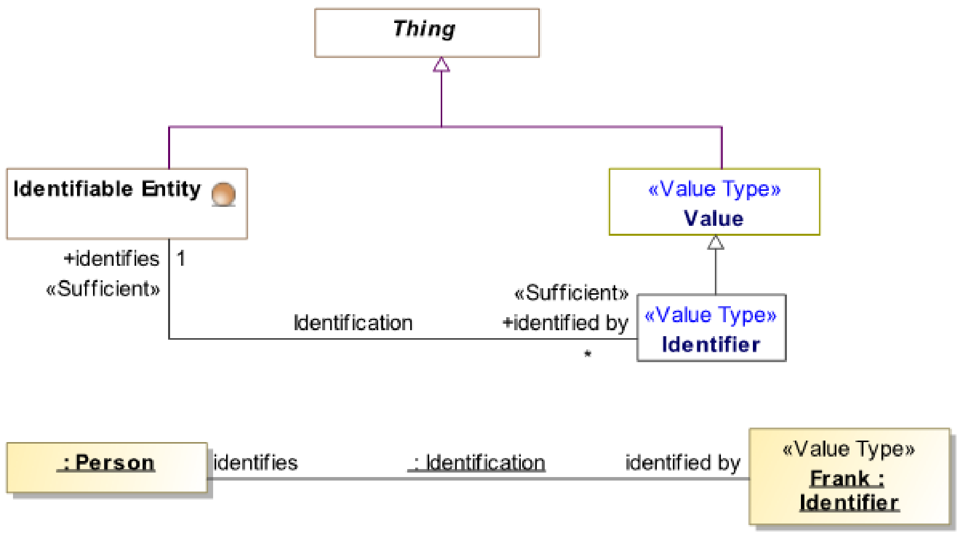


Figure 5.25: Association Example

We have already seem multiple associations, in figure 5.25 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:

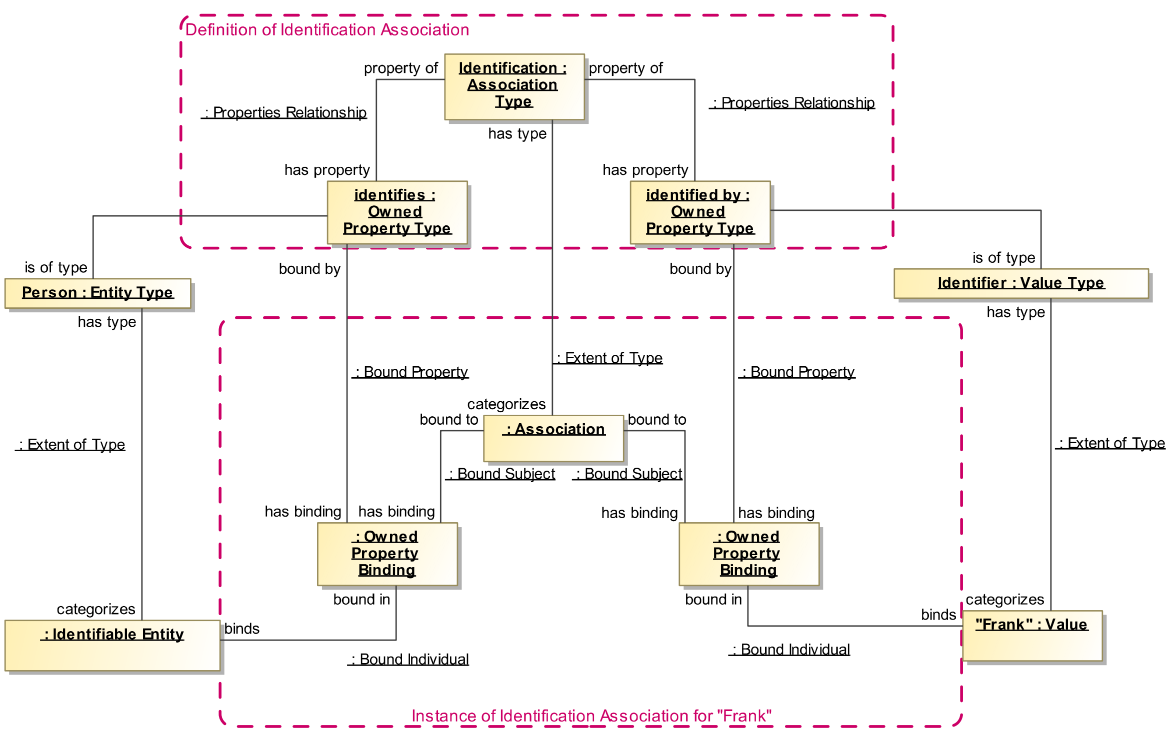


Figure 5.26: SMIF model instances for an association

In 5.26 we see the SMIF model instances corresponding to the UML profile view in figure 5.25. 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.

5.7.5 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.

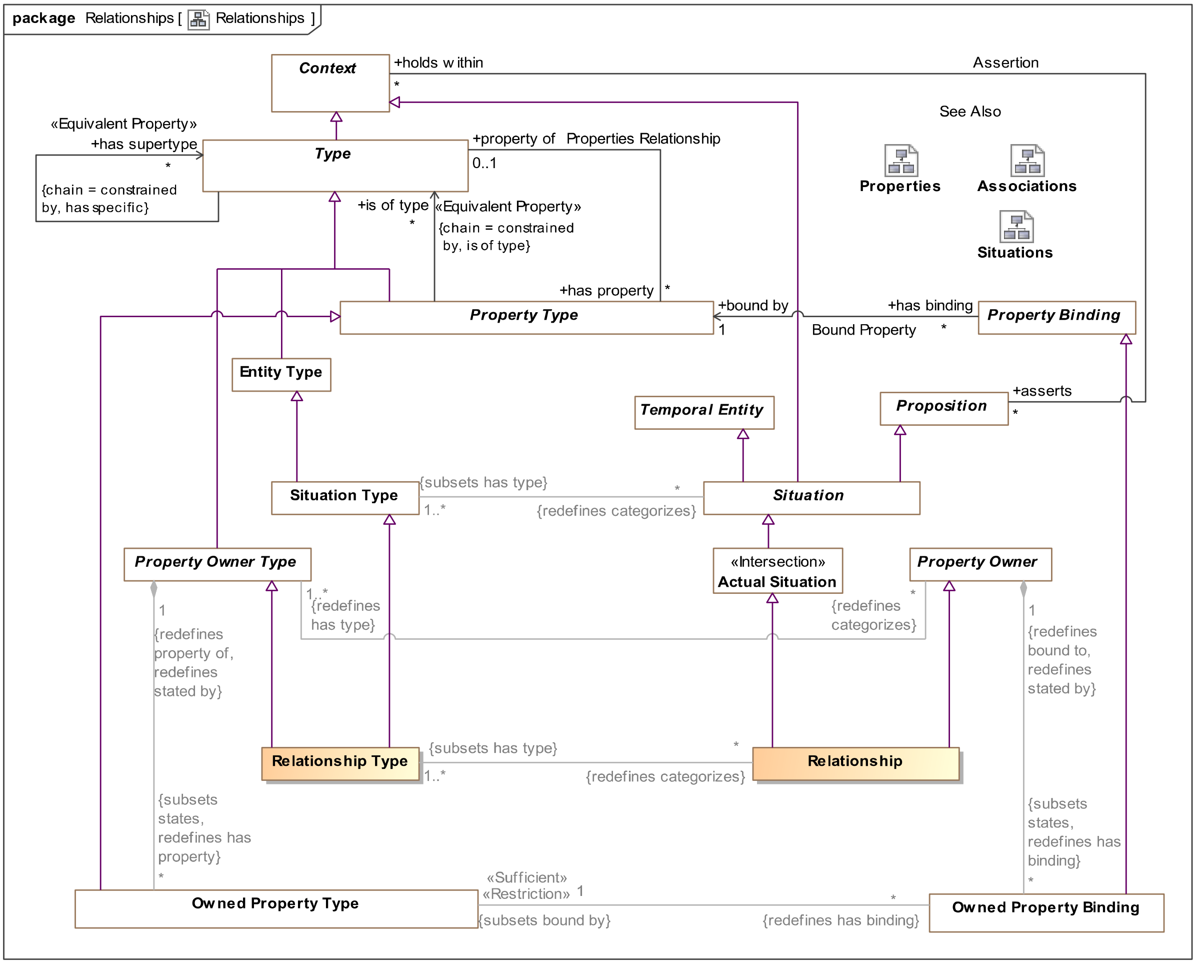


Figure 5.27: Defining Relationships

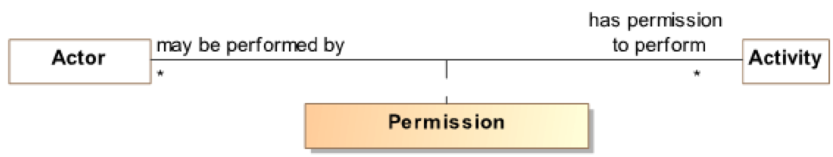
Figure 5.27 shows the SMIF conceptual model defining relationships, building on concepts we have already seen such as situations, associations and properties.

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.

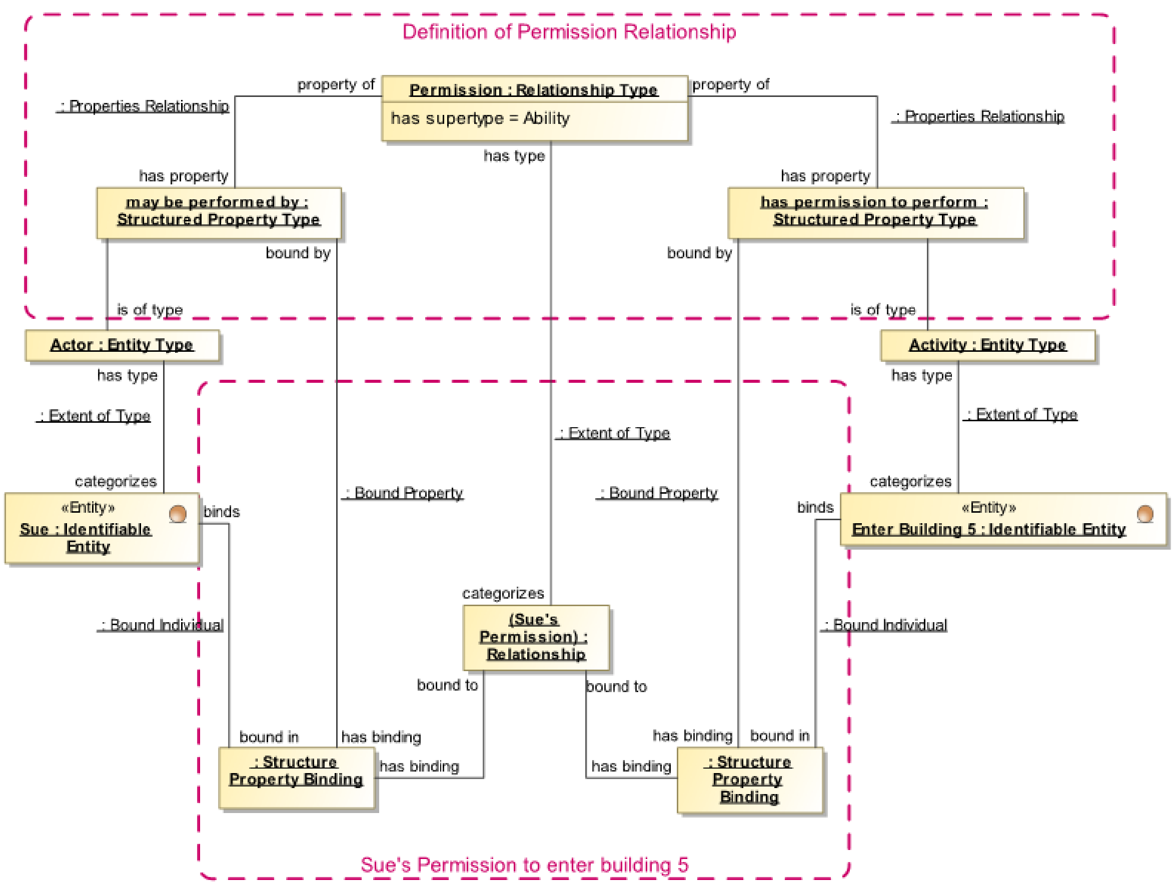


Figure 5.28: Defining and Using a Relationship

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.

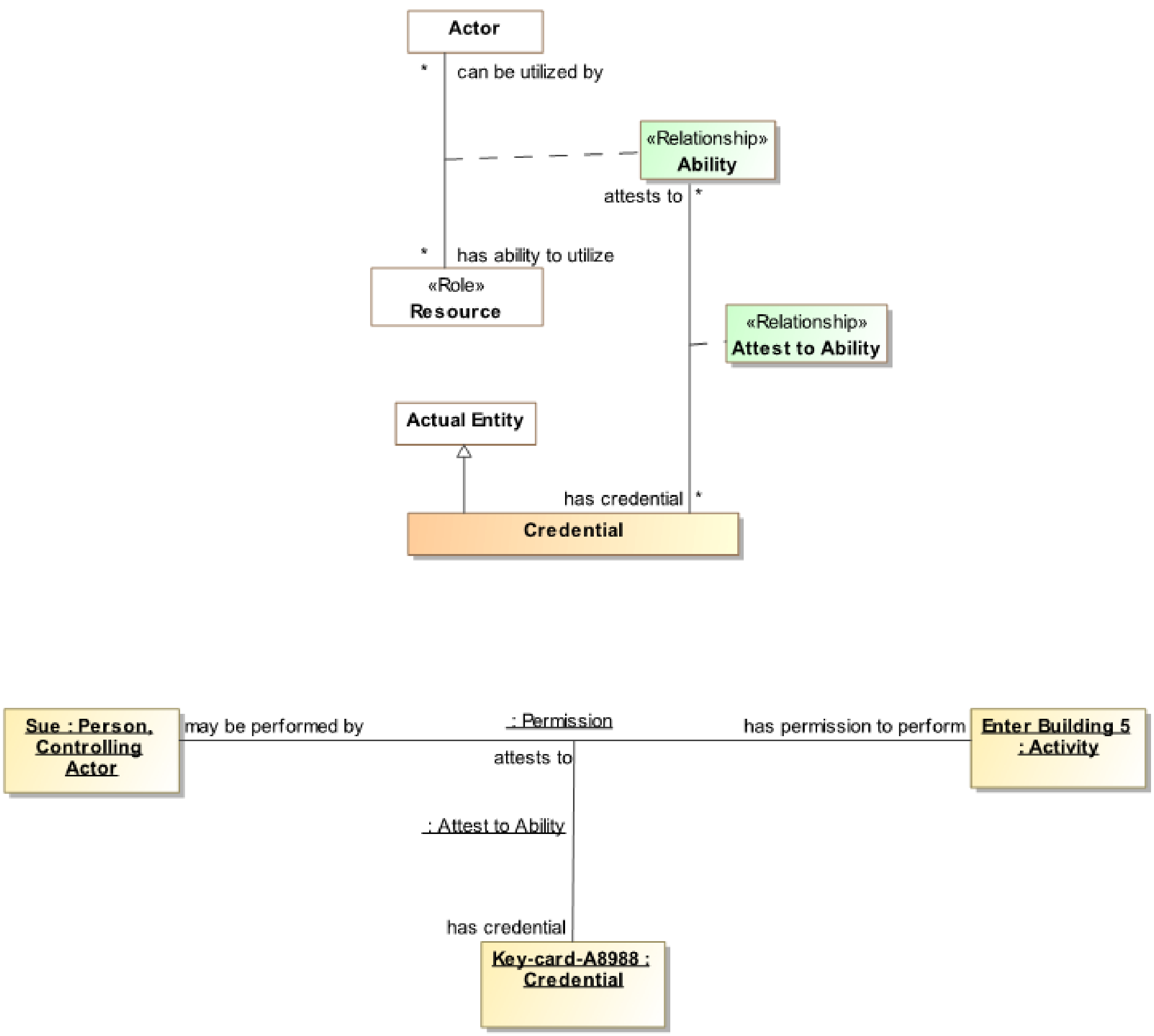


Figure 5.29: Relationship Involving Relationships

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 5.29 we see an instance of the relationship that is at the top of figure 5.29, 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.

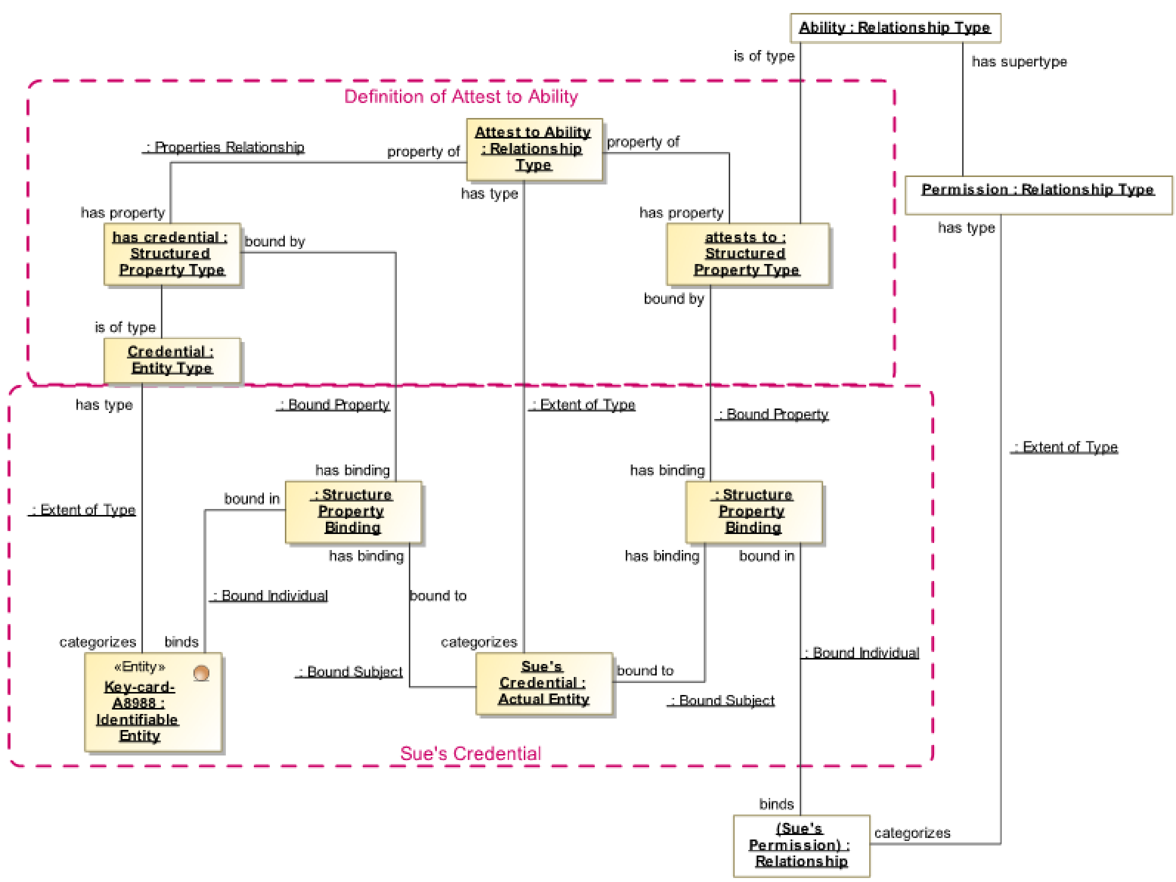


Figure 5.30: Relationship Involving Relationships – instance model

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.

5.8 Composition and Sequencing of Actual Situations

In section 5.4 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.

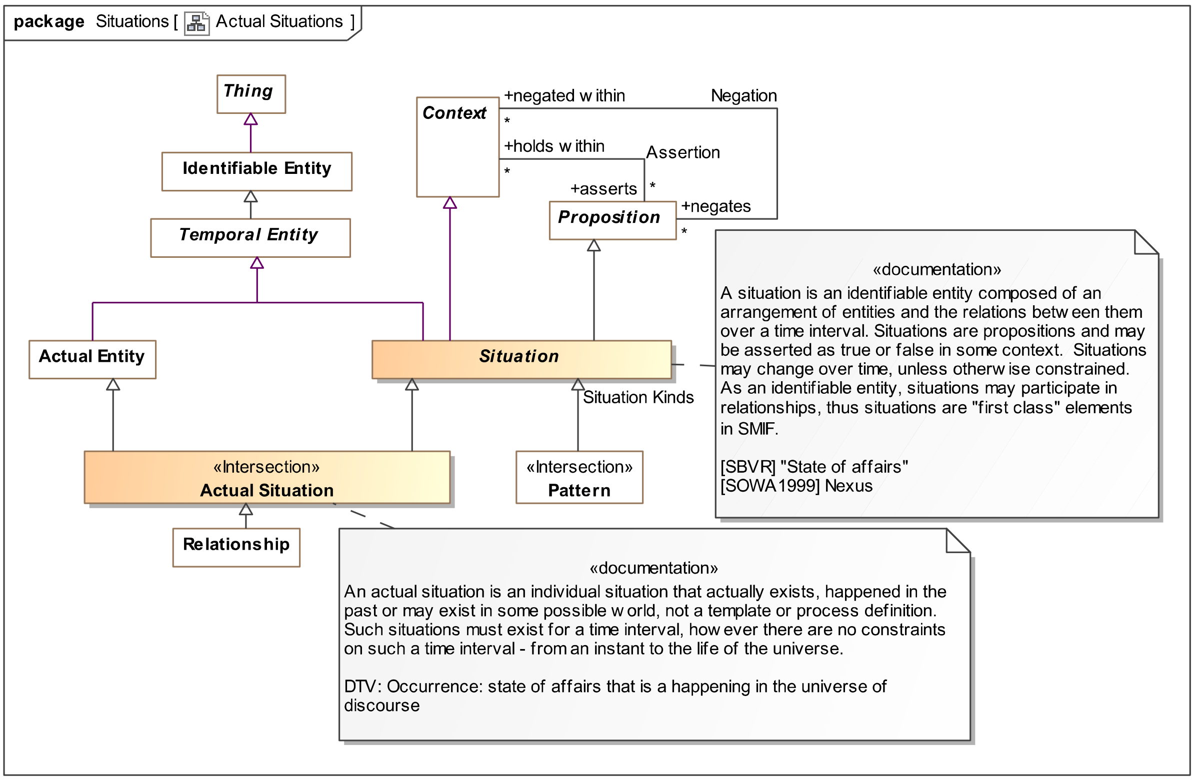


Figure 5.31: Actual Situations

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 5.31 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 5.7, 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.

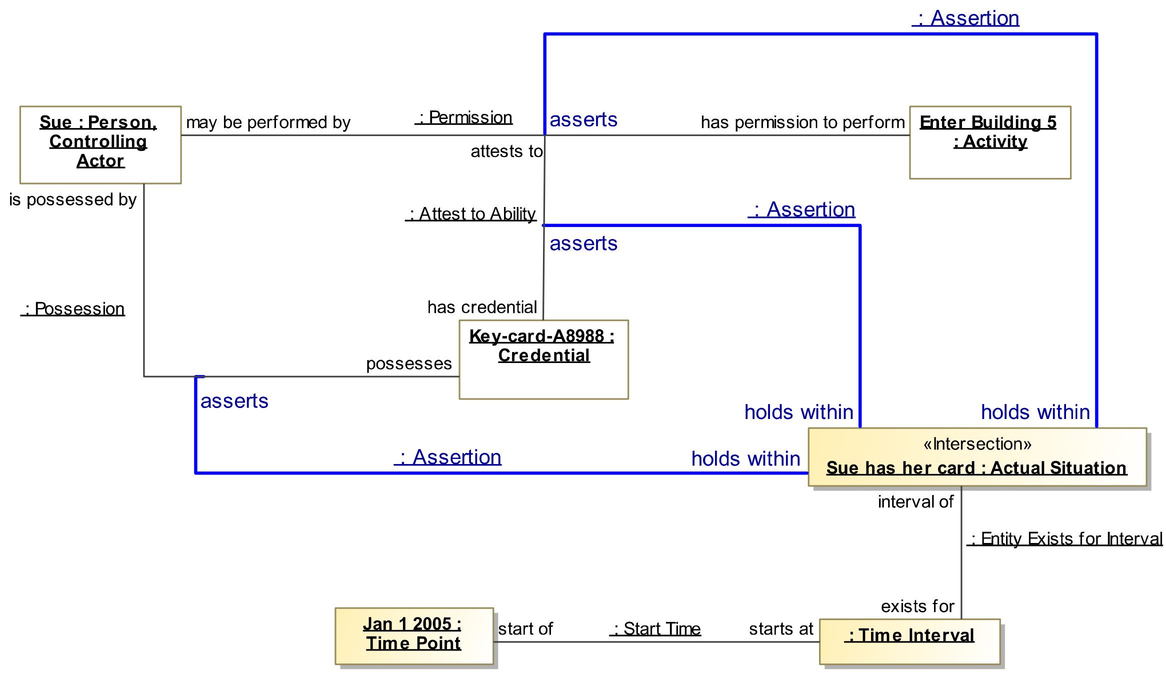


Figure 5.32: Initial Situation Example

Figure 5.32 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?

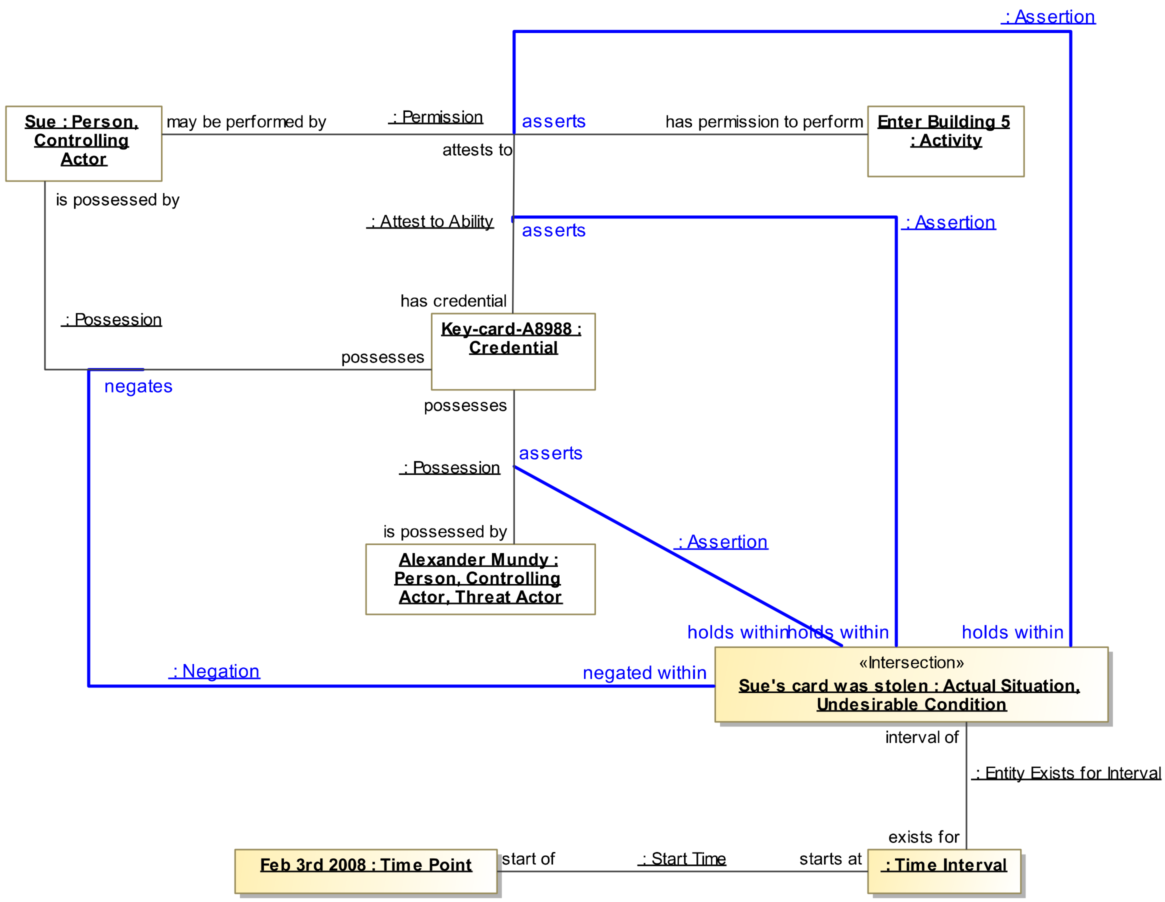


Figure 5.33: Example Situation After Theft

In an attack (not shown), Sue's card was stolen by "Alexander Mundy[6]", 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!

[6] "It takes a thief" Television Series

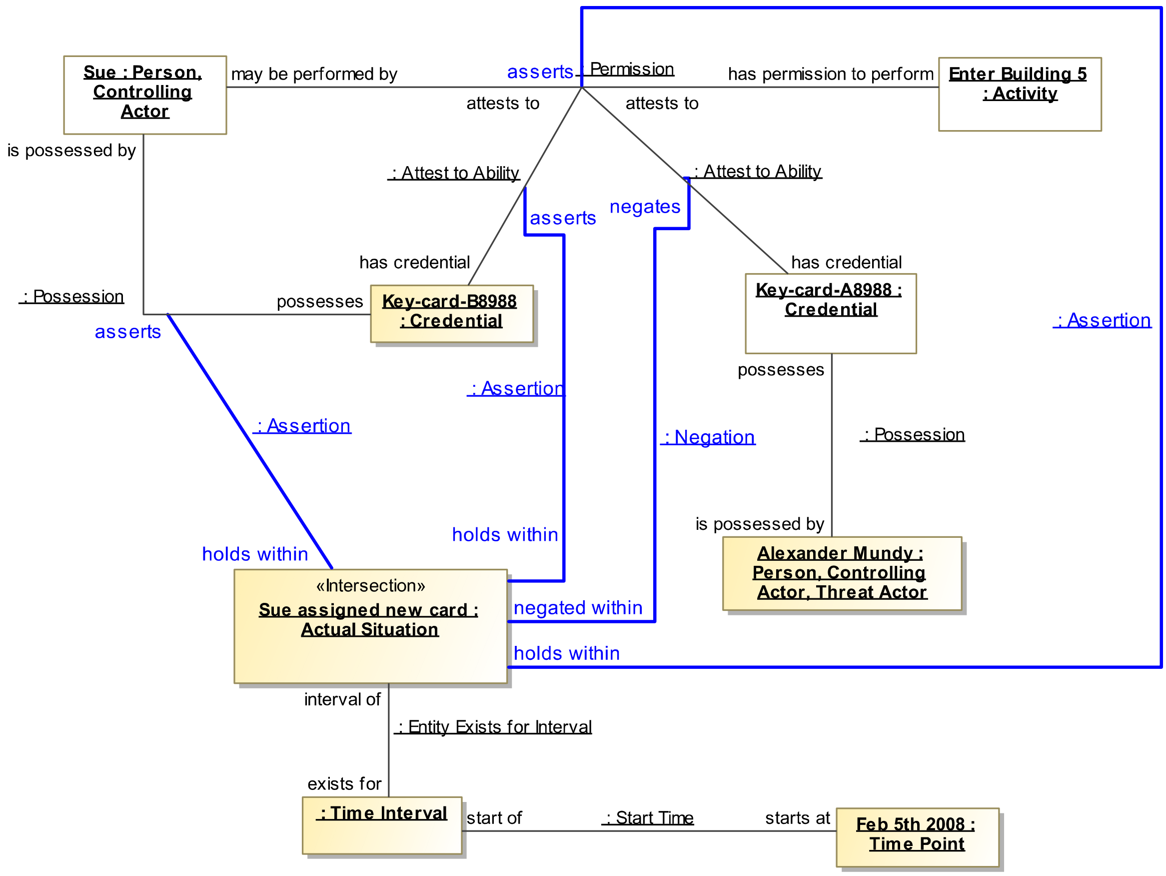


Figure 5.34: Example Situation After Mediation

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.

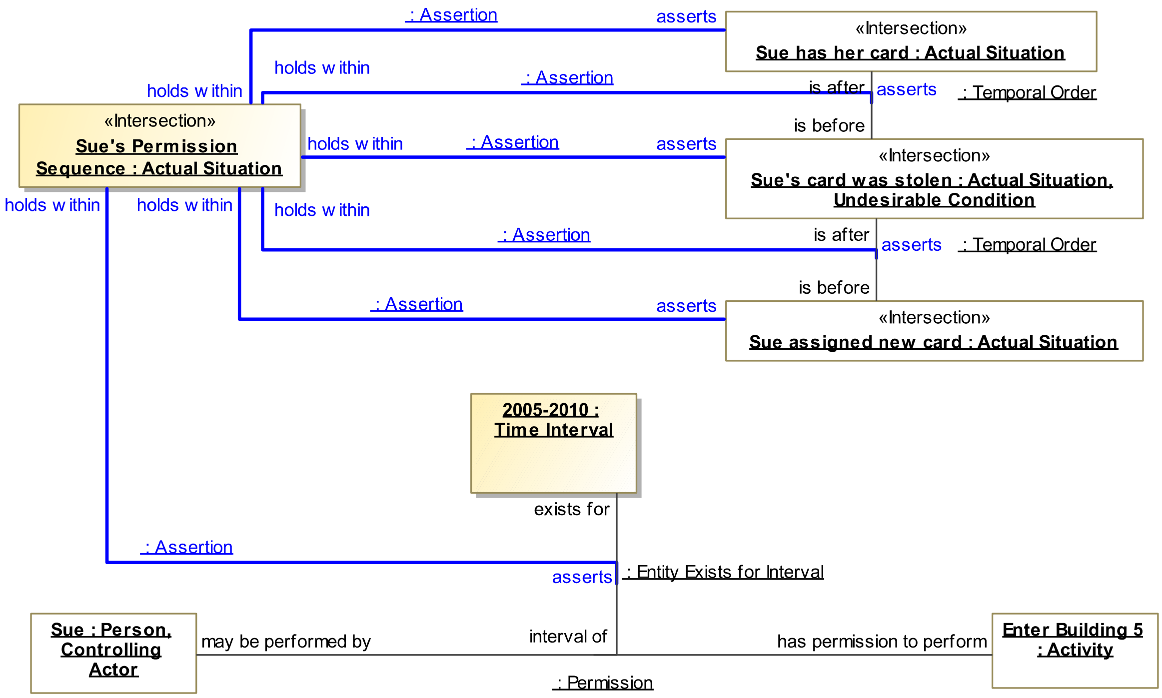


Figure 5.35: Sequence of Situations Example

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.

5.9 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.

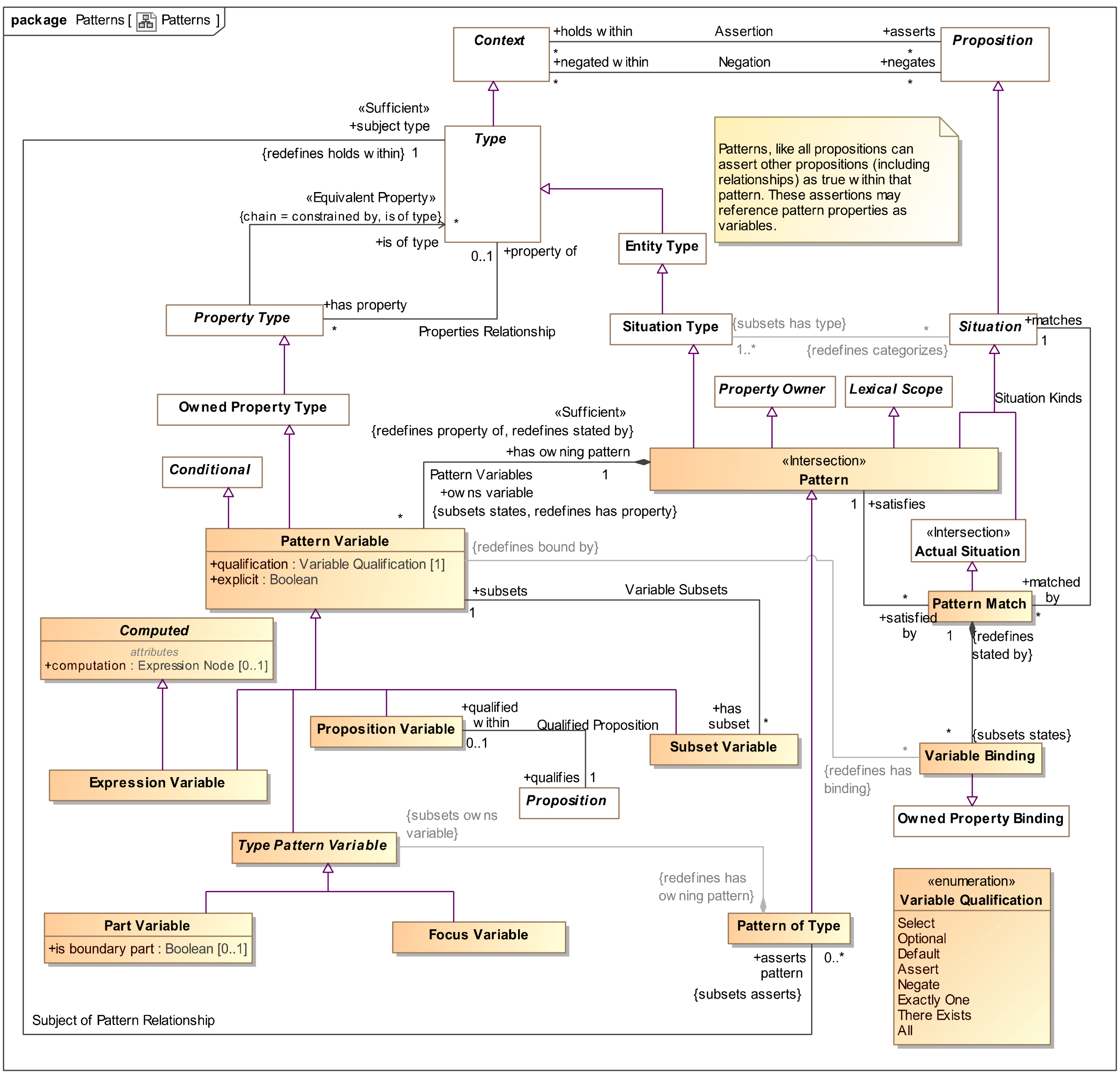


Figure 5.36: Full Pattern Model

For context, figure 5.36 presents the full pattern model without further comment. We will "build up" these concepts one step at a time, below.

5.9.1 Patterns – top level

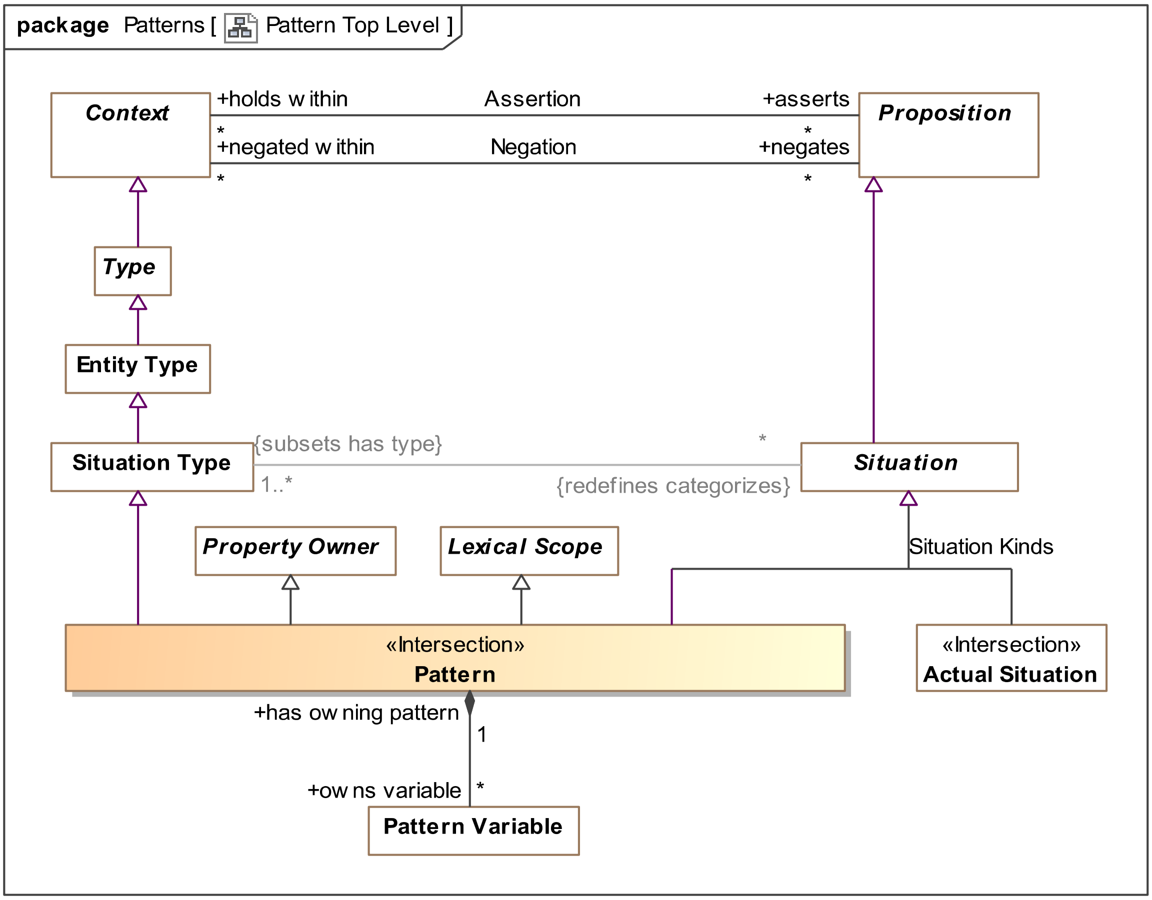


Figure 5.37: Patterns - Top Level Model

Figure 5.37 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.

5.9.2 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.

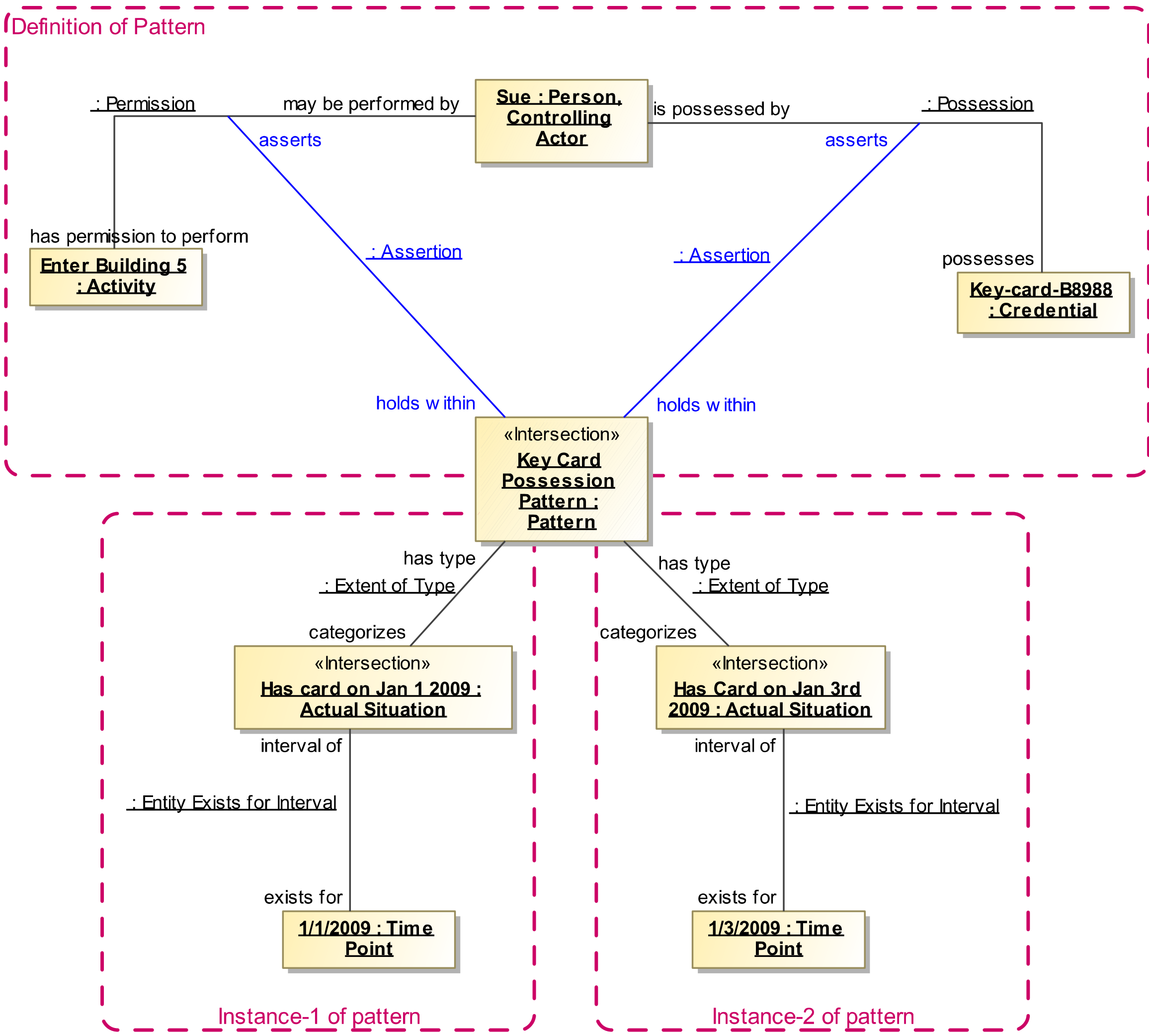


Figure 5.38: Repeated Pattern Example

In figure 5.38 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.

5.9.3 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.

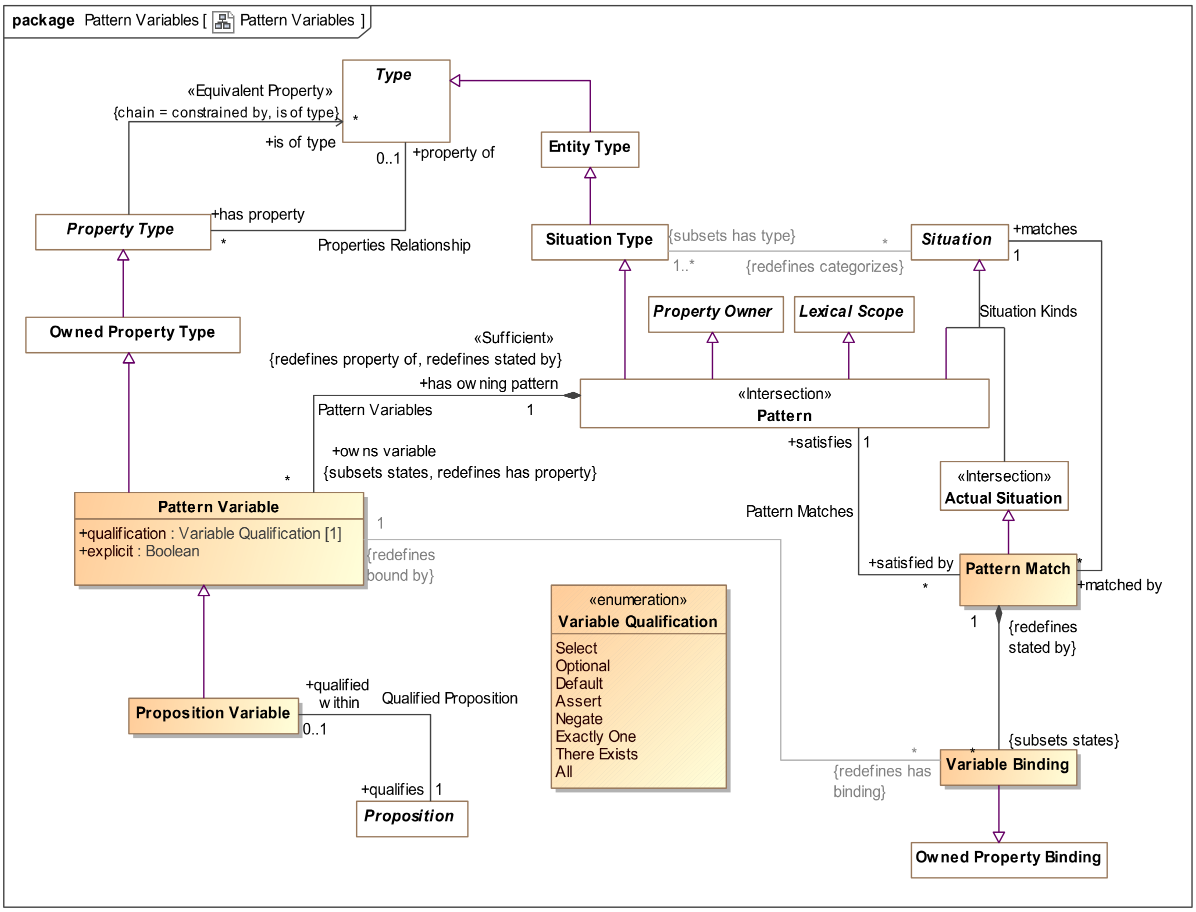


Figure 5.39: Pattern Variables

Figure 5.39 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 5.9.6.

5.9.4 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.

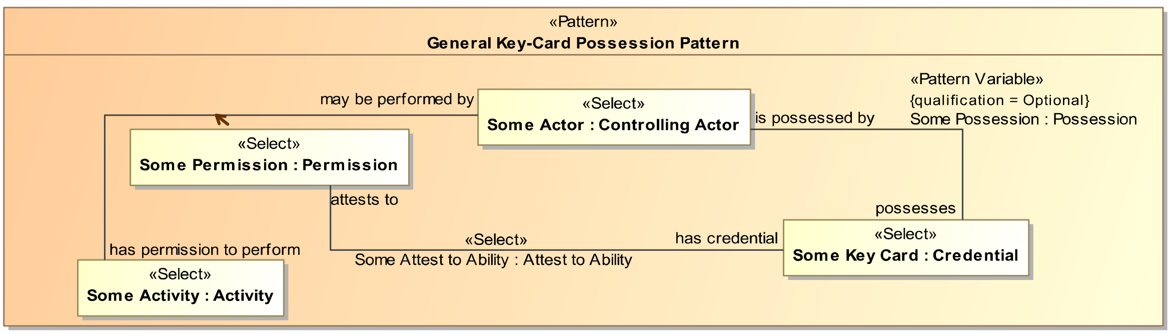


Figure 5.40: Patterns in UML Profile Example

Keeping with our example based on Sue and her key-card, figure 5.40 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.

5.9.5 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:

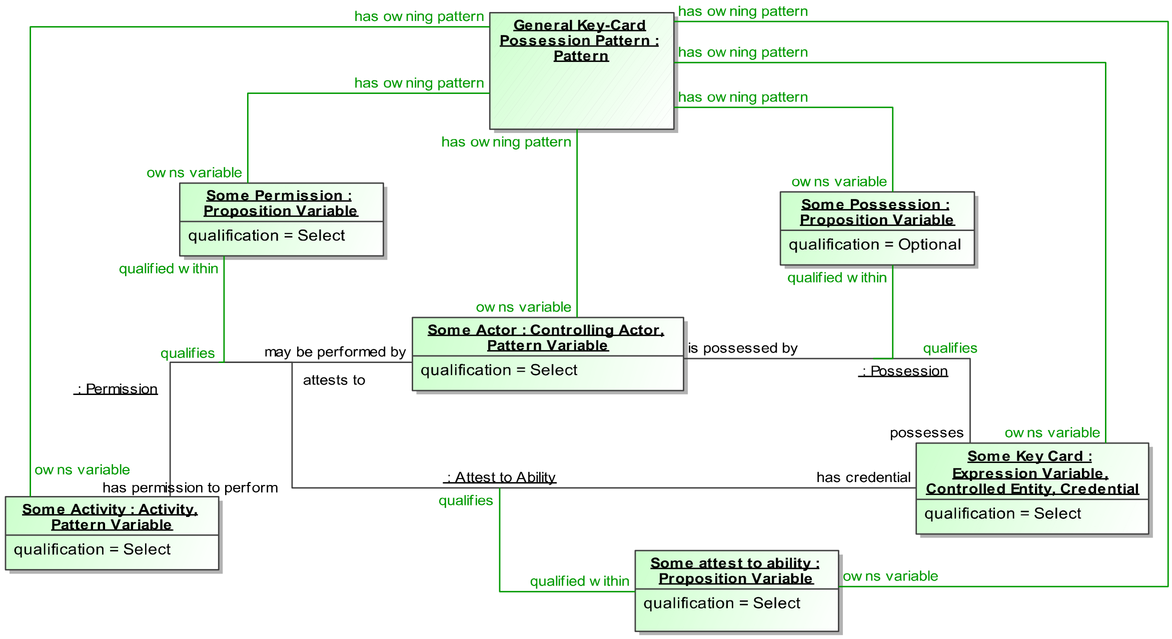


Figure 5.41: SMIF Model Example of Pattern Variables

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.

5.9.6 Pattern Matching

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

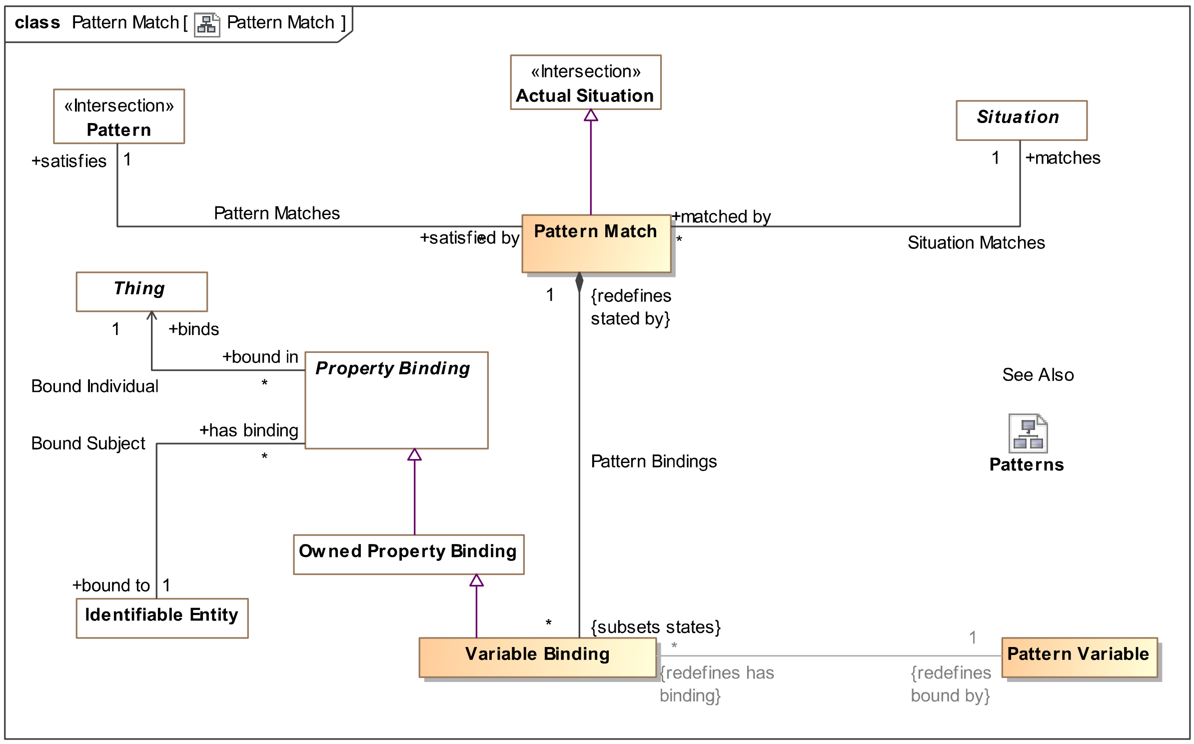


Figure 5.42: Pattern Matching Model

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.

5.9.7 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 5.9.5.

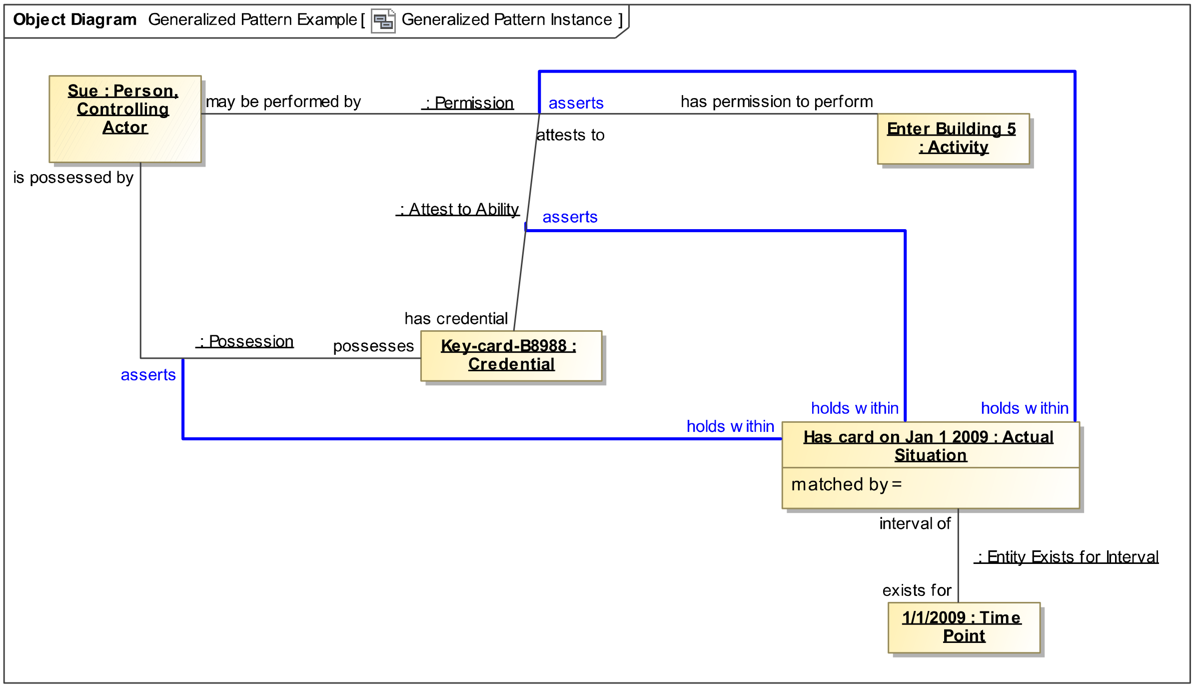


Figure 5.43: Potential instance of a pattern

The above defines 3 “entities” and three relationships.

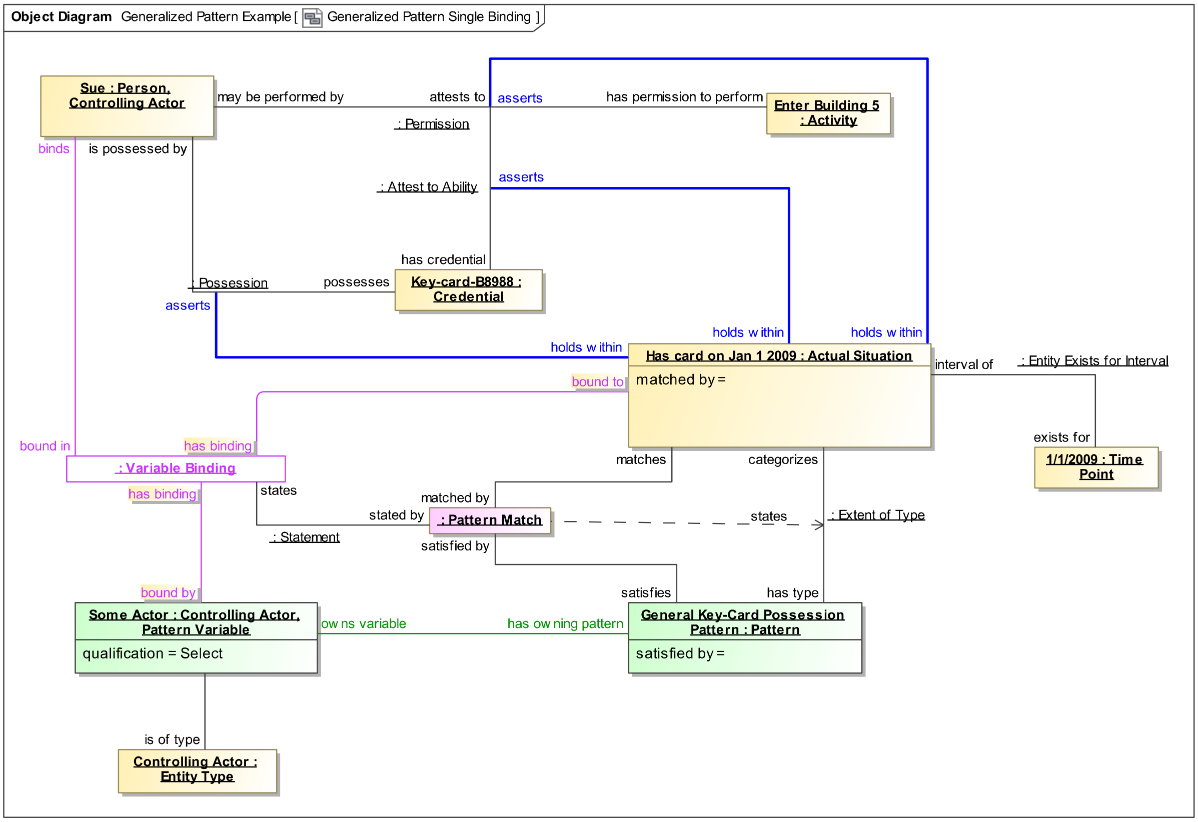


Figure 5.44: Binding of a single variable

Figure 5.44 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.

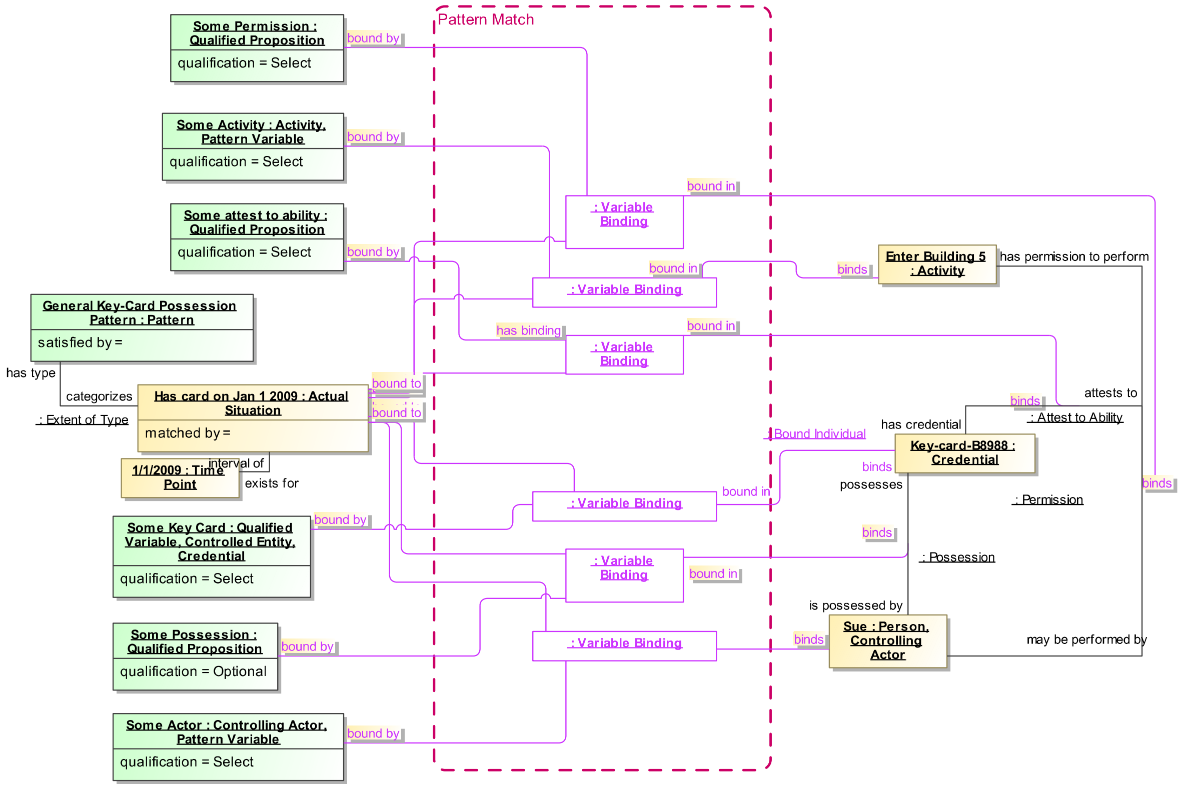


Figure 5.45: Full Binding of Pattern

Figure 5.45 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.

5.9.8 Computed Variables

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

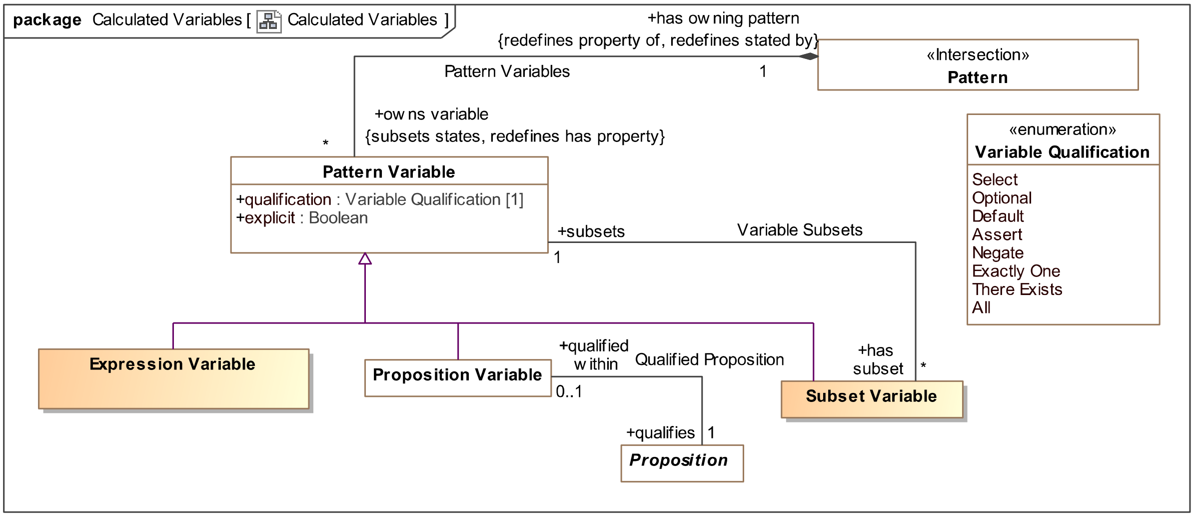


Figure 5.46: Subset and Expression Variables

Figure 5.46 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.

5.9.9 Subset Variable Example

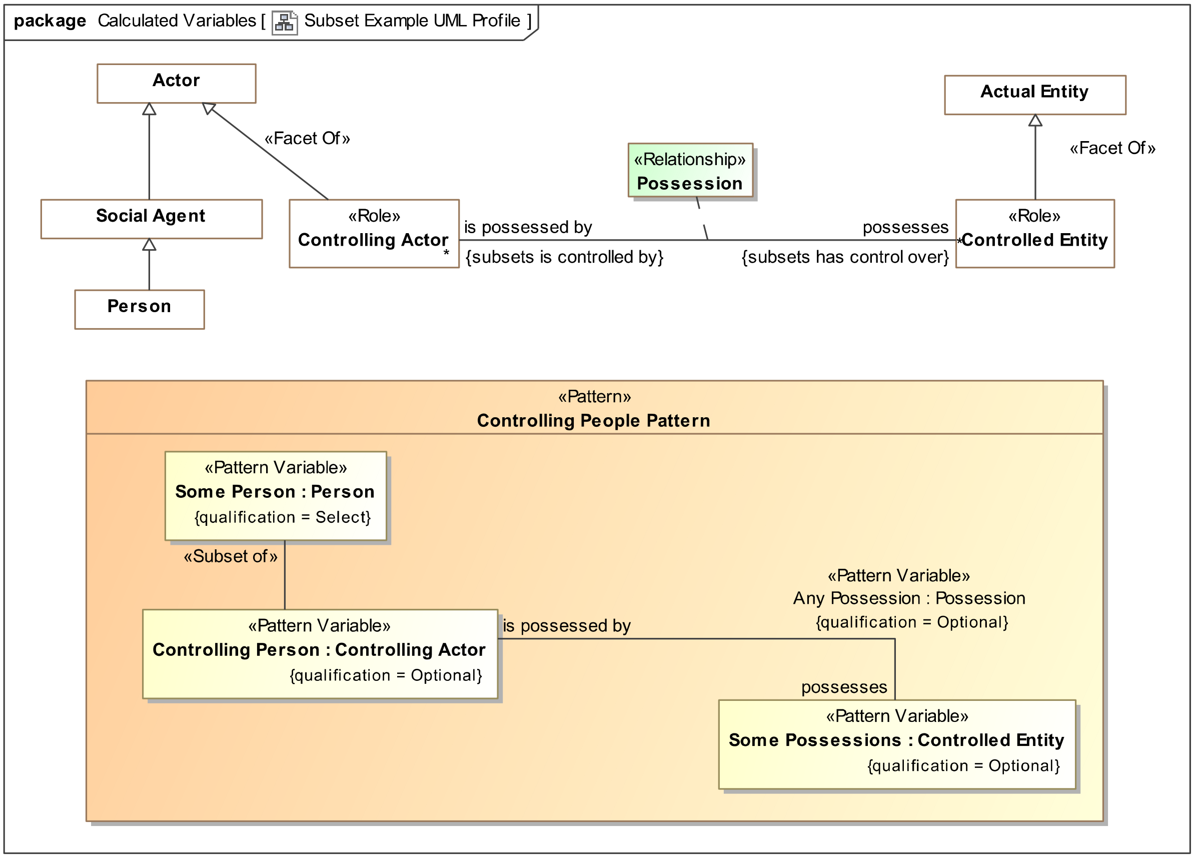


Figure 5.47: Subset Variable Example in UML Profile

Figure 5.47 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.

5.9.10 Controlling Person Pattern in the SMIF Model

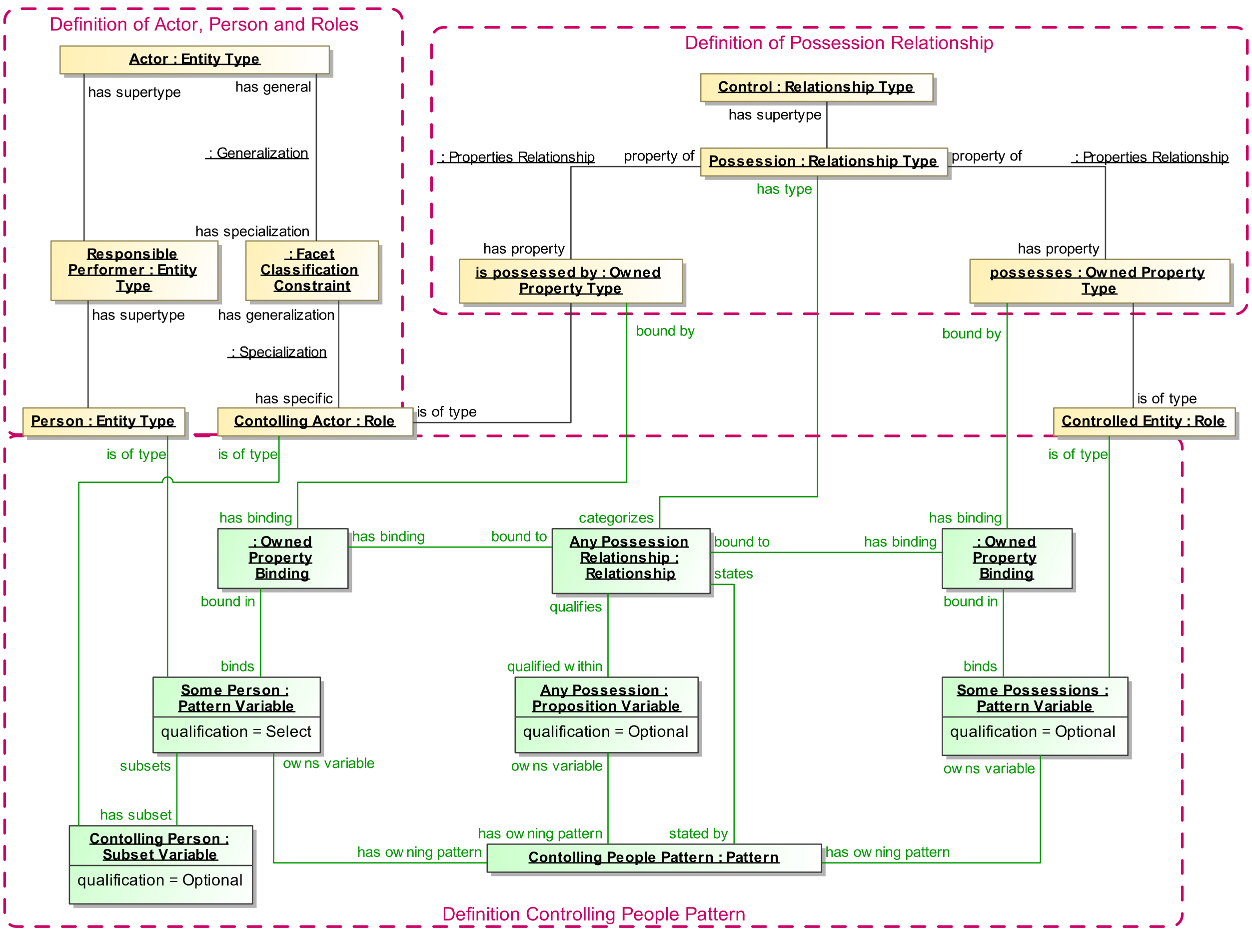


Figure 5.48: SMIF Model Instances of the Controlling Actor Pattern

The UML Profile diagram in figure 5.47 has a corresponding model as instances of the SMIF model as shown in figure 5.48. 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”.

5.10 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 5.9). 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.

5.10.1 Mapping Components Example

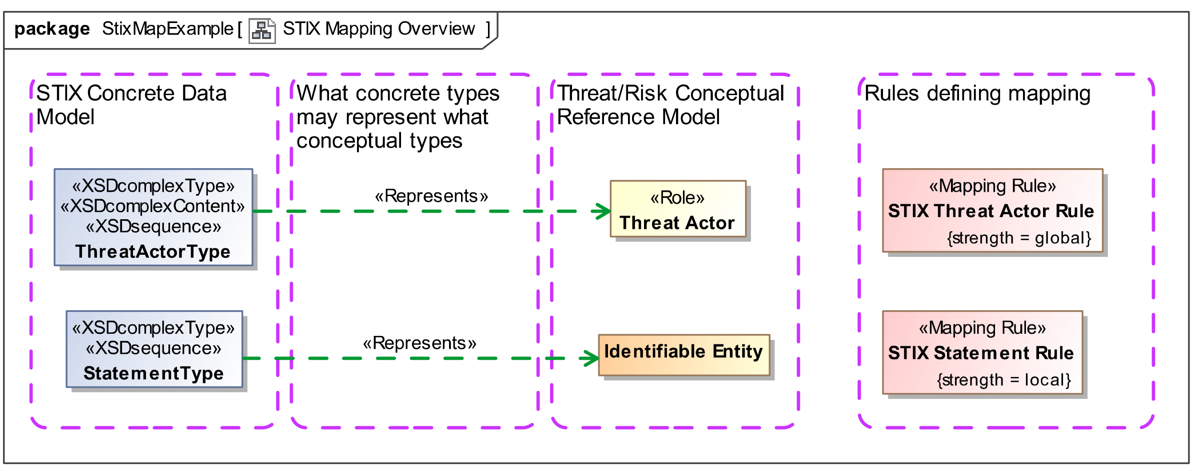


Figure 5.49: STIX Mapping Overview

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 5.49 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.

5.10.2 STIX Concrete Data Model

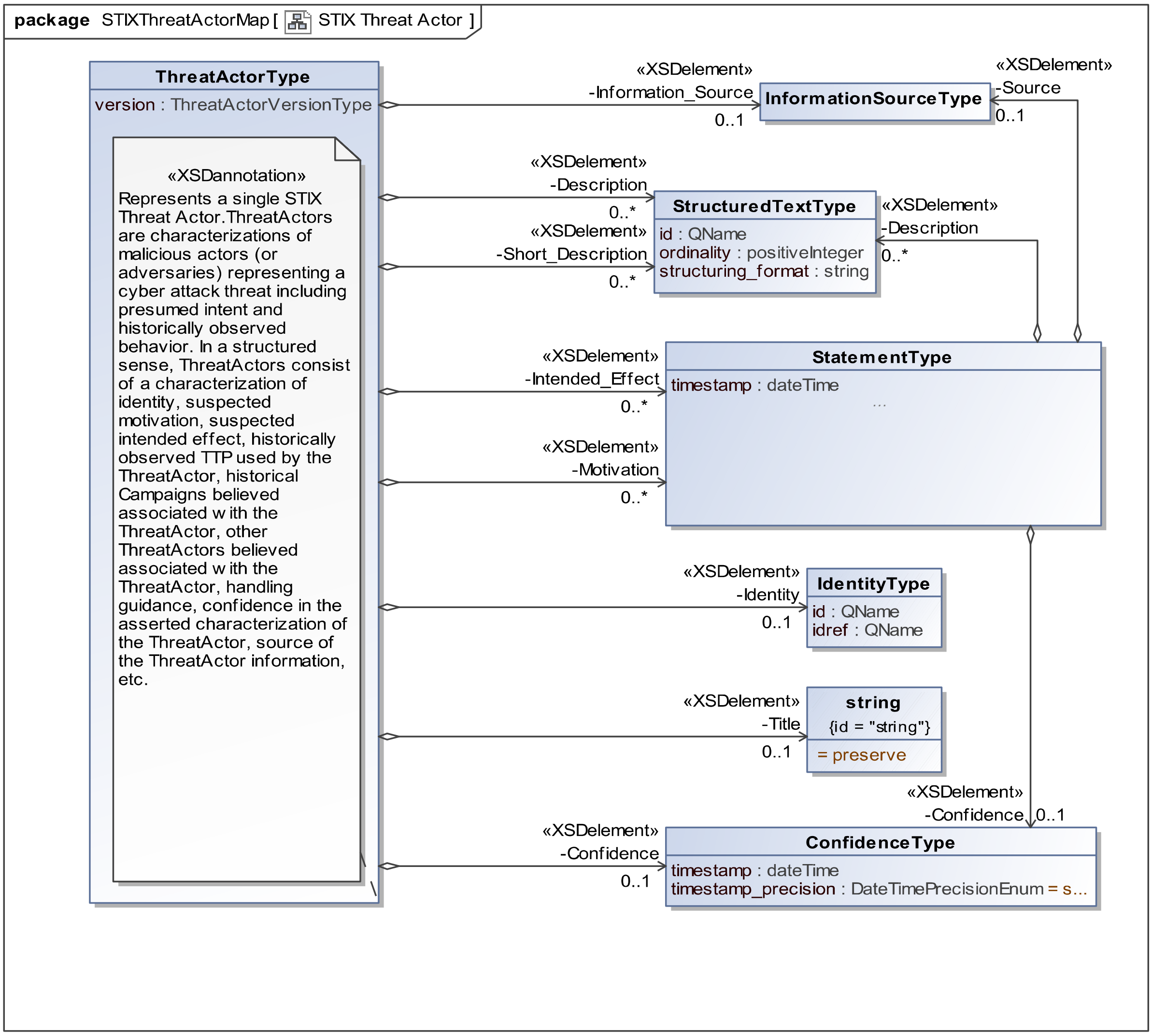


Figure 5.50: STIX Concrete Model Fragment

Figure 5.50 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.

5.10.3 OTR Conceptual Reference Model

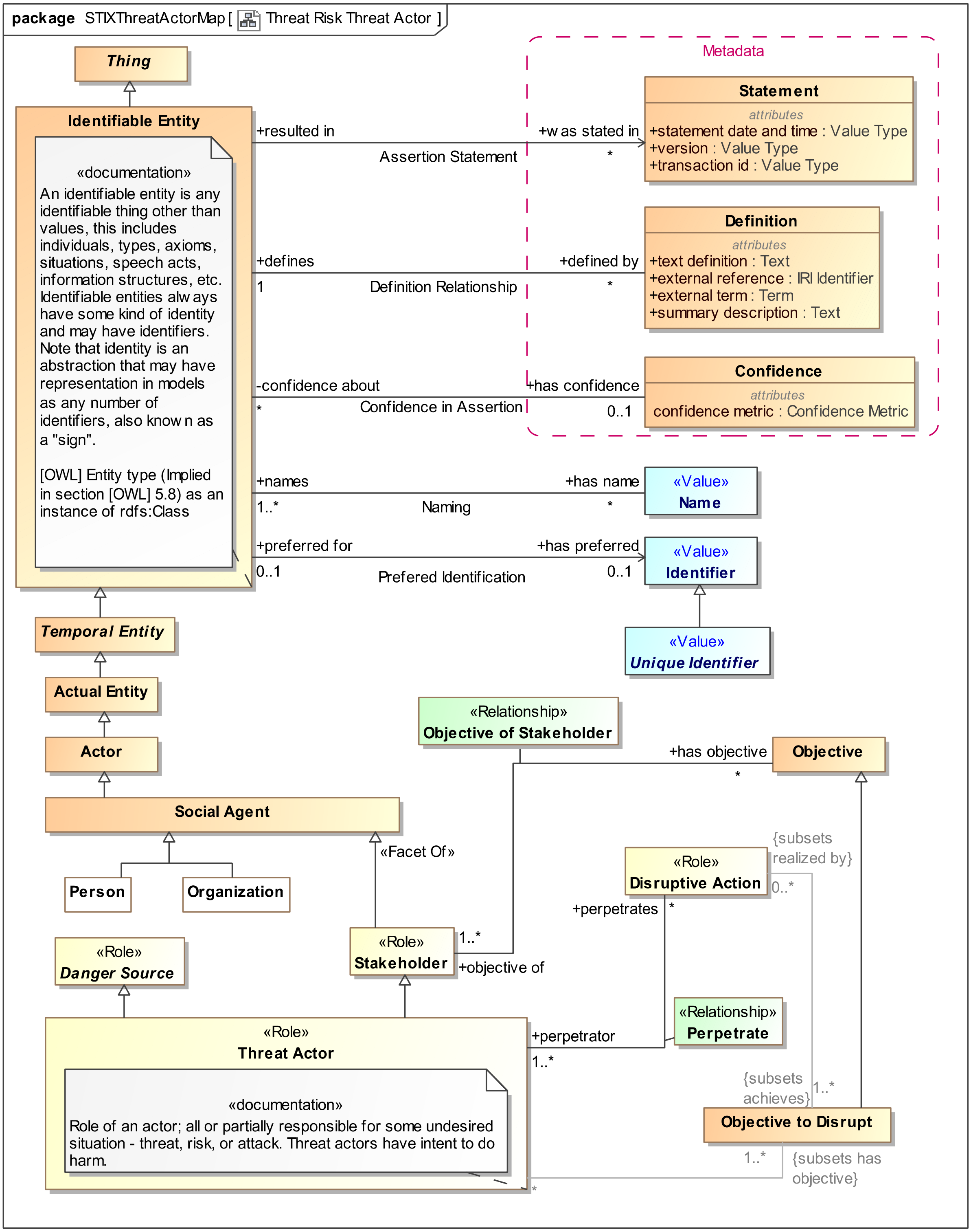


Figure 5.51: OTR Conceptual Reference Model Fragment

The OTR model fragment is figure 5.51 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.

5.10.4 STIX / OTR Mapping Rule

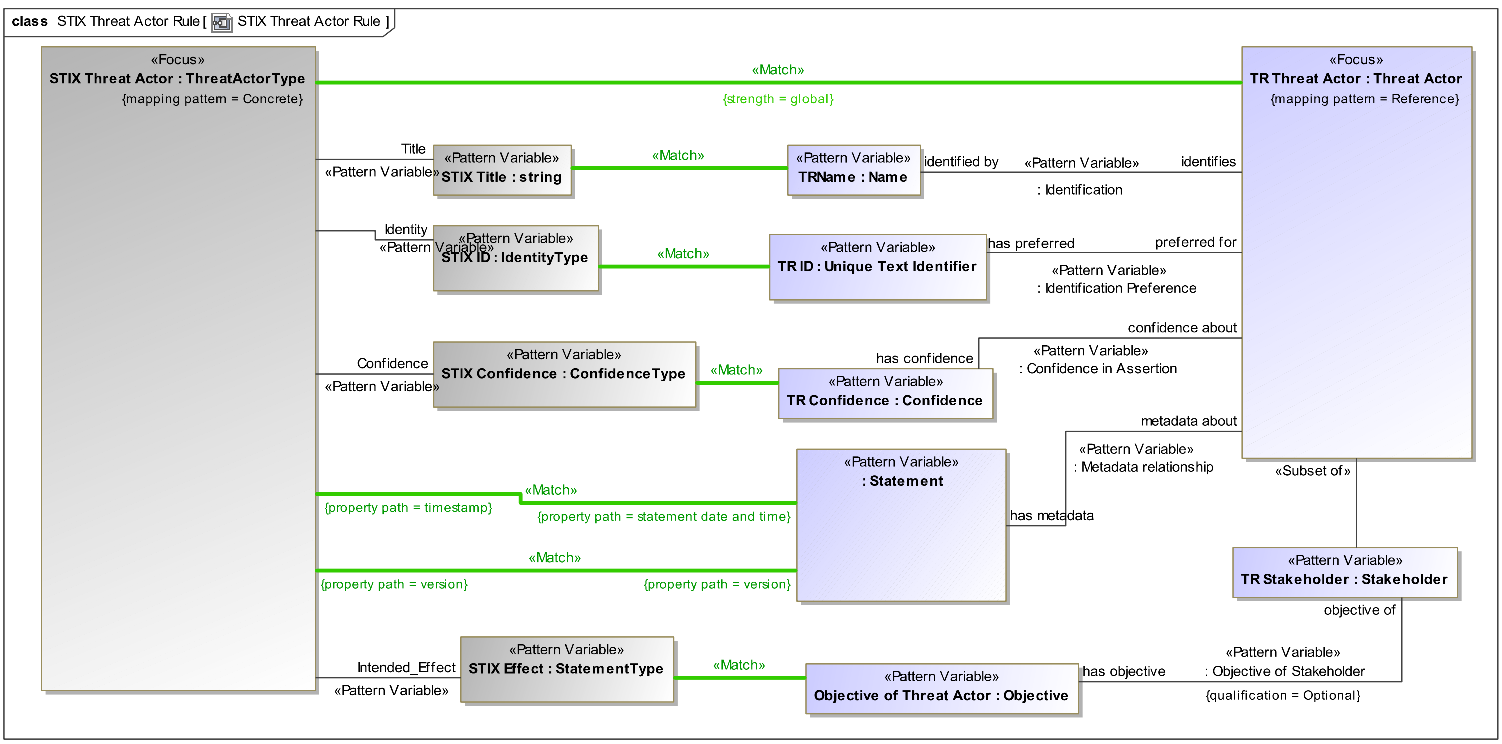


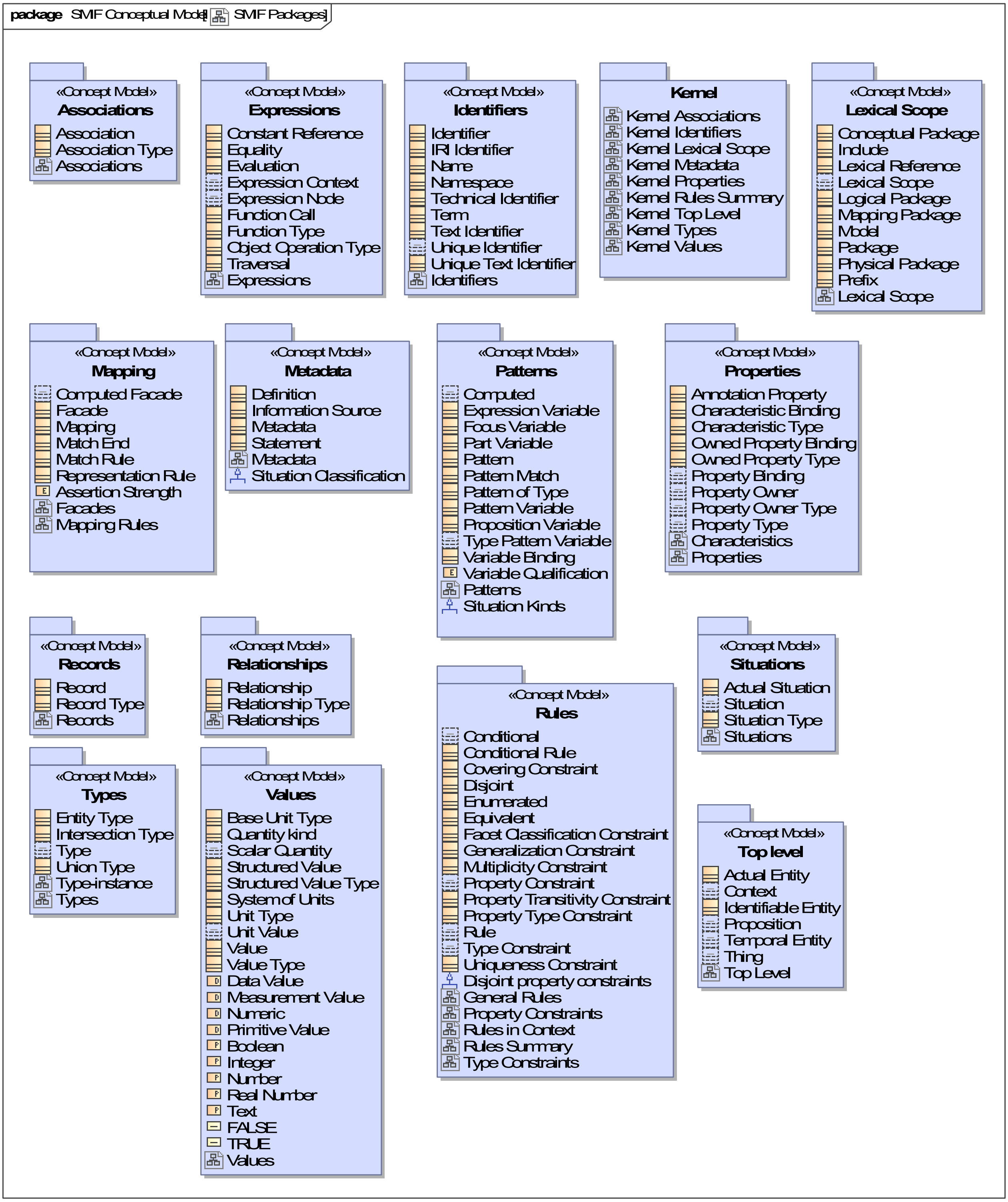
Figure 5.52: STIX - OTR Threat Actor Rule

Figure 5.52 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.

Additional examples of mapping included in the profile section.

6 SMIF Conceptual Model Reference (Normative)

6.1 Diagram: SMIF Packages



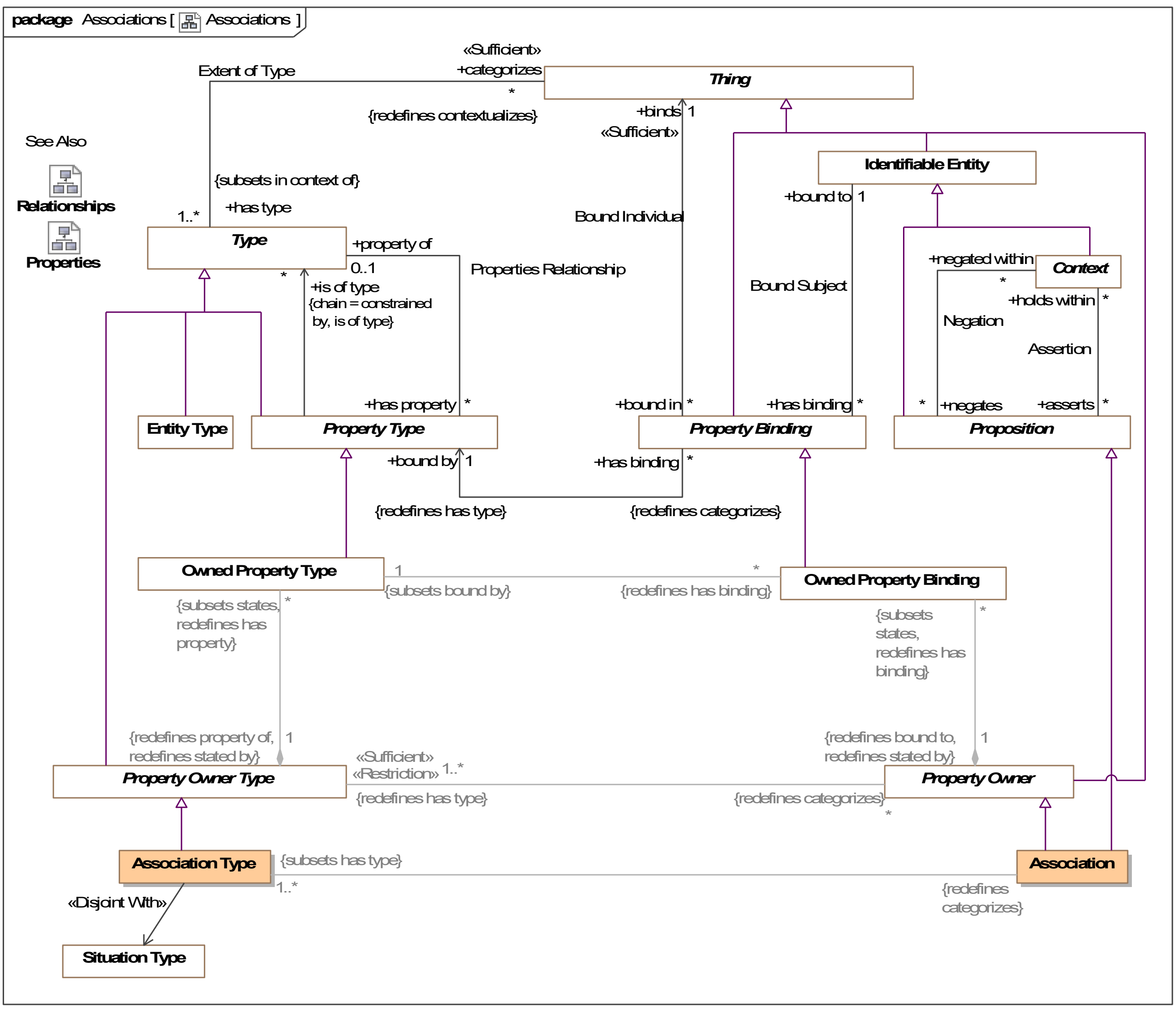
f. SMIF Packages

7 SMIF Conceptual Model::Associations

An association asserts a formal condition involving related things, the association ends. An association may be asserted within a context as true or false within that context. Each association has a number of bindings of which are immutable for that association.

Associations are differentiated from relationships in that associations are fully dependent on the things they relate. These are known as "formal", "thin", "internal" or "intrinsic" relations in much of the literature.

7.1 Diagram: Associations



g. Associations

7.2 Class Association

An association makes a logical statement involving related things, the association ends. An association may be asserted within a context as true or false within that context. Each association type has a number of bindings of which are immutable for that association.
An association may be true or false within its context and is atomic in its truth value.
Associations are differentiated from relationships in that associations are not situations - they are not temporal and do not change over time. Associations may be a consequence of relationships or other situations or may be derived from qualities of associated ends.
Associations can "own" owned property bindings as their "ends".
See also: Relationship
[Guizzardi] Intrinsic Relation
[UML] Link
[DOLCE] Formal Relation

Direct Supertypes

Property Owner, Proposition

Associations

/ <<Restriction>> : Association Type [1..*] *Subsets*: has type:Type

7.3 Class Association Type

A type of Association (See Association for details) which defines a set of "Association Property Types" which are the types of association property bindings. Associations are not situations - the are not temporal things. THis does not prevent subtypes of associations from being situations.
[Guizzardi] Intrinsic Relation Type
[UML] Association
[OWL] For binary associations, may be considered a pair of properties that are Inverse Object Properties.

Direct Supertypes

Property Owner Type

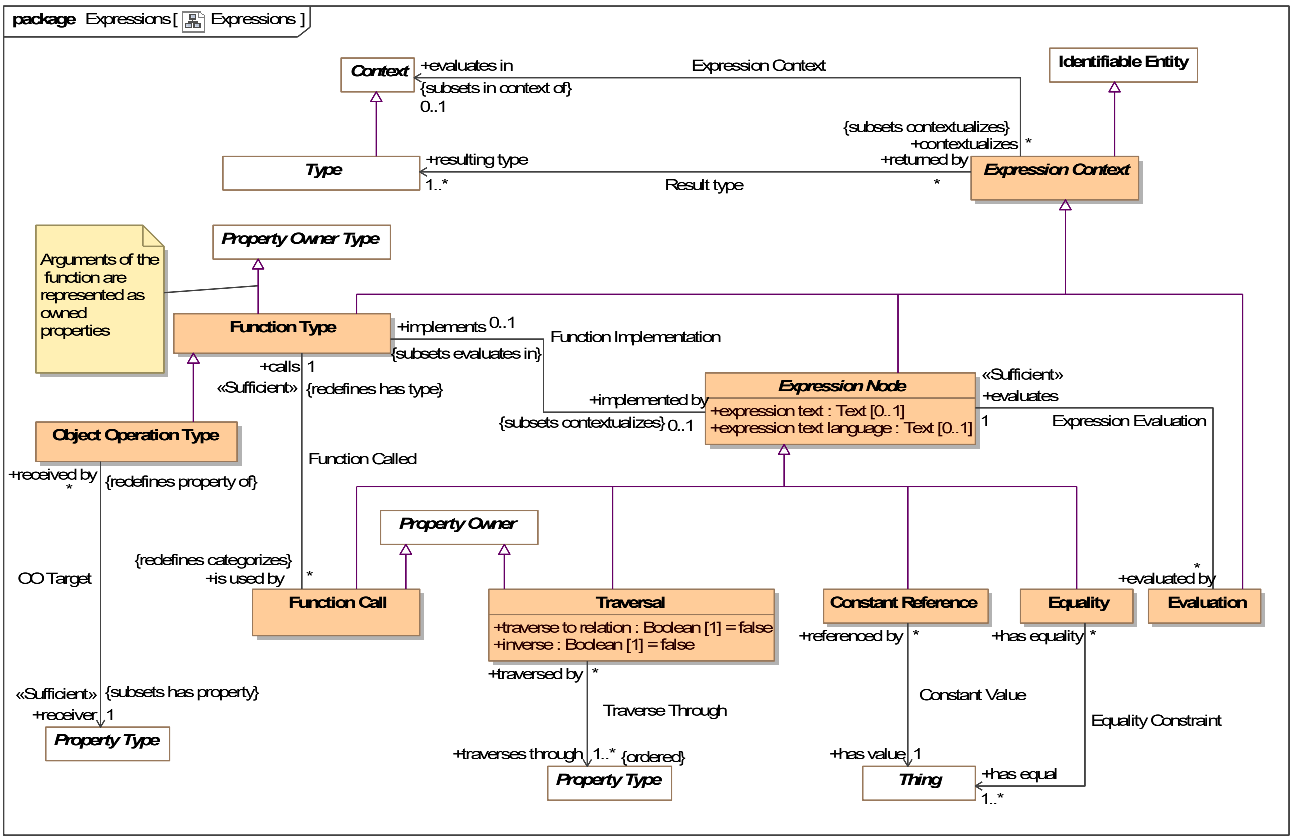
Associations

/ <<Restriction>> : Association *Redefines*: categorizes:Thing

8 SMIF Conceptual Model::Expressions

Expressions define computations across SMIF models.

8.1 Diagram: Expressions



h. Expressions

Expressions define computations

8.2 Class Constant Reference

A calculation that returns a thing identified by <has value>.

[FIBO] Constant
[FUML] LiteralSpecification where subtype of literal is determined by the type of <has value>.
-LiteralInteger->type is Integer or a subtype
-LiteralReal-> type is not integer or a subtype
-LiteralBoolean->type is Boolean
-LiteralString->type is Text

Direct Supertypes

[Expression Node]

Associations

↗ has value : [Thing] [1]
through association: [Constant Value]
A constant value referenced in an expression.

8.3 Association Constant Value

Relationship defining a link to a constant value within an expression.

Association Ends

↗ has value : [Thing] [1]
A constant value referenced in an expression.

↗ referenced by : [Constant Reference] [*]
Referencing constant expression node.

8.4 Class Equality

Returns TRUE if all <has equal> things have the same value or represent the same thing or set of things regardless of how they are represented.
Equality will return TRUE or FALSE.
[ISO11404: EqualityIn every value space there is a notion of equality, for which the following rules hold:
¯ for any two instances (a, b) of values from the value space, either a is equal to b, denoted a = b , or a is not equal to b, denoted a ≠ b ;
¯ there is no pair of instances (a, b) of values from the value space such that both a = b and a ≠ b ;
¯ for every value a from the value space, a = a ;
¯ for any two instances (a, b) of values from the value space, a = b if and only if b = a ;
¯ for any three instances (a, b, c) of values from the value space, if a = b and b = c , then a = c . On every datatype, the operation Equal is defined in terms of the equality property of the value space, by:
¯ for any values a, b drawn from the value space, Equal(a,b) is true if a = b , and false otherwise.

Direct Supertypes

[Expression Node]

Associations

↗ has equal : [Thing] [1..*]
through association: [Equality Constraint]
Set of things that must have the same value or represent the same thing or set of things for Equality to return true.

8.5 Association Equality Constraint

Relationship defining set of things that will be evaluated for equality.

Association Ends

has equal : Thing [1..*]

Set of things that must have the same value or represent the same thing or set of things for Equality to return true.

has equality : Equality [*]

Equality constraints for a thing.

8.6 Class Evaluation

The evaluation of an expression. All references to an evaluation shall return the result of evaluating the <evaluates> expression node. All expression nodes referenced within an evaluation shall return the result of evaluating that expression node.
An evaluation may be used in place of anything that requires the <resulting type> of the evaluation.

Direct Supertypes

Expression Context

Associations

evaluates : Expression Node [1]
through association: Expression Evaluation

The expression node "head" an evaluation evaluates.

8.7 Association Expression Context

Context in which an expression will be evaluated.

Direct Supertypes

Extent of Context

Association Ends

evaluates in : Context [0..1]

Context of evaluation and namespace resolution for an expression.

contextualizes : Expression Context [*]

Expressions referencing a context.

8.8 Class Expression Context

An abstract element defining the static or dynamic evaluation context and resulting type of an expression.
An expression context that is referenced by another expression context inherits the referencing context by default.

Direct Supertypes

Identifiable Entity

Associations

evaluates in : Context [0..1] *Subsets*: in context of:Context
through association: Expression Context

Context of evaluation and namespace resolution for an expression.

resulting type : Type [1..*]

through association: Result type

Type of the result of a function

[UML] type (of an operation or expression).

8.9 Association Expression Evaluation

Relationship defining the expression that will be evaluated by an evaluation.

Association Ends

evaluates : Expression Node [1]

The expression node "head" an evaluation evaluates.

evaluated by : Evaluation [*]

Evaluations of an expression node.

8.10 Class Expression Node

An abstract class representing the computation of a value which is then bound to the context from which it is called. Each expression node has a type of the most general type it can return.

An expression node may reference other elements. Where the other elements are also expression nodes they will be considered part of the referencing expression and evaluated in the context of that expression.

The set of related expression nodes forms a "tree" for evaluation.

[FIBO] Expression
[UML] Expression

Direct Supertypes

Expression Context

Attributes

expression text : Text [0..1]
Textual expression of the expression which is further refined by subtypes of expression.
[UML] StringExpression

expression text language : Text [0..1]
expression language used for the expression text

Associations

evaluated by : Evaluation [*]
through association: Expression Evaluation
Evaluations of an expression node.

8.11 Class Function Call

An element of an expression that performs some operation based on a function type and produces a result. I.e. plus(a,1). Arguments are bound to the function call via bindings.

Direct Supertypes

Expression Node, Property Owner

Associations

calls : Function Type [1] *Redefines*: has type:Type
through association: Function Called
Function called

8.12 Association Function Called

Relationship defining the function (a type) called by a function call.

Direct Supertypes

Extent of Type

Association Ends

calls : Function Type [1] *Redefines*: has type: Type
Function called

is used by : Function Call [*] *Redefines*: has type: Type
Function calls using a function declaration.

8.13 Association Function Implementation

Relationship defining the implementation of a function by an expression.

Direct Supertypes

Expression Context

Association Ends

implemented by : Expression Node [0..1] *Redefines*: has type: Type
Expression which defines the implementation of a function.

implements : Function Type [0..1] *Redefines*: has type: Type
Function implemented by an expression

8.14 Class Function Type

A declaration of a function which performs a calculation on arguments (properties) to produce a result (function result). I.e. the definition of plus(a:Number, b:Number).

Functions are intended to be side-effect free and context free (they only depend on their arguments and don't change anything) but assertions to specify that certain functions are pure may be required,
Note: FUNCTION ARGUMENTS ARE PROPERTIES of the function.

[FUML] Operation where ownedParameter corresponds with <has property> and type corresponds with <resulting type>.

Direct Supertypes

Expression Context, Property Owner Type

Associations

implemented by : Expression Node [0..1] *Subsets*: contextualizes:Thing *through association:* Function Implementation

Expression which defines the implementation of a function.

is used by : Function Call [*] *Redefines*: categorizes:Thing
through association: Function Called

Function calls using a function declaration.

8.15 Class Object Operation Type

An operation bound to a specific "receiver" in the "Object Oriented" sense.
[FUML] Operation

Direct Supertypes

Function Type

Associations

receiver : Property Type [1] *Subsets*: has property:Property Type
through association: OO Target

The property that is the receiver of an object operation.
[UML] class (of Operation)

8.16 Association OO Target

Relationship defining the "target" type of an object oriented function.

Association Ends

receiver : Property Type [1] *Subsets*: has property:Property Type

The property that is the receiver of an object operation.
[UML] class (of Operation)

received by : Object Operation Type [*] *Subsets*: has property:Property Type

The Object Operation for which a receiver is defined.

8.17 Association Result type

Relationship defining the type or types returned by an expression evaluation.

Association Ends

resulting type : Type [1..*] *Subsets*: has property:Property Type

Type of the result of a function
[UML] type (of an operation or expression).

returned by : Expression Context [*] *Subsets*: has property:Property Type

Method returning a type.

8.18 Class Traversal

Traversal from the current <evaluates in> context to another across a relation or other structure.

A traversal is a structure such that the structure's bindings may hold other properties of a traversal constant as independent variables where <traverses through> is the dependent variable. The traversal shall be considered to have the type of the relation it is traversing. Traversing binary relations does not require any bindings.

[OWL] ObjectPropertyChain

Direct Supertypes

Expression Node, Property Owner

Attributes

traverse to relation : Boolean [1] = false

Where traverse to relation is false, the traversal will return the bound element(s) of the <traverses through> property from the current context via any intermediate relationships.

Where traverse to relation is true, the traversal shall return the structure/situation/relationship owning the property binding.

By default, traverse to relation is false.

inverse : Boolean [1] = false

Indicates that the traversal is defined based on properties that reference the current context. This results in traversing "backwards" across a property to an inverse property or the relation.

Associations

traverses through : Property Type [1..*]

through association: Traverse Through

Property or properties through which a traversal traverses as the dependent variable(s).

8.19 Association Traverse Through

Relationship defining the property of the current context which will be traversed.

Association Ends

traverses through : Property Type [1..*]

Property or properties through which a traversal traverses as the dependent variable(s).

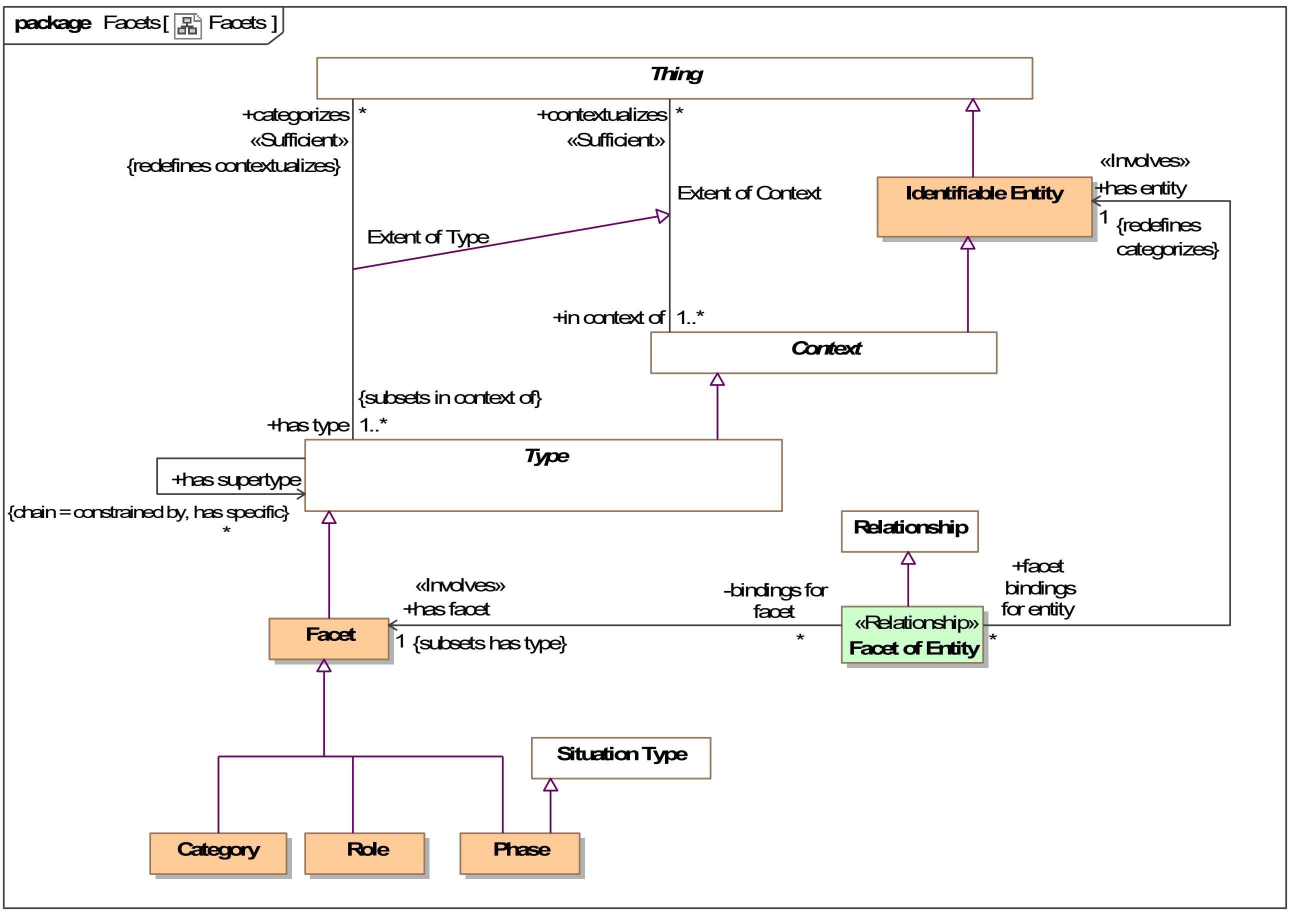
traversed by : Traversal [*]

Traversals through a property.

9 SMIF Conceptual Model::Facets

The facet package defines facets, roles and phases. Types that "mix in" to other types in a specific context or timeframe.

9.1 Diagram: Facets



i. Facets

9.2 Class Category

A category is a classification or division of people, events or things regarded as having particular shared characteristics. Categorization is typically contextual, potentially transient and may or may not be formally defined.
As with all facets, categories are non-rigid. Something classified by a category must also be classified by an entity type.

Direct Supertypes

Facet

9.3 Class Facet

A facet is a "mix in" type that defines an aspect of something but does not define the identity or "fundamental" (A.K.A. "Rigid") type of that thing, but some potentially transient role, phase or other way to classify it. Something must have at least one type that is not a facet to define that things identity.
Facets do not define independent identity of the referent but technology implementations may create independent objects to represent a facet.
An instance of a facet must also have a type that is not a facet to provide the identity of the instance.
The type(s) a facet may categorize may be constrained by a Facet Generalization Constraint. E.g. Policeman is a role of a person.

[Guarino1994] Non-Substantial sortal
[Guizzard] Non-Rigid Universal: A universal G is non-rigid iff for a w ∈W There is an x such that x ∈extw(G), and there is a w'∈W such that x ∉extw'(G)

[SOWA1999] Prehension (Relative

Direct Supertypes

Type

Associations

<<Sufficient>> : Facet Classification Constraint *Subsets*: has general:Type

9.4 Class Facet of Entity <<Relationship>>

Facet of entity is the binding of a particular entity to a facet. May also be considered an "as a" relationship. In the case of a role, it states that an entity plays the role, e.g. "Joe as a policeman". In the case of a phase, it states that an entity has that phase and that it is a phase of that entity, e.g. Sue as a teenager.
Facet of Entity is a kind of contextual categorization in that the entity assumes all of the characteristics of the facet where the Facet of Entity is asserted. E.g. if Joe has a policeman role, Joe is a policeman.

Facet of entity is an "Extent of Type" association reified as a relationship in that the binding of the entity to the facet may be valid in particular context or time frame. Facet of entity may be the consequence of a relationship. Note: Not represented as an association class due to OMG-MOF limitations.

Facet of entity may only relate entities that have a type compatible with the type of the facet, as defined by a Facet Classification Rule.

[FIBO] (for roles of actors) AgentInRole.
[FIBO] (for roles of anything else) ThingInRole
[Guarino1994] Externally Dependent Moment (Also called "Qua individual")
[SOWA1999] Prehension

Direct Supertypes

Relationship

Associations

has facet : Facet [1] *Subsets*: has type:Type

The facet that an entity assumes when it is the facet of an entity.
[FIBO] (for roles) playsRole

has entity : Identifiable Entity [1] *Redefines*: categorizes:Thing

The entity having a facet (including roles and phases).
[FIBO] (for roles) isPlayedBy

9.5 Class Phase

A phase (or state) is a static characteristic of something that exists for limited time(s). Something takes on or looses a phase as a result of some event. E,g, Teenager, living, closed invoice.
A Phase is a situation in that there is a situation coincident with each phase.

[Guizzardi] (Phased-Sortal): Let PS be a universal and let S be a substance sortal specialized (restricted by) PS. Now, let extw(~PS) = extw(S) \ extw(PS) be the complement of the extension of PS in world w. In this formula, the symbol \ represents the set theoretical operation of set difference. The universal PS is a phased-sortal iff for all worlds w ∈W, there is a w′ ∈W such that extw(PS) ∩ extw′(~PS) ≠ Ø

Direct Supertypes

Facet, Situation Type

9.6 Class Role

A role is a facet type that defines a specific purpose or behavior of a class of things. E.g. teacher, policeman, or employer.
[FIBO] Role. Note that partyInRole or thingInRole are implied by classification of a thing.

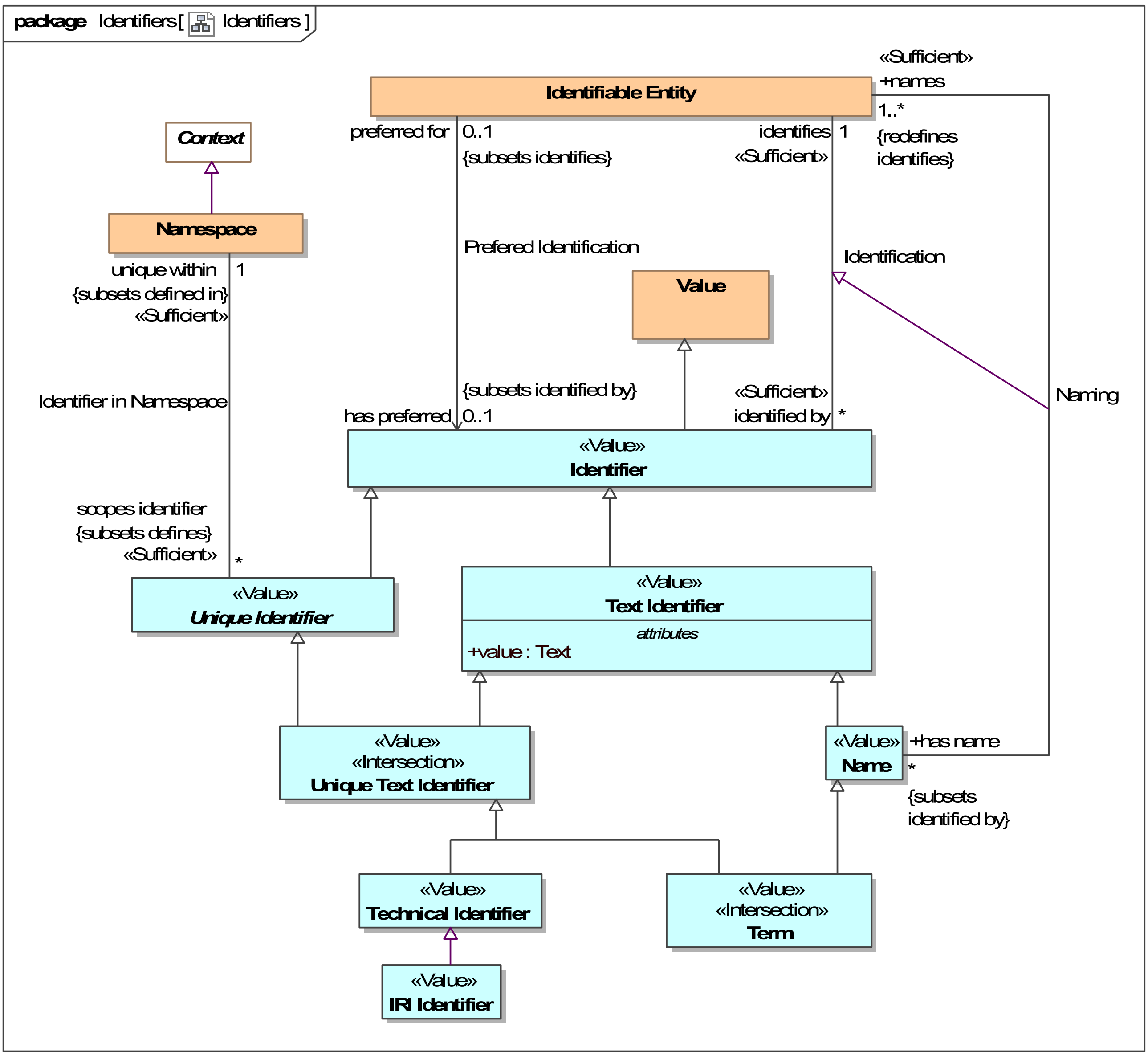
Direct Supertypes

Facet

10 SMIF Conceptual Model::Identifiers

Terms and identifiers provide for signs for (ways to identify) anything.

10.1 Diagram: Identifiers



j. Identifiers

An identifier that can be represented as text. The text is in the "value" property.

[IDEAS] Sign: An Individual that signifies a Thing.

10.2 Association Identification

Relationship defining an identifier for an entity.

[IDEAS] namedBy: A couple that asserts that a Name describes a Thing.

[ISO 1087] Designation

Association Ends

identifies : Identifiable Entity [1] *Redefines*: categorizes: Thing

The entity an identifier identifies.

[FIBO] identifies: is the relationship between something and that which provides a unique reference for it

[ISO 1087] designator: representation of a concept (3.2.1) by a sign which
denotes it

identified by : Identifier [*] *Redefines*: categorizes: Thing

An identifier for an <Entity>.

[FIBO] hasDenotation

10.3 Class Identifier <<Value>>

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

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

Direct Supertypes

Value

Associations

<<Sufficient>> identifies : Identifiable Entity [1]
through association: Identification

The entity an identifier identifies.

[FIBO] identifies: is the relationship between something and that which provides a unique reference for it

[ISO 1087] designator: representation of a concept (3.2.1) by a sign which
denotes it

10.4 Association Identifier in Namespace

Relationship defining the namespace within which a unique identifier is defined and unique.

[ISO 1087] monosemy: relation between designations (3.4.1) and concepts (3.2.1) in a given language in which one designation only relates to one concept

Direct Supertypes

Definition

Association Ends

unique within : Namespace [1]

The namespace in which an identifier is defined and has a unique value.

[FUML] memberNamespace

scopes identifier : Unique Identifier [*]

An Identifier defined within the scope of a namespace.

[FUML] member

10.5 Class IRI Identifier <<Value>>

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

[FIBO] anyURI

Direct Supertypes

Technical Identifier

10.6 Class Name <<Value>>

A word or set of words by which a person, animal, place, or thing is known, addressed, or referred to. Names are not necessarily unique.

[IDEAS] Name: A Representation that identifies a Thing.

[CL] Discourse Name

Direct Supertypes

Text Identifier

Associations

names : Identifiable Entity [1..*] *Redefines*: identifies:Identifiable Entity
through association: Naming

An entity named by a name.

10.7 Class Namespace

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

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

[FIBO] IdentificationScheme: system for allocating identifiers to objects

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

[FUML] Namespace

[CL] Vocabulary

Direct Supertypes

Context

Associations

<<Sufficient>> scopes identifier : Unique Identifier [*] *Subsets*: defines:Thing
through association: Identifier in Namespace

An Identifier defined within the scope of a namespace.
[FUML] member

10.8 Association Naming

Relationship defining a human meaningfully name for an entity.

Direct Supertypes

Identification

Association Ends

names : Identifiable Entity [1..*] *Subsets*: defines:Thing

An entity named by a name.

has name : Name [*] *Subsets*: defines:Thing

A human meaningful name for an entity.
[FIBO] hasName: that by which some thing is known; may apply to anything

[OWL] rdfs:label

10.9 Association Prefered Identification

Relationship defining the preferred identifier for an entity.

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

Direct Supertypes

Identification

Association Ends

has preferred : Identifier [0..1] *Subsets*: defines:Thing

Default identifier to use for an entity.
Where multiple identifiers are preferred in differing context any method for selecting the most preferred identifier is

implementation specific and not specified by this standard.
[FUML] NamedElement.name: Note: An Identifier that is <preferred for> an entity is equivalent to the name of a named element.

preferred for : Identifiable Entity [0..1] *Subsets*: defines:Thing
The entity an identifier is preferred for.

10.10 Class Technical Identifier <<Value>>

A technical identifier is defined within a technical system, information structure or system of systems for references and identity within that system or information element. Such identifiers may have no meaning outside of that system.

Typical technical identifiers include inter document "refs", record numbers, etc. The system should be referenced as the namespace.

Direct Supertypes

Unique Text Identifier

10.11 Class Term <<Value>><<Intersection>>

A word, phrase or name used by stakeholders to uniquely identify entities.

[ISO 1087] term: verbal designation of a general concept in a specific subject field.

Direct Supertypes

Name, Unique Text Identifier

10.12 Class Text Identifier <<Value>>

A code or other simple value that can be represented as text, identifying something that may or may not be unique.
Simple identifiers may be codes, names, numbers or compound values.
[NIEM] IdentificationType (IdentificationID=value)

Direct Supertypes

Identifier

Attributes

value : Text
Text value of an identifier

10.13 Class Unique Identifier <<Value>>

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

Direct Supertypes

Identifier

Associations

<<Sufficient>> unique within : Namespace [1] *Subsets*: defined in:Lexical Scope
through association: Identifier in Namespace

The namespace in which an identifier is defined and has a unique value.
[FUML] memberNamespace

10.14 Class Unique Text Identifier <<Value>><<Intersection>>

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

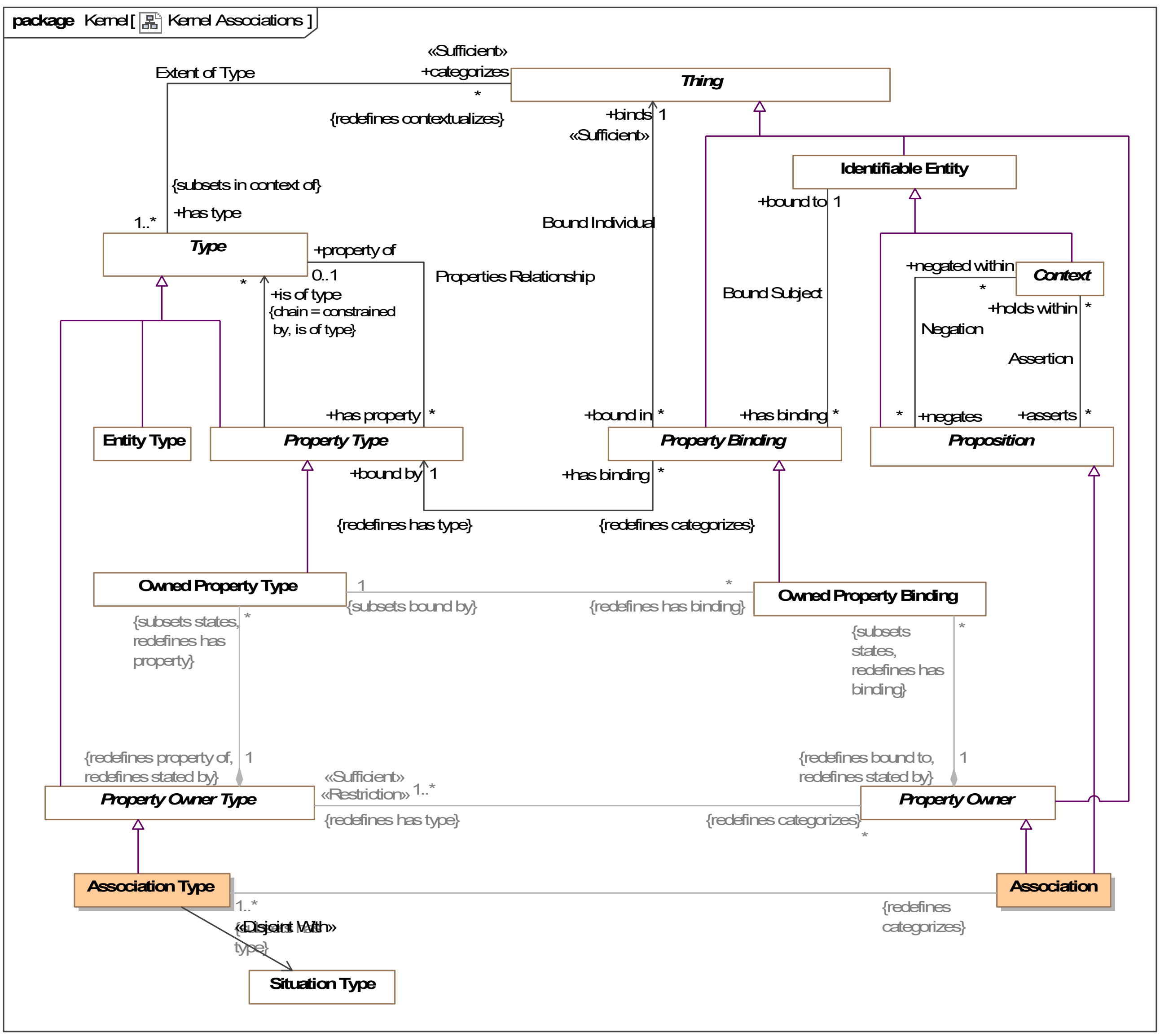
Direct Supertypes

Text Identifier, Unique Identifier

11 SMIF Conceptual Model::Kernel

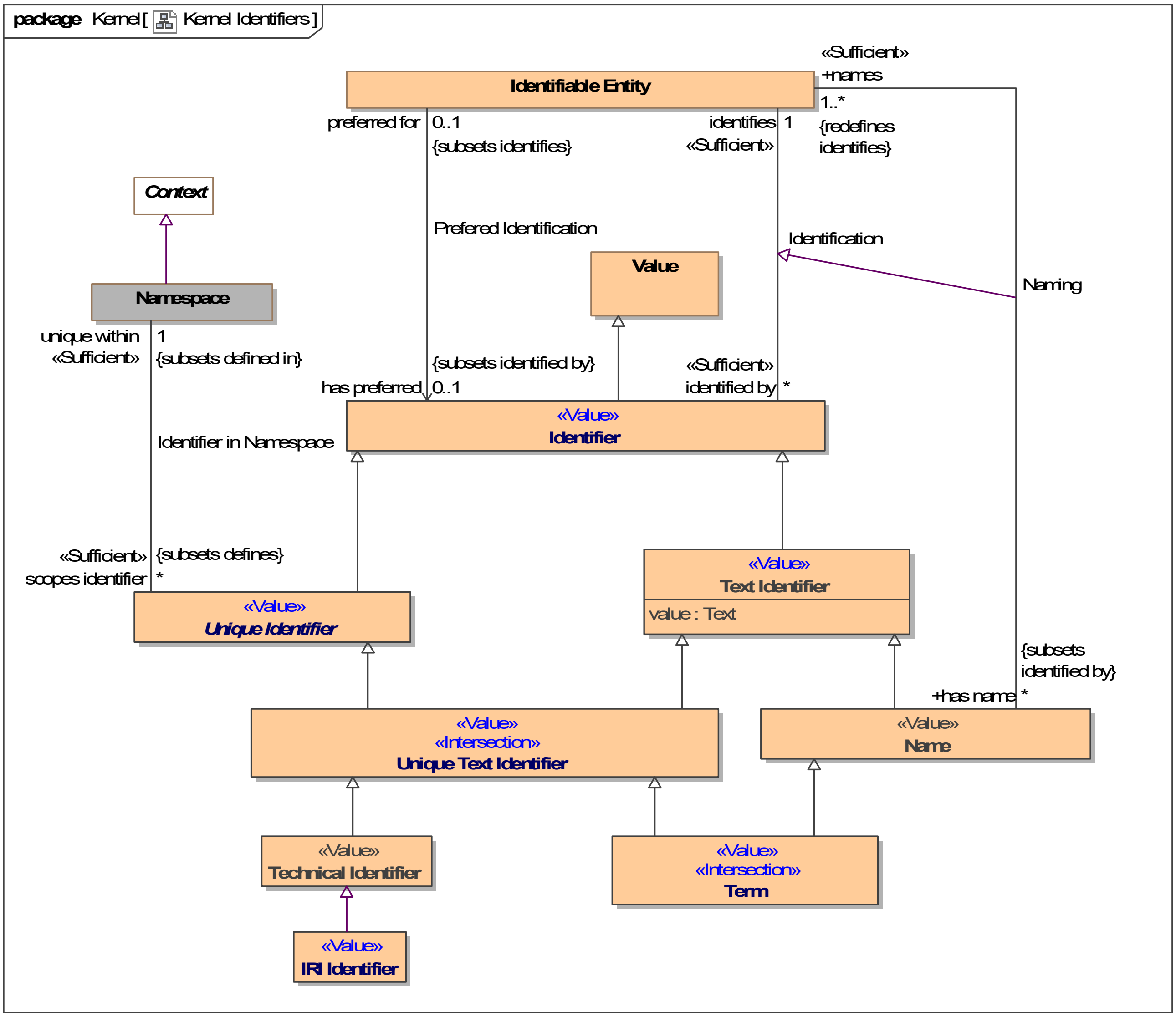
The kernel subsets the SMIF classes. The diagrams in this package illustrate the concrete classes that are used to define the SMIF language.
Note that shaded classes are not instantiated in the kernel and may be "flattened". Specifications for each class and association are defined in the corresponding package for that concept.

11.1 Diagram: Kernel Associations



k. Kernel Associations

11.2 Diagram: Kernel Identifiers

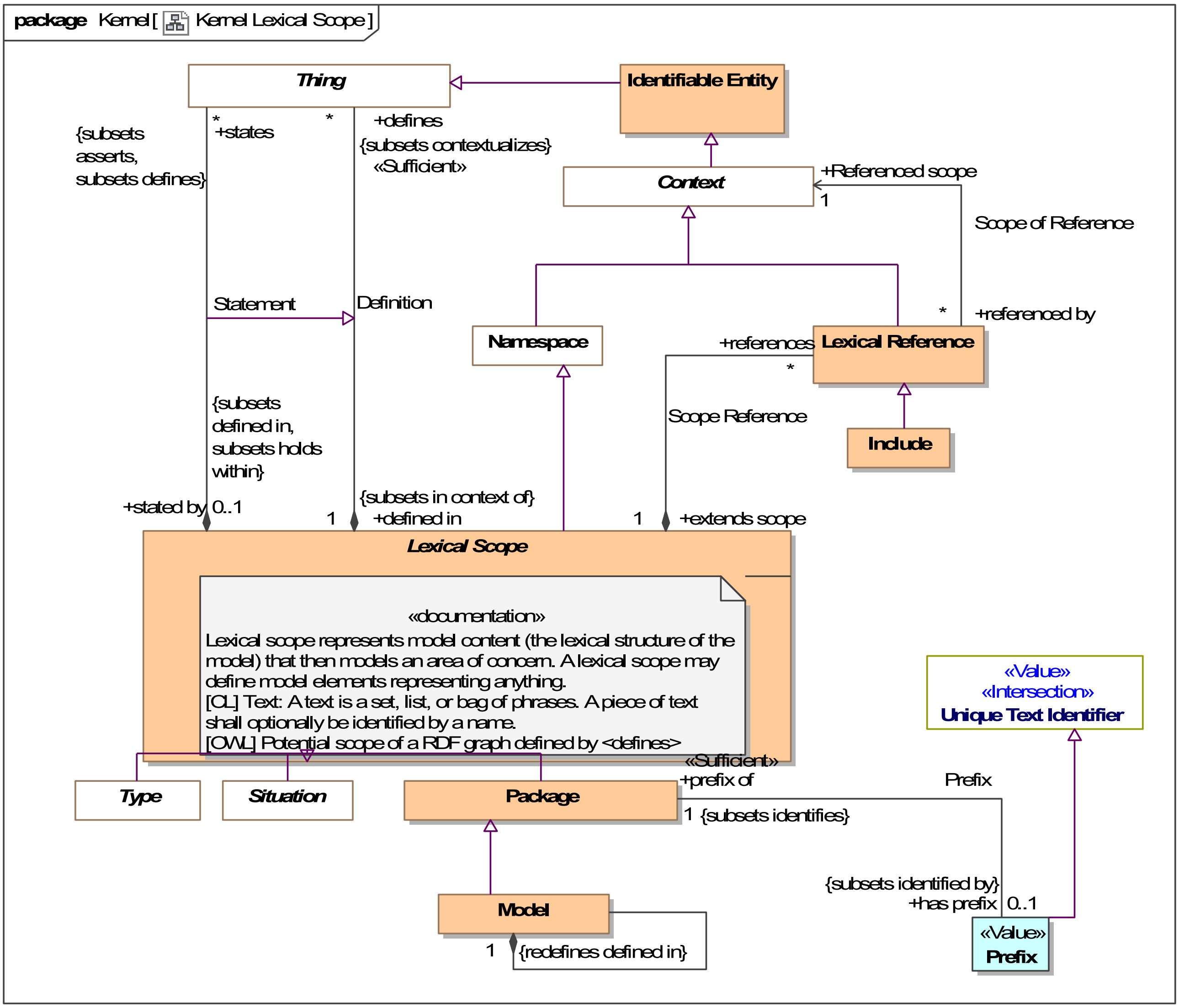


I. Kernel Identifiers

An identifier that can be represented as text. The text is in the "value" property.

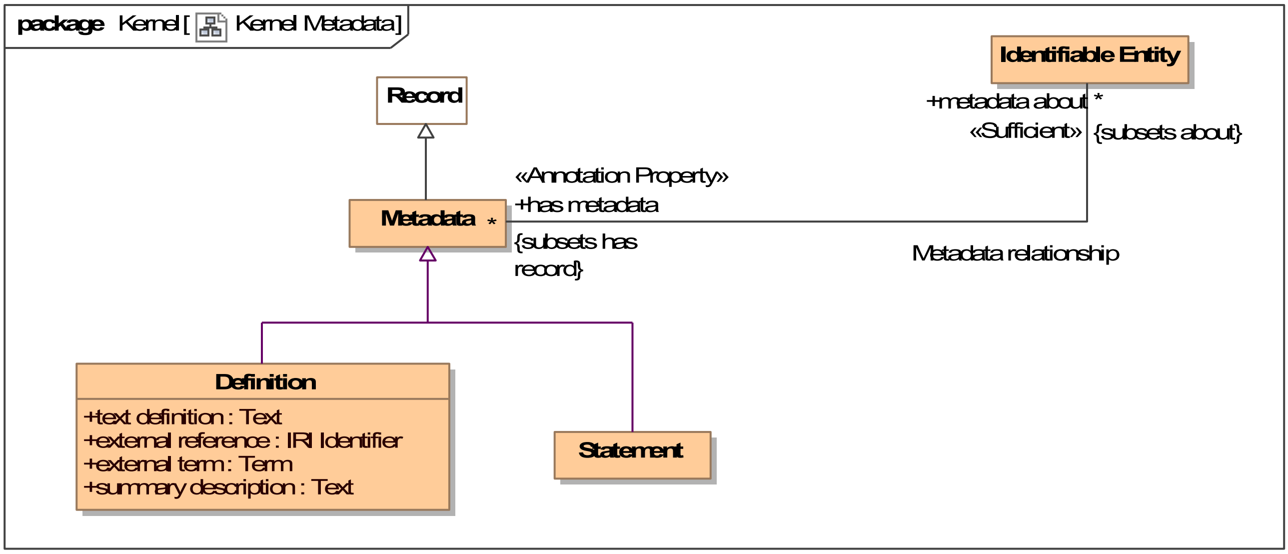
[IDEAS] Sign: An Individual that signifies a Thing.

11.3 Diagram: Kernel Lexical Scope



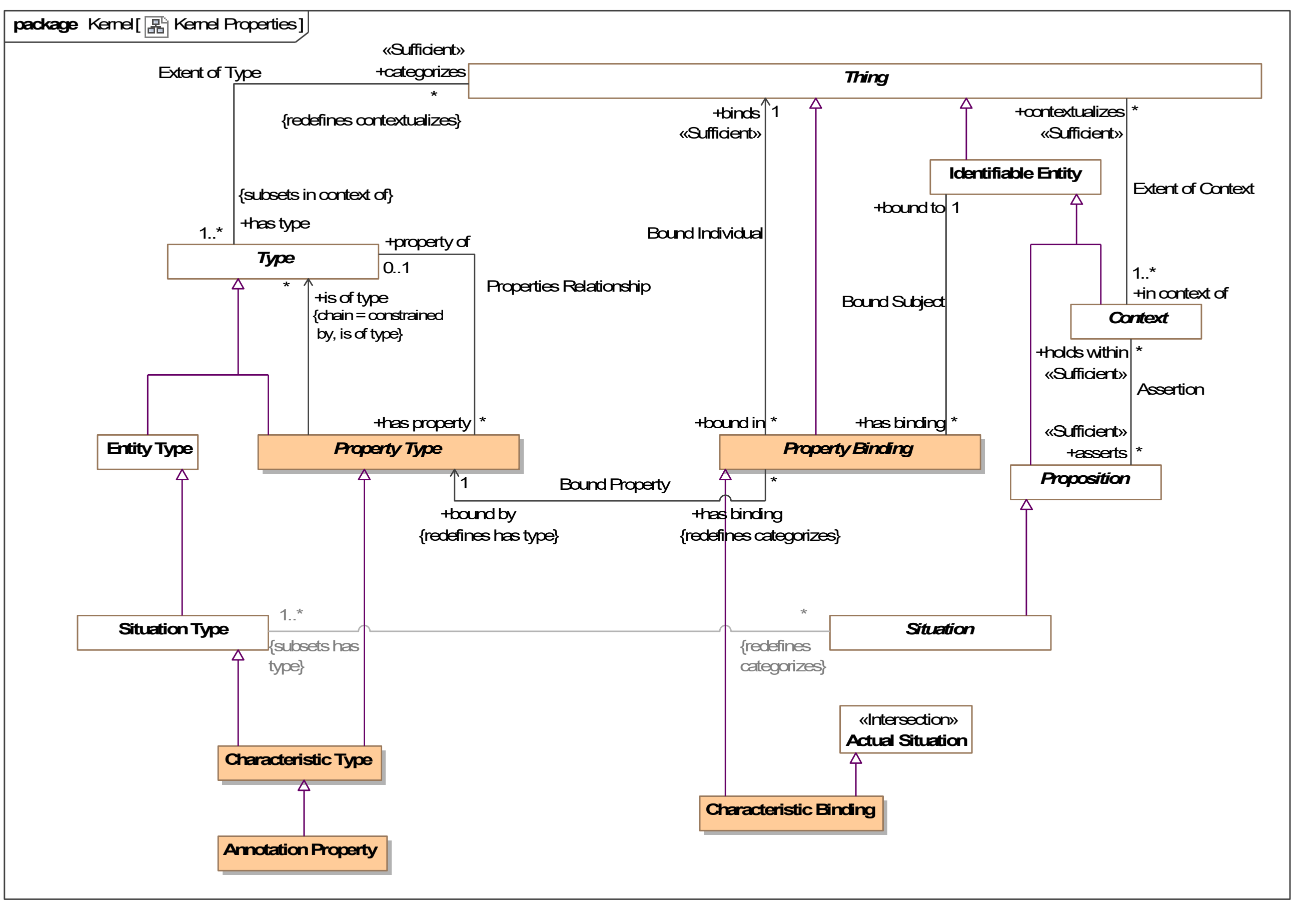
m. Kernel Lexical Scope

11.4 Diagram: Kernel Metadata



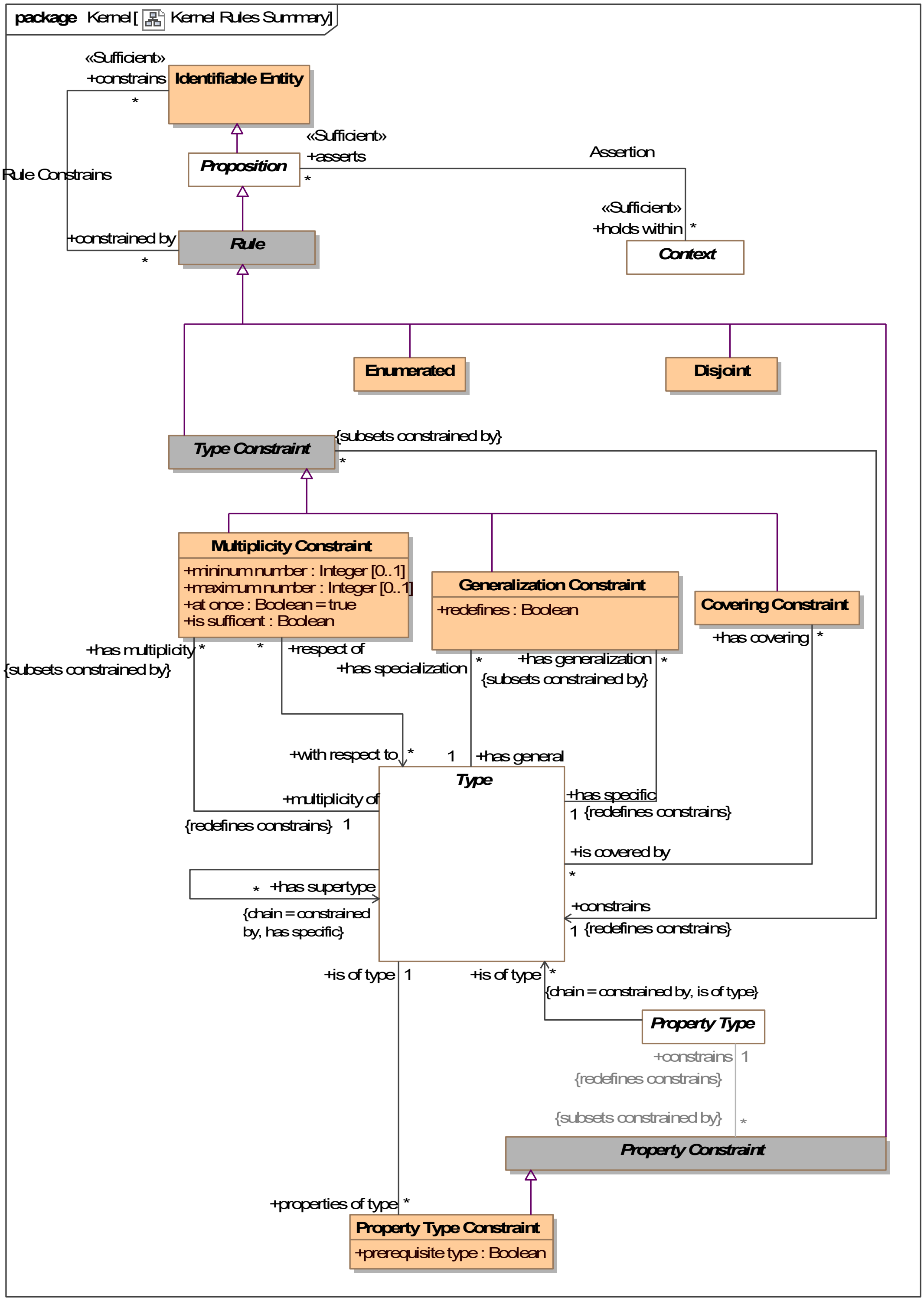
n. Kernel Metadata

11.5 Diagram: Kernel Properties



o. Kernel Properties

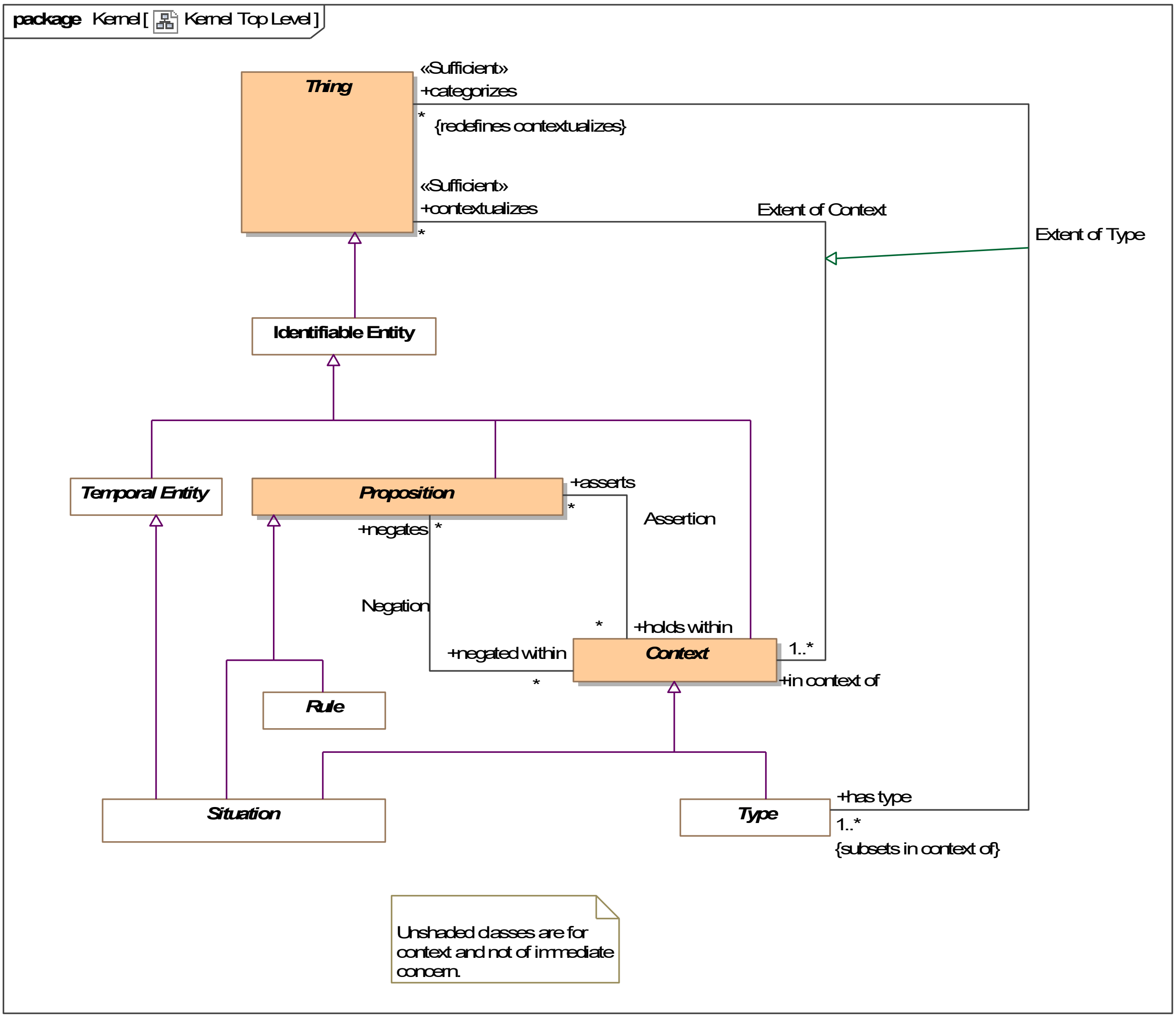
11.6 Diagram: Kernel Rules Summary



p. **Kernel Rules Summary**

This diagram shown a summary of the primary rules.

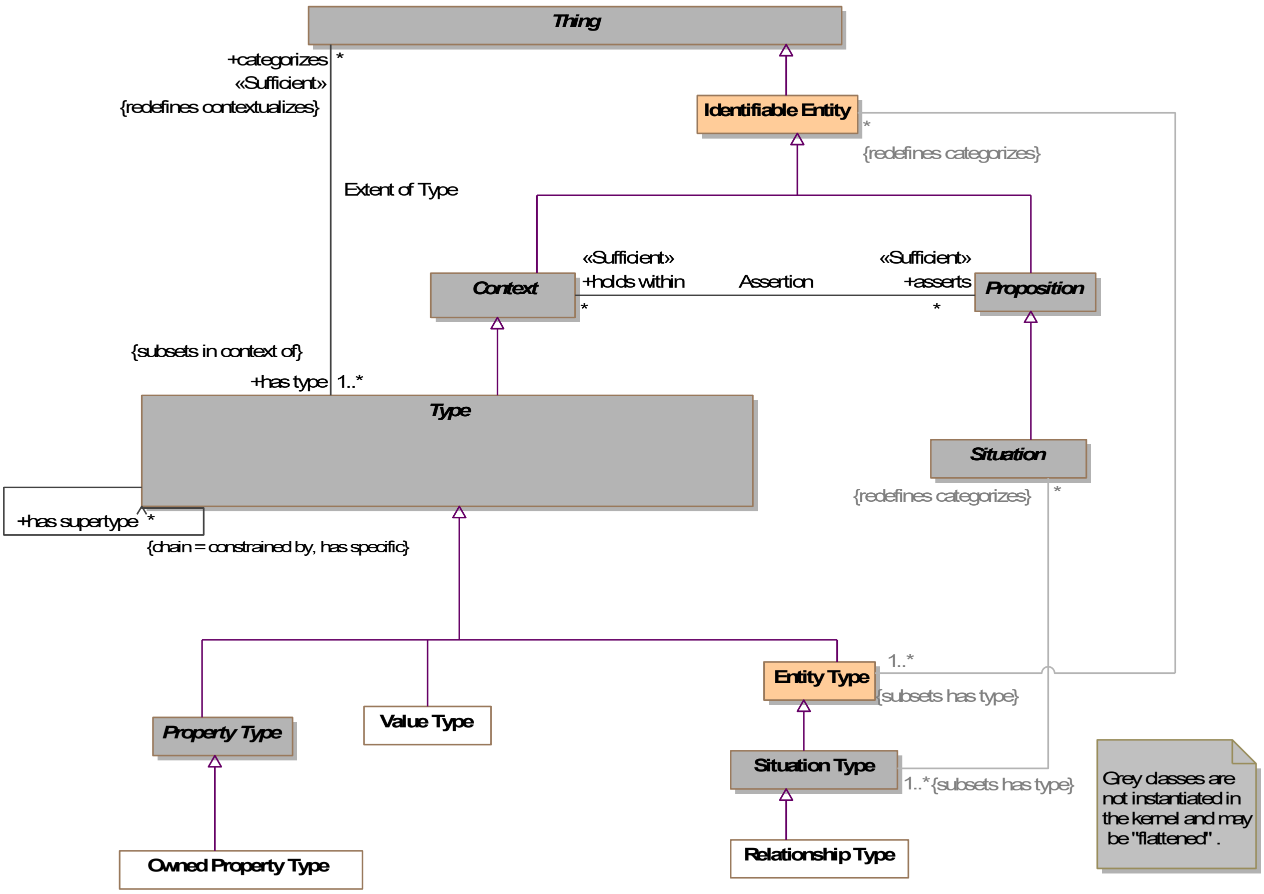
11.7 Diagram: Kernel Top Level



q. Kernel Top Level

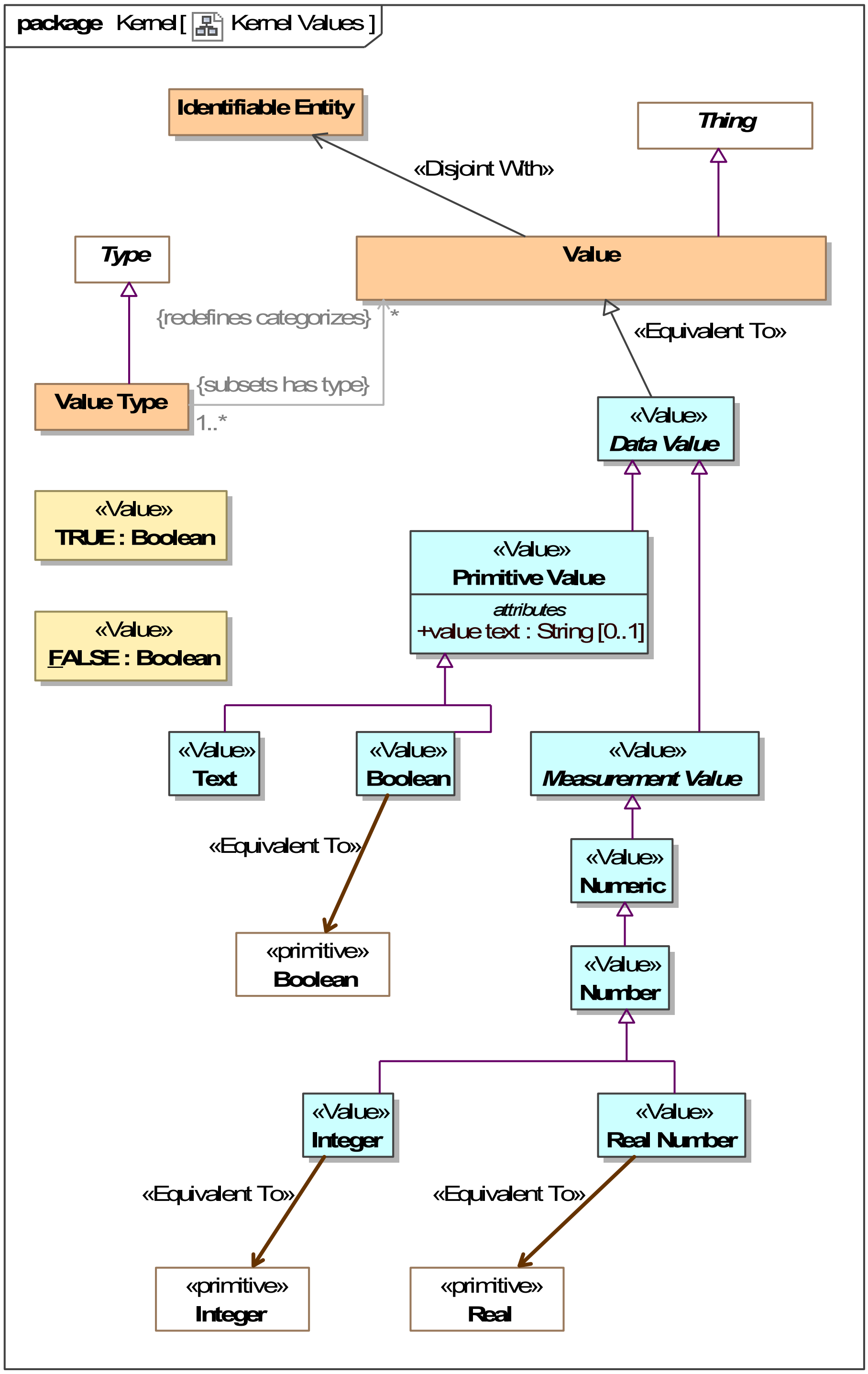
Diagram showing summary of top level classes and significant subtypes.

11.8 Diagram: Kernel Types



r. **Kernel Types**

11.9 Diagram: Kernel Values

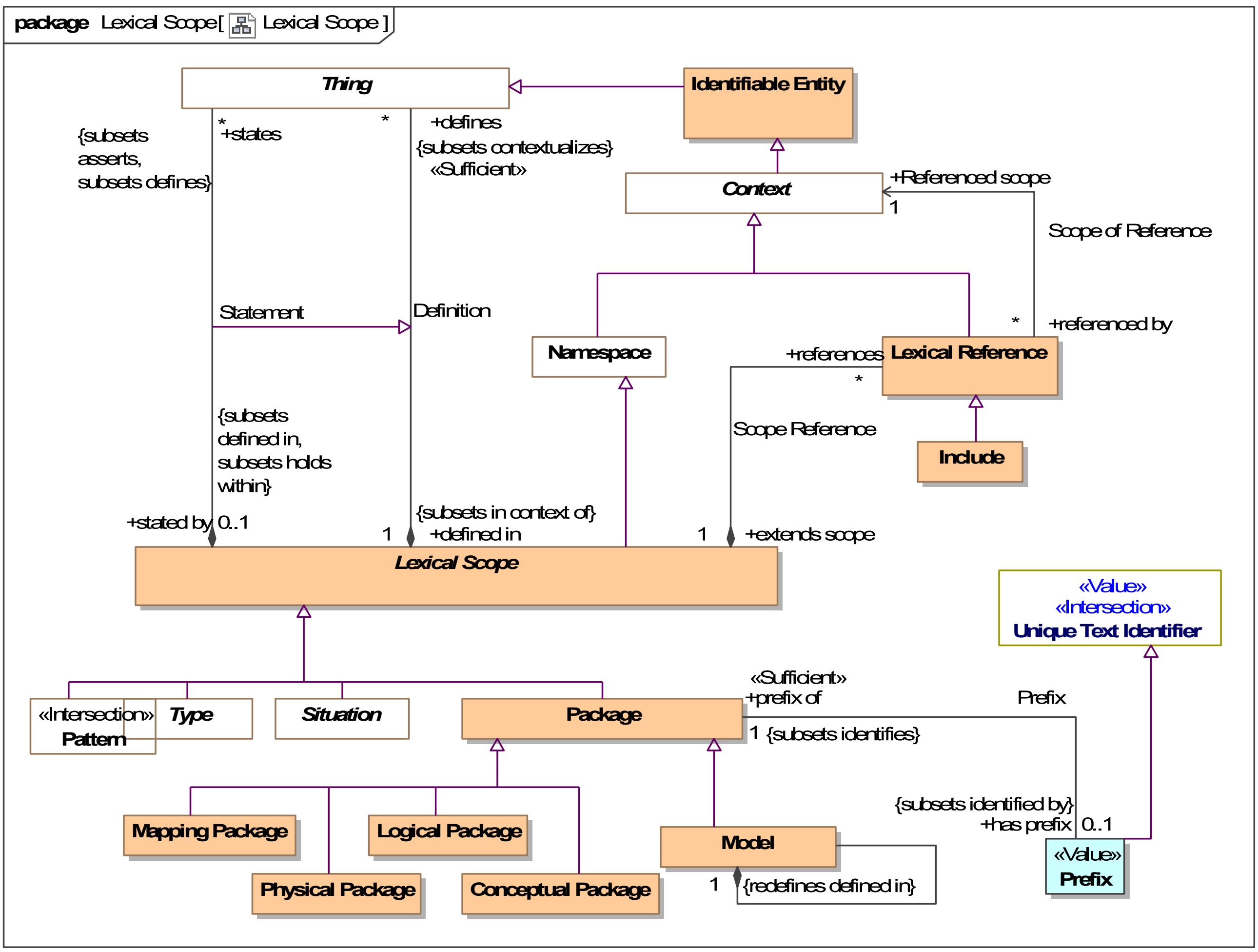


s. Kernel Values

12 SMIF Conceptual Model::Lexical Scope

Lexical scope defines the structure of models and the ownership of model elements.

12.1 Diagram: Lexical Scope



t. Lexical Scope

12.2 Class Conceptual Package

A model of a real or possible world as conceived by the model authors.

Direct Supertypes

Package

12.3 Association Definition

Relationship defining the set of elements defined within a lexical scope.
[OWL] RDF Graph

Direct Supertypes

Extent of Context

Association Ends

defines : Thing [*] *Subsets*: defined in:Lexical Scope

A model element defined within a lexical scope.

Definition within a scope does not assert everything within a scope but the lexical scope may be independently asserted, thus asserting what it defines.
[FUML] ownedElement, ownedMember

defined in : Lexical Scope [1] *Subsets*: defined in:Lexical Scope

Lexical scope defining model elements.

[UML]owner

12.4 Class Include

An "Include" is an external scope that is visible and asserted by the owning lexical scope.
[FUML] PackageImport
[CL] Importation: An importation contains a name. The intention is that the name identifies a piece of Common Logic content represented externally to the text, and the importation re-asserts that content in the text.

Direct Supertypes

Lexical Reference

12.5 Class Lexical Reference

A Lexical Reference is an external scope that is visible to but not necessarily asserted by the owning lexical scope.

Direct Supertypes

Context

Associations

Referenced scope : Context [1]
through association: Scope of Reference

A referenced context, potentially in another model, that provides visibility to the elements in that context.

[FUML] importedPackage
[OWL] directlyImports (implies "Include")

extends scope : Lexical Scope [1]
through association: Scope Reference

A lexical scope that is extended by a lexical reference.

[FUML] importingNamespace

12.6 Class Lexical Scope

Lexical scope represents model content (the lexical structure of the model) that then models an area of concern. A lexical scope may define model elements representing anything.
[CL] Text: A text is a set, list, or bag of phrases. A piece of text shall optionally be identified by a name.
[OWL] Potential scope of a RDF graph defined by <defines>

Direct Supertypes

Namespace

Associations

- defines : Thing [*]   *Subsets*: contextualizes:Thing
  through association: Definition

  A model element defined within a lexical scope.

Definition within a scope does not assert everything within a scope but the lexical scope may be independently asserted, thus asserting what it defines.
[FUML] ownedElement, ownedMember

- references : Lexical Reference [*]
  through association: Scope Reference

  A reference providing visibility of a lexical scope to an internal or external context.

- states : Thing [*]   *Subsets*: defines:Thing   asserts:Proposition
  through association: Statement

  <states> combines <defines> with <has assertion> to both define and assert an element within a lexical scope.

<states> provides a more "structural" organization of concepts that are both defined and asserted in the same structure.

<states> is a convenience for the common case where assertion and lexical containment are combined.

12.7 Class Logical Package

A model of information about systems independent of technical representation.

Direct Supertypes

Package

12.8 Class Mapping Package

A model defining relationships between other models.

Direct Supertypes

Package

12.9 Class Model

A root package. A model has no owner and may be directly referenced as an independent information resource. A model is defined in it's self.

Direct Supertypes

Package

Associations

: Thing [*] *Subsets*: defines:Thing asserts:Proposition

: Thing [*] *Subsets*: defines:Thing asserts:Proposition

12.10 Class Package

A model element that provides a definitional scope for other model elements. A package may be represented as a "graph".

[ISO 1087] concept system: system of concepts set of concepts (3.2.1) structured according to the relations among them

[FUML] Package. FUML ownedMember corresponds with SMIF <defines>. FUML "nestedPackage" corresponds with "defines" where the element defined is a package.

[CL] Module: A module consists of a name, an optional set of names called the exclusion set, and a text called the body text.

Direct Supertypes

Lexical Scope

Associations

<<Sufficient>> has prefix : Prefix [0..1] *Subsets*: identified by:Identifier
through association: Prefix
An abbreviation that can be used to identify a package.

12.11 Class Physical Package

A physical, technology specific, data schema representing information about a real or possible world.

Direct Supertypes

Package

12.12 Association Prefix

Relationship defining the prefix for a package.

Direct Supertypes

Identification

Association Ends

has prefix : Prefix [0..1] *Subsets*: identified by:Identifier
An abbreviation that can be used to identify a package.

prefix of : Package [1] *Subsets*: identified by:Identifier
An abbreviation for a package.

12.13 Class Prefix <<Value>>

A technical abbreviation for a package.

Direct Supertypes

Unique Text Identifier

Associations

<<Sufficient>> prefix of : Package [1] *Subsets*: identifies:Identifiable Entity
through association: Prefix
An abbreviation for a package.

12.14 Association Scope of Reference

Relationship defining internal or external context that are referenced by a lexical scope using a lexical reference.

Association Ends

Referenced scope : Context [1] *Subsets*: identifies:Identifiable Entity

A referenced context, potentially in another model, that provides visibility to the elements in that context.
[FUML] importedPackage
[OWL] directlyImports (implies "Include")

referenced by : Lexical Reference [*] *Subsets*: identifies:Identifiable Entity

References to a context.

12.15 Association Scope Reference

Relationship defining references for a scope.

Association Ends

references : Lexical Reference [*] *Subsets*: identifies:Identifiable Entity

A reference providing visibility of a lexical scope to an internal or external context.

extends scope : Lexical Scope [1] *Subsets*: identifies:Identifiable Entity

A lexical scope that is extended by a lexical reference.
[FUML] importingNamespace

12.16 Association Statement

Relationship defining the set of elements defined within and asserted by a lexical scope.

Direct Supertypes

Definition

Association Ends

states : Thing [*] *Subsets*: identifies:Identifiable Entity

<states> combines <defines> with <has assertion> to both define and assert an element within a lexical scope.
<states> provides a more "structural" organization of concepts that are both defined and asserted in the same structure.

<states> is a convenience for the common case where assertion and lexical containment are combined.

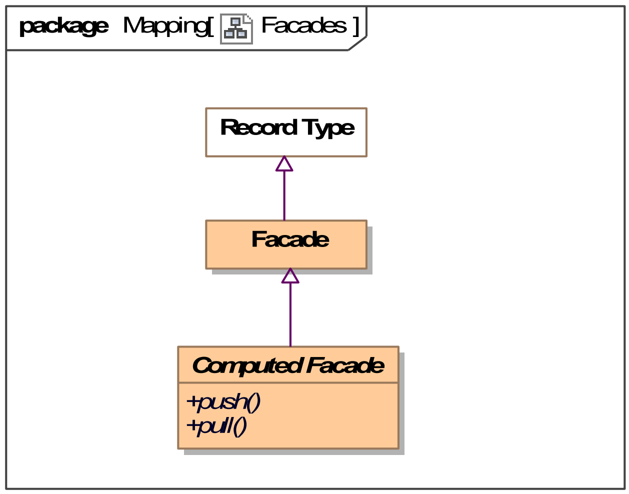
stated by : Lexical Scope [0..1] *Subsets*: identifies:Identifiable Entity

<stated by> is a lexical scope that both defines and asserts a model element.

13 SMIF Conceptual Model::Mapping

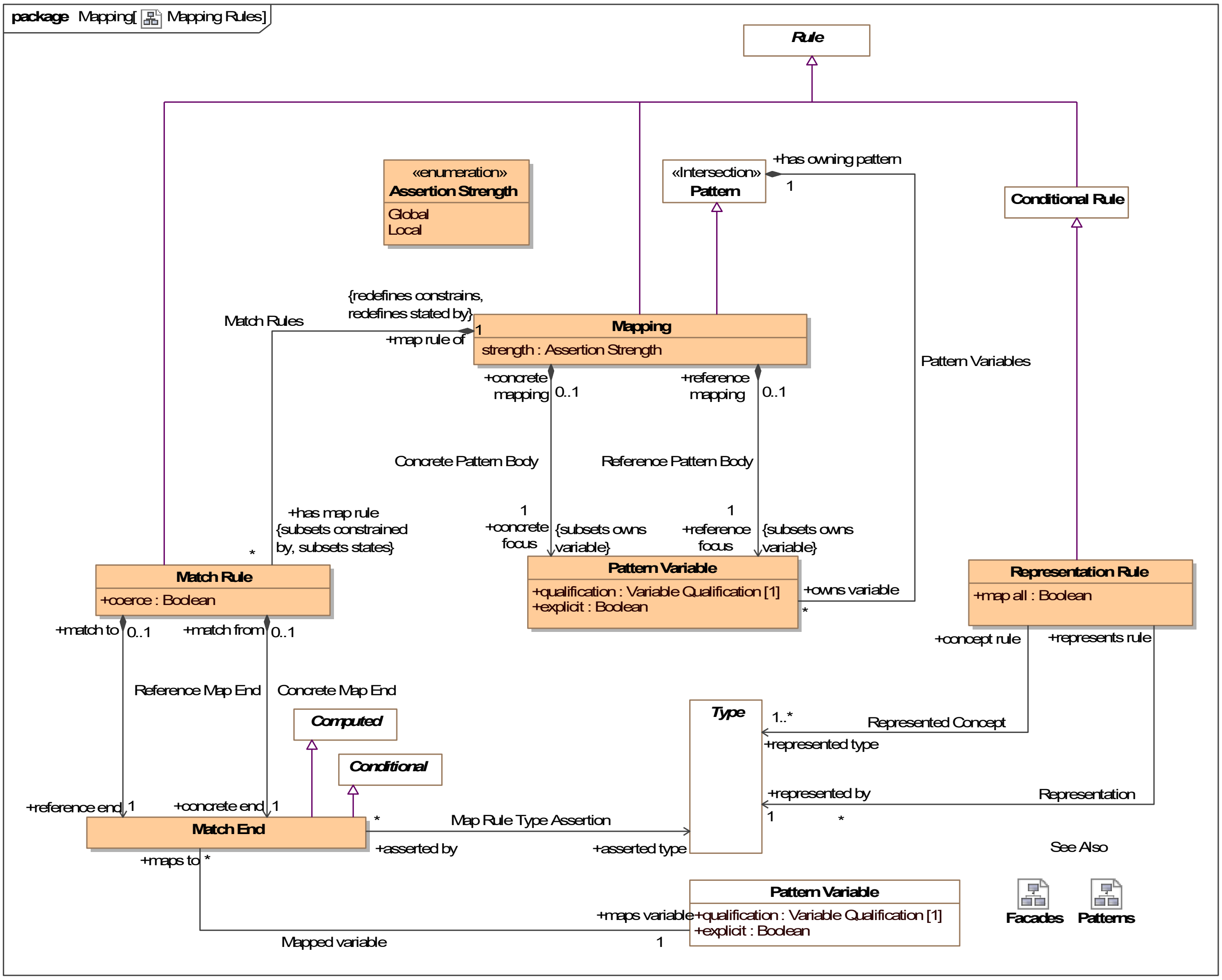
Mapping rules define how data represents concepts or how different data representations are related.

13.1 Diagram: Facades



u. Facades

13.2 Diagram: Mapping Rules



v. **Mapping Rules**

13.3 Class Computed Facade

A facade that is computed by calling external methods.

Direct Supertypes

[Facade]

Operations

- public push ()

An operation called to evoke the behavior associated with a new facade element being created or modified. Push asserts the more concrete type based on a reference type.

- public pull ()

An operation called to evoke the behavior associated with a facade representing existing elements. Pull asserts the reference type based on a more concrete type.

13.4 Association Concrete Map End

Relationship to the more concrete end of a match rule.

Association Ends

concrete end : Match End [1] *Subsets*: identifies:Identifiable Entity

One end of a mapping, to be used for more concrete end.

match from : Match Rule [0..1] *Subsets*: identifies:Identifiable Entity

Mapping rule owning a "concrete" end.

13.5 Association Concrete Pattern Body

Relationship between a mapping and a pattern of the more concrete concepts to be mapped.

Association Ends

concrete focus : Pattern Variable [1] *Subsets*: identifies:Identifiable Entity

The variable or variables that form the basis for the portion of the pattern for the more concrete (physical) model. The concrete portion of the pattern is derived from the transitive closure of all variables reachable from the pattern variable via characteristics, associations or relationships.
When a pattern matching the set of concrete variables is created or altered the mapping "fires" and the reference pattern is asserted.
The qualification of the referenced variable is constrained to be ""select".

concrete mapping : Mapping [0..1] *Subsets*: identifies:Identifiable Entity

Mapping for which a more concrete pattern is defined.

13.6 Class Facade

An intermediary data type used to hold common mappings. Facades may be computed and/or have mapping rules.

Direct Supertypes

Record Type

13.7 Association Map Rule Type Assertion

Relationship defining more concrete types that shall be asserted for an end of a match rule.

Association Ends

asserted type : Type [*] *Subsets*: identifies:Identifiable Entity

Type that will be asserted for the end that is more concrete than the defined type of a property or relationship. e.g. a unit type.

asserted by : Match End [*] *Subsets*: identifies:Identifiable Entity

Map rule and that asserts a type

13.8 Association Mapped variable

Relationship defining the property that is the source or target of a mapping

Association Ends

maps variable : Pattern Variable [1] *Subsets*: identifies:Identifiable Entity

Variable that defines a set of elements to map to the other side of the mapping rule. The set of elements shall be those bound to the property on evaluation of the mapping.

maps to : Match End [*] *Subsets*: identifies:Identifiable Entity

Map rule end for a property

13.9 Class Mapping

A mapping is a rule based on a pattern that defines how different representations of the same things correspond. There are two "sub patterns", defined by the concrete and reference variables and other variables reachable from them via characteristics, associations and relationships. These sub-patterns are matched (made to correspond) using "Match Rules"
Patterns define a set of related elements to be mapped based on two distinguished variables, the "concrete body" and the "reference body".
Types in a "concrete" body may be defined to be a representation (data about) a concept in a "reference" pattern.
Match rules define how elements in each of the sub-patterns are mapped, bidirectionally.
A mapping utilizing more specific types subsumes maps for more general types.
Note that the roles of "concrete" and "reference" may or may not reflect different levels of abstraction and in some cases the choice may be arbitrary.

Direct Supertypes

Pattern, Rule

Attributes

strength : Assertion Strength
Strength defines what will cause a rule to be considered for being asserted (firing).

Associations

concrete focus : Pattern Variable [1] *Subsets*: owns variable:Pattern Variable
through association: Concrete Pattern Body

The variable or variables that form the basis for the portion of the pattern for the more concrete (physical) model. The concrete portion of the pattern is derived from the transitive closure of all variables reachable from the pattern variable via characteristics, associations or relationships.
When a pattern matching the set of concrete variables is created or altered the mapping "fires" and the reference pattern is asserted.
The qualification of the referenced variable is constrained to be ""select".

has map rule : Match Rule [*] *Subsets*: constrained by:Rule states:Thing
through association: Match Rules

Map rule that is asserted by a mapping.

reference focus : Pattern Variable [1] *Subsets*: owns variable:Pattern Variable
through association: Reference Pattern Body

The variable or variables that form the basis for the portion of the pattern for the more abstract/reference (conceptual) model. The reference portion of the pattern is derived from the transitive closure of all variables reachable

from the pattern variable via characteristics, associations or relationships.
When a pattern matching the set of reference variables is created or altered the mapping "fires" and the concrete pattern is asserted.
The qualification of the referenced variable is constrained to be ""select".

13.10 Class Match End

One end of a mapping from one thing to another that may be qualified with a condition.
The set of elements to be mapped is the union of the sets of all mapped types and mapped variables that conform to the condition.
Match rules are constrained to apply to only conforming types or types that represent the mapped ends (as specified by a representation rule).
Representation rules applied to a supertype apply to a subtype unless a more specific representation rule is specified for the corresponding types.

Direct Supertypes

Computed, Conditional

Associations

asserted type : Type [*]
through association: Map Rule Type Assertion

Type that will be asserted for the end that is more concrete than the defined type of a property or relationship. e.g. a unit type.

maps variable : Pattern Variable [1]
through association: Mapped variable

Variable that defines a set of elements to map to the other side of the mapping rule. The set of elements shall be those bound to the property on evaluation of the mapping.

13.11 Class Match Rule

A rule that the 2 ends represent the same things or information about a thing.
Redundant mappings are ignored and identity is preserved across all mappings.

Direct Supertypes

Rule

Attributes

coerce : Boolean
Where <coerce> has a value of TRUE a map rule will be evaluated even if the <reference end> is not type compatible with the <concrete end> type.
Where <coerce> is FALSE or unstated a map rule will be evaluated only if the <reference end> is type compatible with the <concrete end> type.
Type compatible shall be defined as one of: Being the same type, <concrete end> being a subtype of <reference end> (as defined by a type generalization rule), <concrete end> being a representation of <reference end> (as defined by a representation rule).
Representation rules applied to a supertype apply to a subtype.

Associations

concrete end : Match End [1]
through association: Concrete Map End
One end of a mapping, to be used for more concrete end.

reference end : Match End [1]
through association: Reference Map End
One end of a match rule, to be used for more abstract end.

map rule of : Mapping [1] *Redefines*: constrains:Identifiable Entity stated by:Lexical Scope
through association: Match Rules
Mapping containing a map rule.

13.12 Association Reference Map End

Relationship to the reference end of a match rule.

Association Ends

reference end : Match End [1] *Redefines*: constrains: Identifiable Entity stated by: Lexical Scope
One end of a match rule, to be used for more abstract end.

match to : Match Rule [0..1] *Redefines*: constrains: Identifiable Entity stated by: Lexical Scope
Mapping rule owning a reference" end.

13.13 Association Reference Pattern Body

Relationship between a mapping and a pattern of the more abstract concepts to be mapped.

Association Ends

reference focus : Pattern Variable [1] *Redefines*: constrains: Identifiable Entity stated by: Lexical Scope
The variable or variables that form the basis for the portion of the pattern for the more abstract/reference (conceptual) model. The reference portion of the pattern is derived from the transitive closure of all variables reachable from the pattern variable via characteristics, associations or relationships.
When a pattern matching the set of reference variables is created or altered the mapping "fires" and the concrete pattern is asserted.
The qualification of the referenced variable is constrained to be ""select".

reference mapping : Mapping [0..1] *Redefines*: constrains: Identifiable Entity stated by: Lexical Scope
Mapping for which a more abstract pattern is defined.

13.14 Association Representation

More concrete type that represents information about the represented concept of a representation rule.

Association Ends

represented by : Type [1] *Redefines*: constrains: Identifiable Entity stated by: Lexical Scope
The representation of a concept in a more specific form

represents rule : Representation Rule *Redefines*: constrains: Identifiable Entity stated by: Lexical Scope

Rule defining a representation of a type.

13.15 Class Representation Rule

A representation rule states that the <represented type> has a representation defined by the <represented by> type. Representation rules are used to filter Map Rules such that only represented concepts may be mapped.
A representation is usually complimented with one or more mapping rules.

Direct Supertypes

Conditional Rule

Attributes

map all : Boolean

Specifies a direct mapping between instances of the types in both directions.
<map all> is equivalent to a mapping with a rule mapping properties of each type but is lower precedence than other mappings - if types have a more specific map it will apply first.

Associations

represented by : Type [1]
through association: Representation

The representation of a concept in a more specific form

represented type : Type [1..*]
through association: Represented Concept

A more general or abstract concept that is being represented.

13.16 Association Represented Concept

More abstract type that is <represented by> a more concrete type of a representation rule.

Association Ends

represented type : Type [1..*]

A more general or abstract concept that is being represented.

concept rule : Representation Rule

Rule defining a concept that is represented by another, more concrete, concept.

13.16.1 Enumeration Assertion Strength

Rule strength defines what will cause a rule to be considered for being asserted (firing).

```
package SMIF Conceptual Model::Mapping
public enum Assertion Strength
{Global, Local}
```

Literals

Global

The rule will be in effect globally.

Local

The rule will only be in effect if required to fulfill another rule.

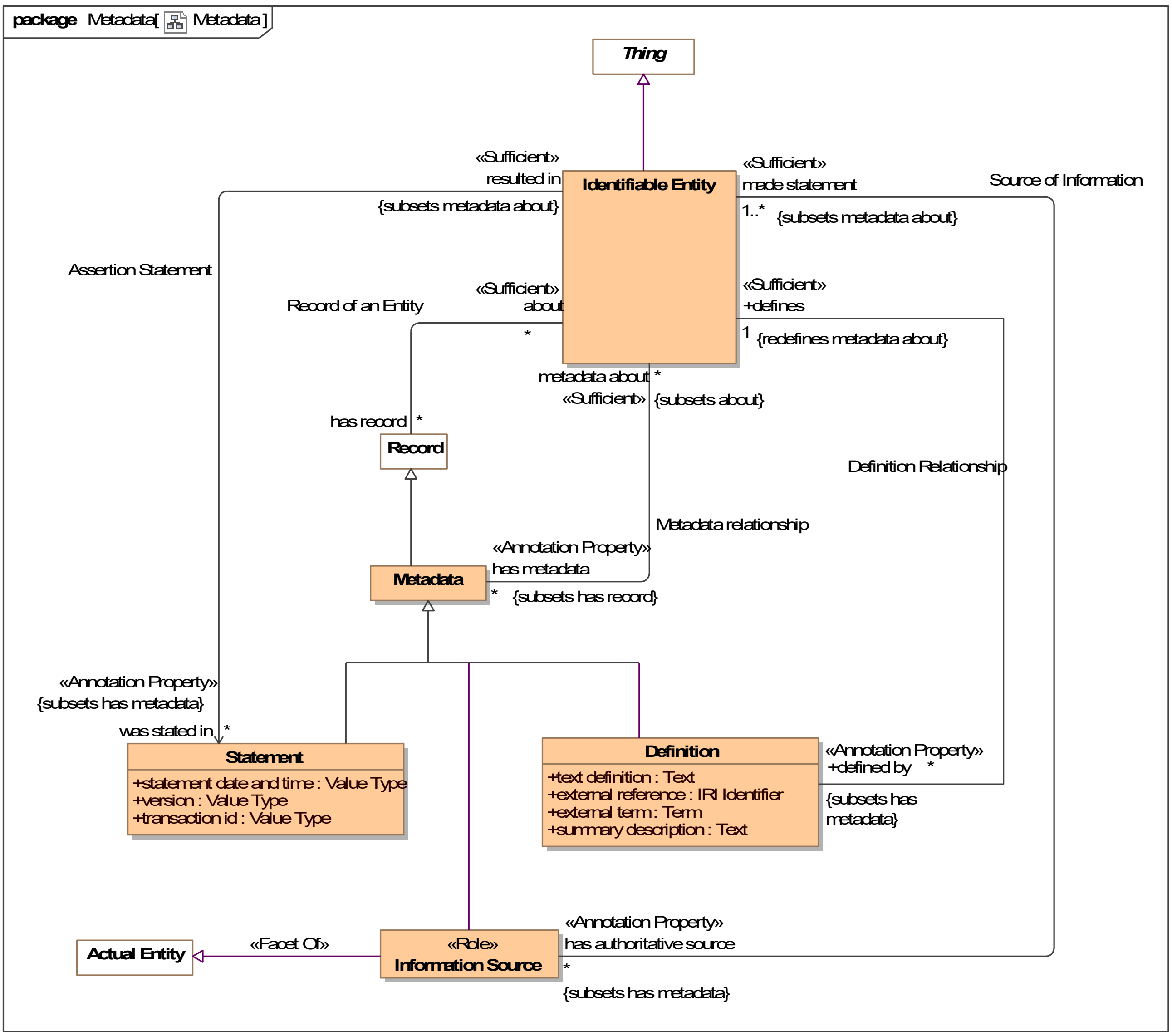
Known other enumerations

Enumeration Assertion Strength

14 SMIF Conceptual Model::Metadata

Metadata defines data about model elements (their source, definition or trust), which can be differentiated from model elements about the subject domain.

14.1 Diagram: Metadata



w. Metadata

14.2 Association Assertion Statement

Relationship defining the original statement, speech act or information artifact that asserted something in a model.

Direct Supertypes

Metadata relationship

Association Ends

was stated in : Statement [*]

Metadata representing the speech act, document or other record where a statement captured in a model was made.
[OWL] rdfs:isDefinedBy

resulted in : Identifiable Entity

Statement made in a statement by an information source.

14.3 Class Definition

An informal or natural language definition of a something and potentially a reference to external definitions.
A Definition may be in the context of a natural language to scope the language it is expressed in.

[ISO 1087] definition: representation of a concept (3.2.1) by a descriptive statement which serves to differentiate it from related concepts

[FUML] Comment (where body corresponds with "text definition").

Direct Supertypes

Metadata

Attributes

text definition : Text
Text describing a something in natural language. The language may be indicated by a context of the definition.
[OWL] rdfs:comment

external reference : IRI Identifier
A reference to an external information resource that further defines something.
[FIBO] ReferenceDOcument

external term : Term
Specific term in an external resource that further defines something.

summary description : Text
A short description of something.

Associations

<<Annotation Property>> defines : Identifiable Entity [1] *Redefines*: metadata about:Identifiable Entity
through association: Definition Relationship

Some thing described by a definition.
[FIBO] defines
[FUML]annotatedElement

14.4 Association Definition Relationship

Relationship between a thing and its definitions.

Direct Supertypes

Metadata relationship

Association Ends

defines : Identifiable Entity [1] *Redefines*: metadata about: Identifiable Entity

Some thing described by a definition.
[FIBO] defines
[FUML]annotatedElement

defined by : Definition [*] *Redefines*: metadata about: Identifiable Entity

An informal description of something.
[FIBO] hasDefinition
[UML] comment
[FUML] ownedComment

14.5 Class Information Source <<Role>>

Metadata defining the origin or provenance of a set of statements in a model or data.
Note that the source could be a human, an organization, a mapping or other automated processes.

Direct Supertypes

Actual Entity, Metadata

Associations

<<Annotation Property>> made statement : Identifiable Entity [1..*] *Subsets*: metadata about:Identifiable Entity
through association: Source of Information

Metadata representing statements made by an authoritative source.
Sources may be people, organizations, documents, information systems, etc.

14.6 Class Metadata

Information about the source, provenance or origin of information. Metadata may be a managed entity, providing for provenance.
[NIEM] MetadataType

Direct Supertypes

Record

Associations

<<Annotation Property>> metadata about : Identifiable Entity [*] *Subsets*: about:Identifiable Entity
through association: Metadata relationship

The subject of metadata, the entity described by the metadata.
[OWL] annotationSubject of Annotation Assertion

14.7 Association Metadata relationship

Relationship between something and metadata about that thing; data about data.
[OWL] AnnotationAssertion

Association Ends

metadata about : Identifiable Entity [*] *Subsets*: about:Identifiable Entity

The subject of metadata, the entity described by the metadata.
[OWL] annotationSubject of Annotation Assertion

has metadata : Metadata [*] *Subsets*: about:Identifiable Entity

Metadata associated with (data about the information concerning) the subject entity.
[OWL] AnnotationProperty, annotationValue of Annotation Assertion

14.8 Association Record of an Entity

Relationship between a thing and records (or information) about that thing.
Note that in SMIF, things refer to the actual thing they represent, not data about it (unless the type is a record, in which case the "thing" is the data). This relationship recognizes that both a thing and data about the thing are things.

[IDEAS] describedBy: A representedBy that asserts that a Description describes a Thing.

Association Ends

about : Identifiable Entity [*] *Subsets*: about:Identifiable Entity

The thing described by a record.

has record : Record [*] *Subsets*: about:Identifiable Entity

A record about something.

14.9 Association Source of Information

Relation defining an entity making a statement represented within a model. E.g. the person or organization that made a statement.

[ISO 1087] source identifier: information in a terminological entry (3.8.2) which indicates the source documenting the terminological data (3.8.1)

Association Ends

made statement : Identifiable Entity [1..*] *Subsets*: about:Identifiable Entity

Metadata representing statements made by an authoritative source.
Sources may be people, organizations, documents, information systems, etc.

has authoritative source : Information Source [*] *Subsets*: about:Identifiable Entity

Metadata representing the authority behind a statement - who or what made a statement captured in a model.

14.10 Class Statement

Statements provide metadata as to the source of information - who or what said it.
This source of the information may be captured using "InformationSource" metadata about the metadata.

[ISO11404] provision that conveys information

Direct Supertypes

Metadata

Attributes

statement date and time : Value Type
Metadata representing the date and time the statement was made or modified.

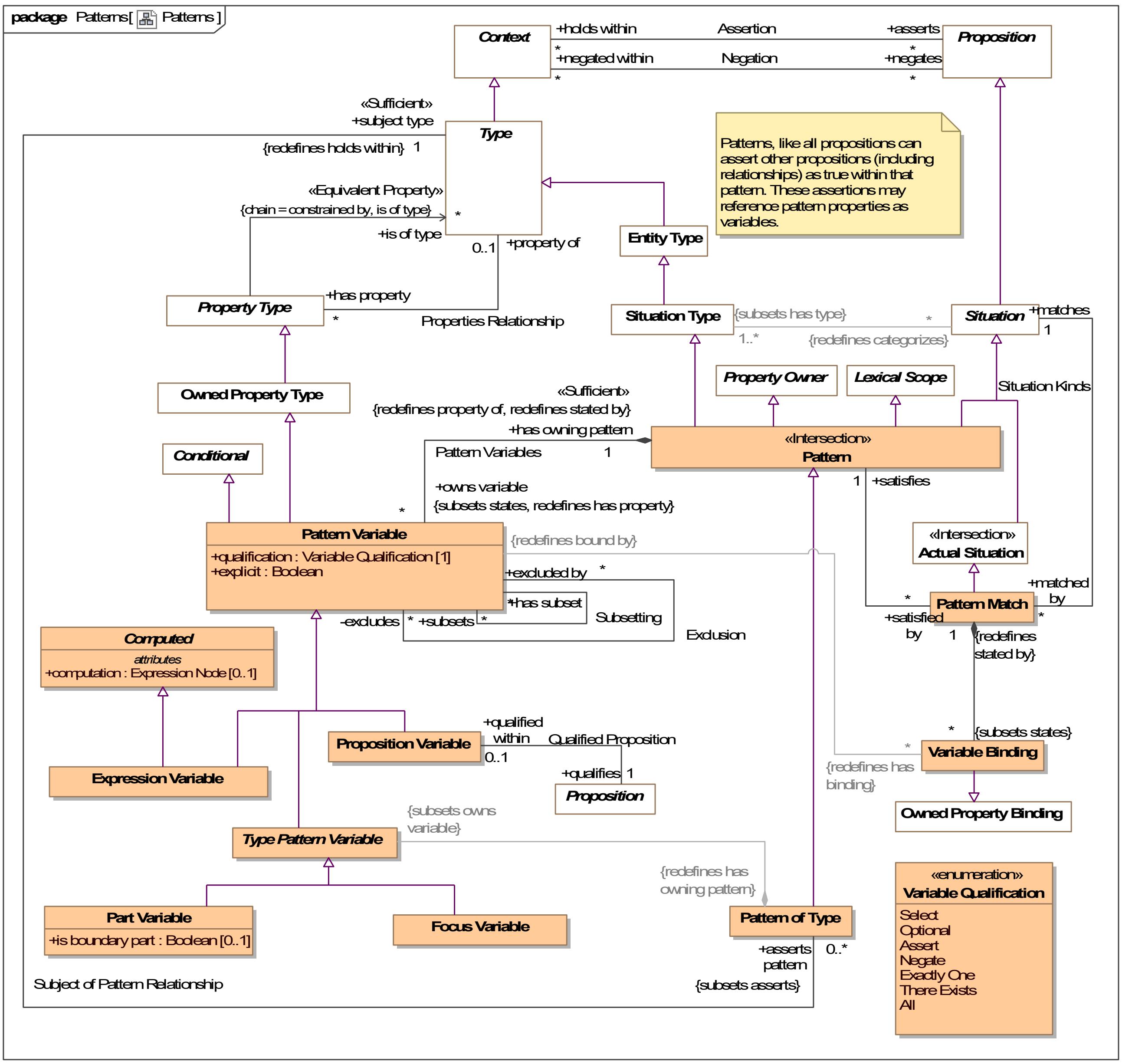
version : Value Type
Metadata representing an identifier for a version of information.

transaction id : Value Type
Identifier for an act or transaction creating or modifying information.

15 SMIF Conceptual Model::Patterns

Patterns are templates for structures or compositions of things that may then be expressed as instances of the pattern.

15.1 Diagram: Patterns



x. Patterns

15.2 Class Computed

Attributes

computation : Expression Node [0..1]
<computation> provides an expression that computes a value for the variable based on the expression applied to the current context..

15.3 Association Exclusion

Association Ends

- excluded by : Pattern Variable [*] *Subsets*: about:Identifiable Entity
- excludes : Pattern Variable [*] *Subsets*: about:Identifiable Entity

15.4 Class Expression Variable

An expression variable defines the value of the variable as computed by <computation>. Note that expression variables are not always able to be asserted or reversed and may therefore not provide for bi-directional mapping patterns. Any ability to assert or reverse a computation is implementation specific.

Direct Supertypes

Computed, Pattern Variable

15.5 Class Focus Variable

A property variable of a pattern representing the extent of the subject type within the context of the owning pattern.
The value of qualification shall be "Select".
The <has type> of the variable is asserted be the same as the subject type of the pattern.

Direct Supertypes

Type Pattern Variable

15.6 Association Match Rules

Relationship defining the match rules for a mapping.

Direct Supertypes

Rule Constrains, Statement

Association Ends

has map rule : Match Rule [*] *Subsets*: about:Identifiable Entity
Map rule that is asserted by a mapping.

map rule of : Mapping [1] *Subsets*: about:Identifiable Entity
Mapping containing a map rule.

15.7 Class Part Variable

A pattern property variable representing a part of the subject type. Additional relations and rules may be made about the part. A type with parts is by its nature a composition.

Direct Supertypes

Type Pattern Variable

Attributes

is boundary part : Boolean [0..1]
True if the property is on the boundary of the pattern and connectible (may have relationships) external to the pattern. e.g. "Port"

15.8 Class Pattern <<Intersection>>

A pattern represents a set of assertions true about individuals or sets of individuals qualified by pattern properties. All propositions asserted or negated by a pattern (as a context) are considered "templates" where identity is not required to match.

The structure of the pattern is defined by the properties and asserted (sub) situations (including relationships) that are asserted by the pattern.

In many cases the relationships and rules defined for a pattern will reference pattern properties. These relationships will hold for instances of the pattern where things are bound to the pattern properties.

[DTV] general situation kind: situation kind that is not an individual situation kind. A situation kind is a general situation kind if it can be exemplified by more than one Event in some possible world, even when it cannot have more than one Event in the possible world chosen to be the universe of discourse.

[UML] StructuredClassifier. Also Similarity with TemplateSignature

[OWL] May be used to represent Class Expressions

Direct Supertypes

Lexical Scope, Property Owner, Situation, Situation Type

Associations

<<Sufficient>> owns variable : Pattern Variable [*] *Subsets*: states:Thing *Redefines*: has property:Property Type
through association: Pattern Variables
A variable property defined within the context of a pattern that is used as part of the patterns definition.
[UML] ownedAttribute

satisfied by : Pattern Match [*]
through association: Pattern Matches
Pattern match that satisfies a pattern.

15.9 Association Pattern Bindings

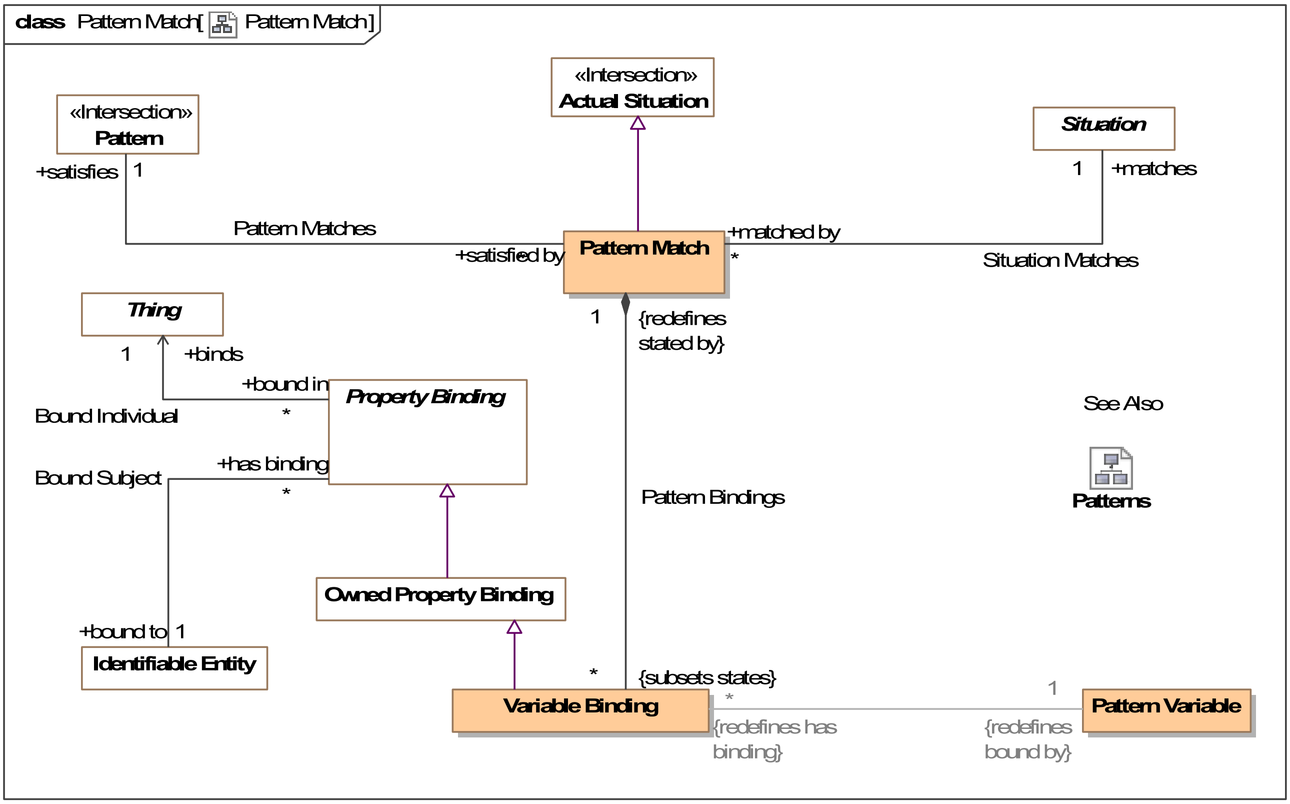
Association Ends

: Variable Binding [*]

: [Pattern Match](#) [1]

15.10 Class Pattern Match

A pattern match provides the corespondents between a pattern and the situations it matches using variable bindings.
A pattern match implies and proves that the pattern <categorizes> the situation.
The matched pattern <states> any consequences of the matching, such as the pattern <categorizes> the pattern instance.



• **Pattern Match**

Direct Supertypes

[Actual Situation](#)

Associations

: [Variable Binding](#) [*] *Subsets*: states:[Thing](#)
through association: [Pattern Bindings](#)

satisfies : [Pattern](#) [1]
through association: [Pattern Matches](#)
Pattern that is satisfied by a "Pattern Match" based on a set of "Variable Bindings".

matches : [Situation](#) [1]
through association: [Situation Matches](#)
The situation qualified as matching the <satisfies> pattern based on the set of "Variable Bindings" stated.

15.11 Association Pattern Matches

Association Ends

satisfies : Pattern [1]

Pattern that is satisfied by a "Pattern Match" based on a set of "Variable Bindings".

satisfied by : Pattern Match [*]

Pattern match that satisfies a pattern.

15.12 Class Pattern of Type

A pattern of type defines a set of properties and relationships that must hold true for all instances of a type. Where the pattern includes parts, the subject type is a composition.
Patterns augment the semantics of th subject type in the context of the pattern.

Direct Supertypes

Pattern

Associations

<<Restriction>> : Type Pattern Variable *Subsets*: owns variable:Pattern Variable

<<Sufficient>> subject type : Type [1] *Redefines*: holds within:Context
through association: Subject of Pattern Relationship

The type which is the context of a pattern of type. The pattern is "about" the subject type.

15.13 Class Pattern Variable

A pattern variable is a property of a pattern that provides a contextual property within that pattern for rules and relationships to be bound to.
A pattern variable is a placeholder for all or a subset of the instances of the variables type.
Properties of an association or relationship may be bound to a pattern variable where the type of the pattern variable is compatible with the type of the relationship's property type.

[UML] Similarity with TemplateParameter
[CL] Functional Term

Direct Supertypes

Conditional, Owned Property Type

Attributes

qualification : Variable Qualification [1]
<qualification> defines the behavior of an element with respect to a pattern - how the variable impacts the selection, evaluation or assertion of the pattern.

explicit : Boolean
If true, Element must be explicitly asserted as the indicted type, not derived or inferred from a supertype or super property.

Associations

<<Restriction>> : Variable Binding [*] *Redefines*: has binding:Property Binding

<<Sufficient>> has owning pattern : Pattern [1] *Redefines*: property of:Type stated by:Lexical Scope
through association: Pattern Variables

Pattern owning a pattern variable.

has subset : Pattern [1] *Redefines*: property of:Type stated by:Lexical Scope
through association: Subsetting

Subsets of the variable.

maps to : Match End [*]
through association: Mapped variable

Map rule end for a property

subsets : Match End [*]
through association: Subsetting

Variable that a subset variable subsets. The subset variable shall be populated by a subset of the <subsets> variable based on the type and constraints of the subset variable.

excluded by : Match End [*]
through association: Exclusion

excludes : Match End [*]
through association: Exclusion

15.14 Association Pattern Variables

Relationship defining variable properties within a pattern.

Direct Supertypes

Statement

Association Ends

owns variable : Pattern Variable [*]

A variable property defined within the context of a pattern that is used as part of the patterns definition. [UML] ownedAttribute

has owning pattern : Pattern [1]

Pattern owning a pattern variable.

15.15 Class Proposition Variable

A proposition variable utilizes some proposition (e.g. relationships) as a part of the definition of a pattern, it extends a basic proposition in that it adds properties to determine the effect the assertion has on pattern instances.
A Proposition Variable is a lexical scope context that <asserts> or <negates> other propositions qualified by <has strength> and <explicit>. As a lexical scope it may "own" the asserted propositions.
Proposition Variable is often used with associations and relationships to define the way pattern properties are related to other pattern properties or actual entities.
For a pattern associations, [UML] Connector. (type = has type). Each ConnectorEnd corresponds with a Structured Property Binding.

Direct Supertypes

Pattern Variable

Associations

qualifies : Proposition [1]
through association: Qualified Proposition

15.16 Association Qualified Proposition

Association defining exactly one proposition (such as an association) qualified by a qualified proposition variable.

Association Ends

qualifies : Proposition [1]
qualified within : Proposition Variable [0..1]

15.17 Association Situation Matches

Association Ends

matches : Situation [1]

The situation qualified as matching the <satisfies> pattern based on the set of "Variable Bindings" stated.

matched by : Pattern Match [*]

Pattern matches that match the subject situation.

15.18 Association Subject of Pattern Relationship

Relationship defining the subject pattern of a type specific pattern.

Direct Supertypes

Assertion

Association Ends

asserts pattern : Pattern of Type [0..*]

A pattern asserted for all instances of a type. Where the pattern includes parts, the type defines a composition.

subject type : Type [1]

The type which is the context of a pattern of type. The pattern is "about" the subject type.

15.19 Association Subsetting

In a pattern or mapping rule, defines a variable that represents a subset of another property (or if multiple, their union). The subset may be constrained by a more specific type, expressions or required cardinalities.
Subset: Set A is a subset of set B if all of the elements (if any) of set A are contained in set B

Association Ends

subsets : Pattern Variable [*]

Variable that a subset variable subsets. The subset variable shall be populated by a subset of the <subsets> variable based on the type and constraints of the subset variable.

has subset : Pattern Variable [*]

Subsets of the variable.

15.20 Class Type Pattern Variable

Type Pattern variable is an abstract supertype that provides for a restriction that parts and focus properties must be owned by a pattern of a type.

Direct Supertypes

Pattern Variable

Associations

- <<Restriction>> : Pattern of Type *Redefines*: has owning pattern:Pattern

15.21 Class Variable Binding

A variable binding defines a value for a particular variable of a particular owning pattern as part of a pattern match.

Direct Supertypes

Owned Property Binding

Associations

- <<Restriction>> : Pattern Variable [1] *Redefines*: bound by:Property Type
- : Pattern Match [1] *Redefines*: stated by:Lexical Scope
  through association: Pattern Bindings

15.21.1 Enumeration Variable Qualification

Variable qualification values define the behavior of an element with respect to a pattern - how it impacts the selection, evaluation or assertion of the pattern.

```
package SMIF Conceptual Model::Patterns
public enum Variable Qualification
{Select, Optional, Assert, Negate, Exactly One, There Exists, All}
```

Literals

- Select

Select is used in query and mapping patterns, all elements of the classified type that match the pattern are selected as instances of the pattern.
Select may be considered a qualified "All". Select does not assert the existence of something, it determines the existence of a pattern match such that other assertions may be made.
Where a pattern is asserted, "Select" variables shall be asserted.
Relationships between properties with <quantifier>=Select must hold between the selected properties for the pattern to be asserted.

- Optional

Optional is used in query and mapping patterns, the property shall be populated as a consequence of the pattern matching.
Where a pattern is asserted, "Optional" variables shall not be asserted.
Optional is the default if no qualification is stated.

- Assert

The property does not impact the selection of the pattern, it is an asserted consequence of the pattern.

- Negate

The property does not impact the selection of the pattern, it is negated consequence of the pattern - it may not exist.

- Exactly One

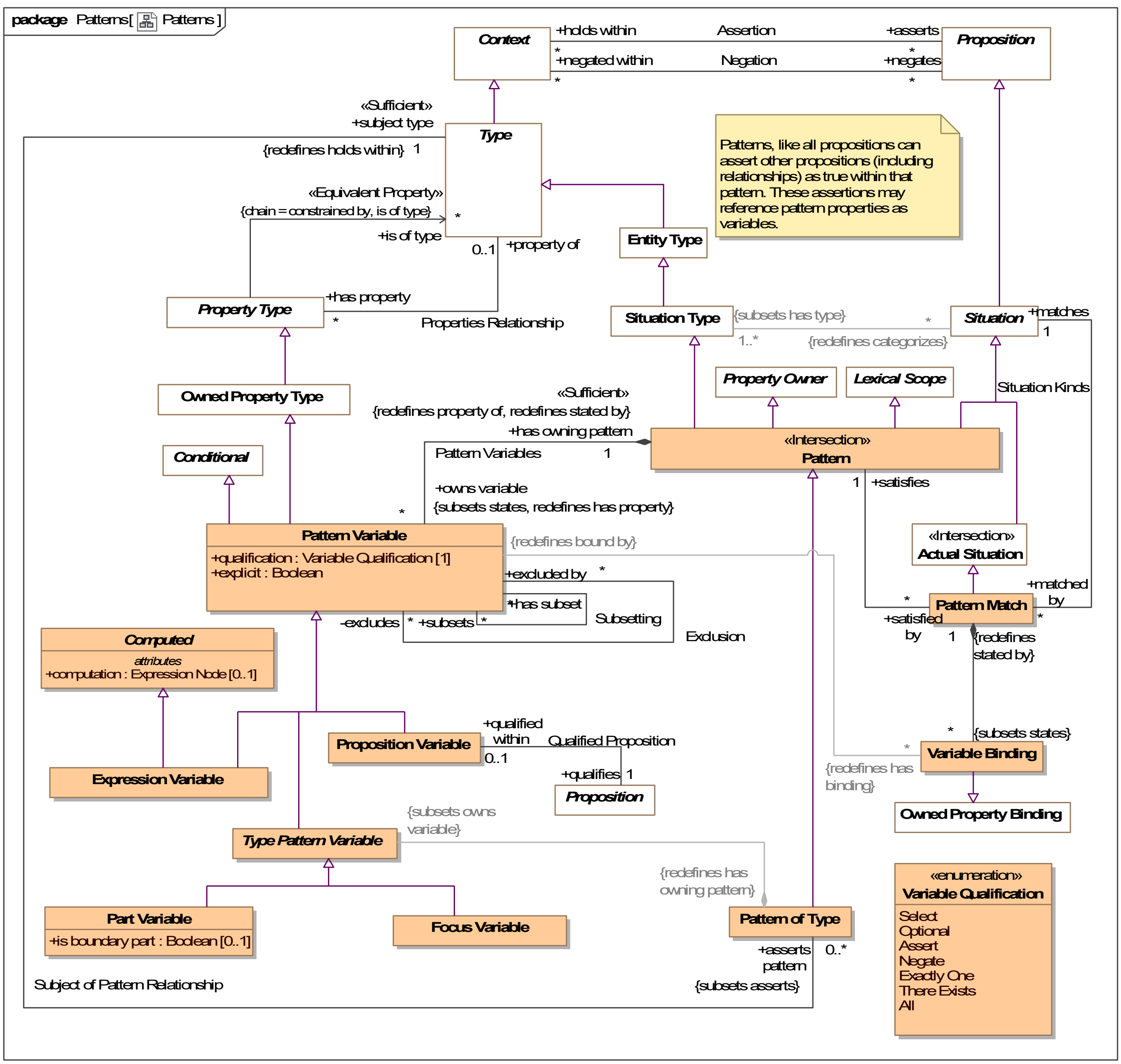
The existential quantifier limited to exactly one of a potentially larger set of the properties type.

- There Exists

The existential quantifier - at least one of the properties type.

- All

The universal quantifier - the quantified property is a stand-in for all elements of the existent of the quantified type

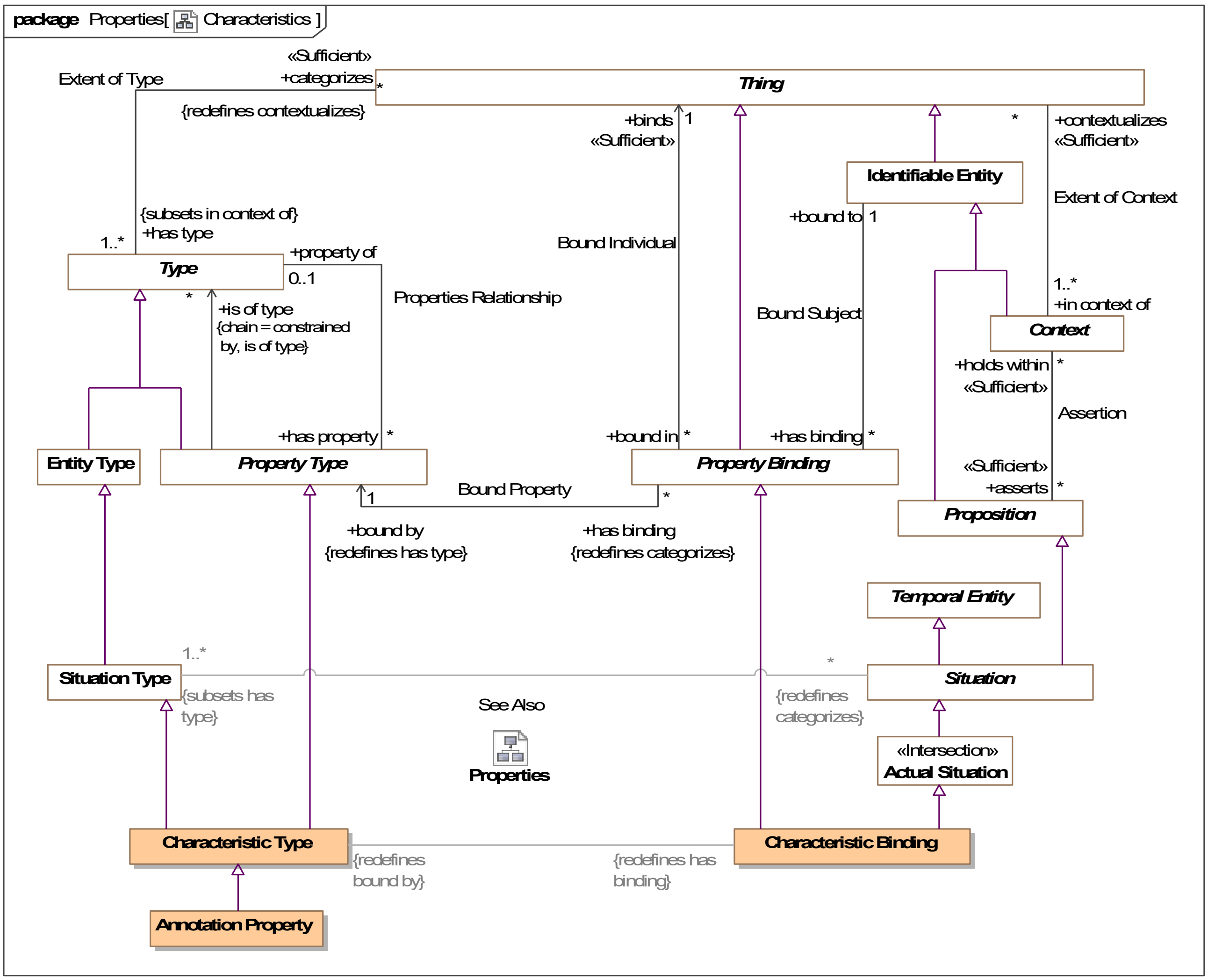


y.   **Patterns**

16 SMIF Conceptual Model::Properties

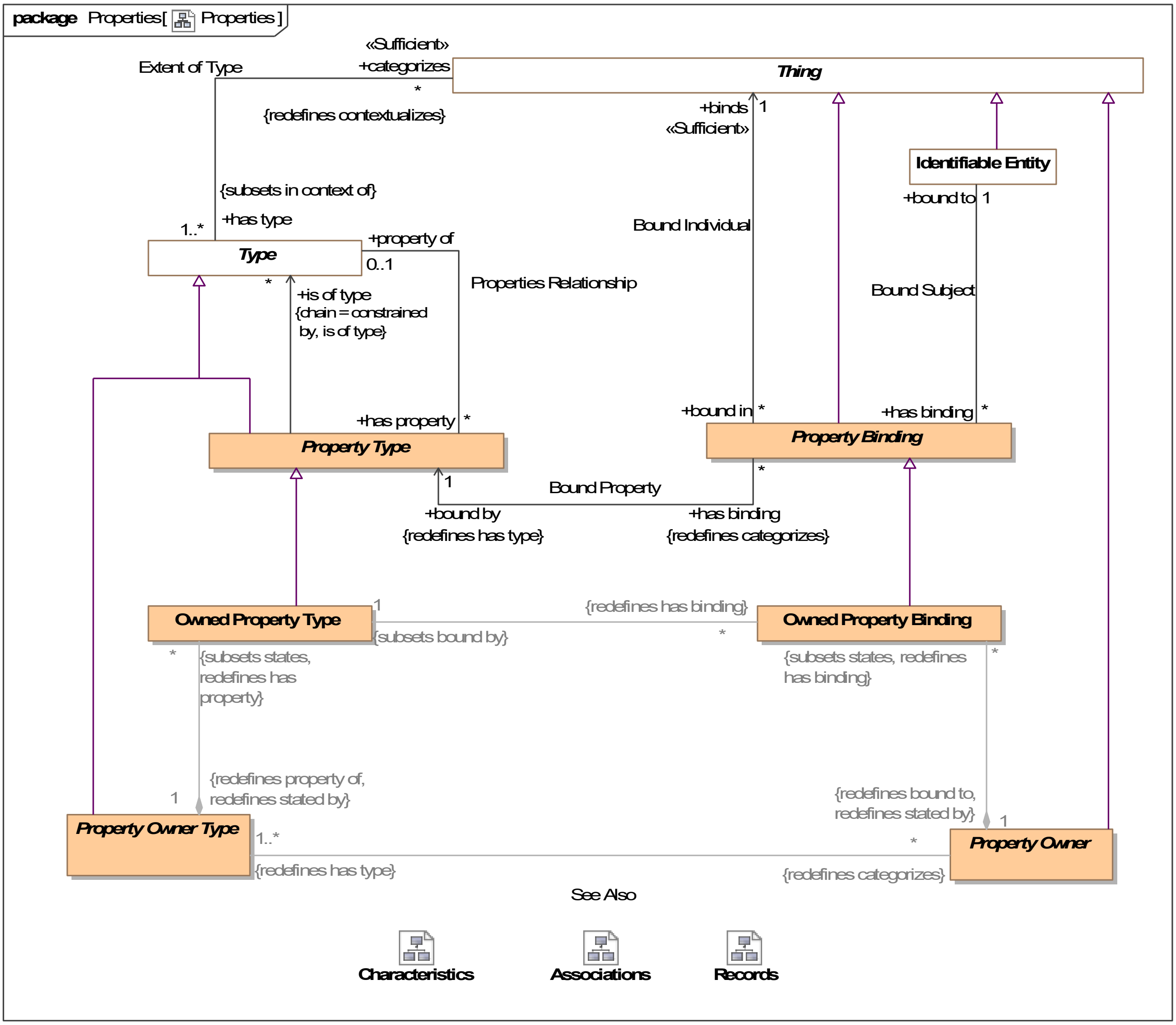
Properties define the most granular connections between entities or values. Properties may be used as the ends of relationships, to represent individual characteristics or as elements of a data structure.

16.1 Diagram: Characteristics



z. **Characteristics**

16.2 Diagram: Properties



aa. Properties

16.3 Class Annotation Property

An annotation property is a specialization of property where the referenced elements represent metadata about the related proposition, structure or information (or model element) rather than a fact or condition of the domain being represented.

For an annotation property, <is of type> describes instances of the structured type for which the property is defined. Typical uses of annotations include provenance of information, when a record was created, etc.

[ISO11404] annotation: descriptive information unit attached to a datatype, or a component of a datatype, or a procedure (value), to characterize some aspect of the representations, variables, or operations associated with values of the datatype

Direct Supertypes

Characteristic Type

16.4 Association Bound Individual

Relationship defining the thing bound to a subject based on a bound property - the "object" of the property binding.

Association Ends

binds : Thing [1]   *Redefines*: stated by: Lexical Scope

The thing bound to a property in a specific situation. E.g. if the weight of truck-XYZ is 4500 LBS, the bound individual would be "4500 LBS".
[FUML] value
[OWL] rdf:object

bound in : Property Binding [*]   *Redefines*: stated by: Lexical Scope

Bindings in which a thing participates.

16.5 Association Bound Property

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

Direct Supertypes

Extent of Type

Association Ends

has binding : Property Binding [*]   *Redefines*: stated by: Lexical Scope

Bindings referencing a property.

bound by : Property Type [1]   *Redefines*: stated by: Lexical Scope

The property a binding binds a thing to.
[FUML] definingFeature
[OWL] rdf:predicate

16.6 Association Bound Subject

Relationship defining the subject of a bound property. Where the subject is a relationship, the relationship becomes transparent and the applicable subject(s) are the other ends of the relationship. E.g. if the weight of truck-XYZ is 4500 LBS, the bound subject would be Truck-XYZ".

Association Ends

has binding : Property Binding [*]   *Redefines*: stated by: Lexical Scope

Bindings asserted for properties within a situation.

bound to : Identifiable Entity [1]   *Redefines*: stated by: Lexical Scope

The subject of a property binding.
[FUML] owningInstance (note that in SMIF the owner and subject may not be the same). Where the are the same, the semantics are the same as FUML.
[OWL] rdf:subject

16.7 Class Characteristic Binding

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

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

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

[Guizzardi] quality(x) =def $\exists!$U qualityUniversal(U) $\wedge$ (x::U)

[DOLCE] Quality

Direct Supertypes

Actual Situation, Property Binding

Associations

<<Restriction>> : Characteristic Type *Redefines*: bound by:Property Type

16.8 Class Characteristic Type

A kind of characteristic a type of thing may have, e.g. paint may have a color. Characteristic type is the type of characteristic bindings which are "first class" elements and may participate in relationships and have other characteristics.

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

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

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

[Guizzardi] qualityUniversal(U)

[DOLCE] Quality Type

[OWL] rdf:Statement

[UML] Property

Direct Supertypes

Property Type, Situation Type

Associations

<<Restriction>> : Characteristic Binding *Redefines*: has binding:Property Binding

16.9 Class Owned Property Binding

An owned property binding defines a value for a particular property of a particular owning property type (or structure). Similar to an OWL triple, an owned property binding does not have independent identity.
Constraint: Each owned property binding must be <bound by> an owned property type that is owned by the <has type> owned type of the <bound to> property owner.

Owned property type is abstract and not intended to directly represent semantic elements.

Direct Supertypes

Property Binding

Associations

<<Restriction>> : Owned Property Type [1] *Subsets*: bound by:Property Type

A structure property binding may bind a characteristic.

<<Restriction>> : Property Owner [1] *Redefines*: bound to:Identifiable Entity stated by:Lexical Scope

16.10 Class Owned Property Type

An owned property type is a property definition defined as a composite part of an association type - most often used in data structures and relationships. Association property types are the types of association property bindings. Also known as "association end".

[FIBO] Relationship Property
[FUML] memberEnd (of association) Property

Direct Supertypes

Property Type

Associations

<<Restriction>> : Owned Property Binding [*] *Redefines*: has binding:Property Binding
<<Restriction>> : Property Owner Type [1] *Redefines*: stated by:Lexical Scope property of:Type

16.11 Association Properties Relationship

Relationship defining the set of properties defined for a type.
Where the <property of> type is a relationship type, the "subject" of the property is the other ends (properties) of the relationship.
Where the <property of> type is not a relationship, the subject of the property is the <property of> type.

Association Ends

has property : Property Type [*] *Redefines*: stated by: Lexical Scope property of: Type

A property of a structured type such that there may be bindings of a thing to instances of the structured type with reference to the property which defines the semantics of the bound thing withing the context of the structure.
[FUML] feature
[UML] memberEnd. attribute (of classifier).

property of : Type [0..1] *Redefines*: stated by: Lexical Scope property of: Type

Type for which a property is relevant. The domain of the property.
<property of> excludes "Owned Property Type" and ("Association Type" that is not "Relationship Type")
[FUML] featuringClassifier
[OWL] Domain

16.12 Class Property Binding

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

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

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

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

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

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

[CL] Binding:

[OWL] Union(ObjectPropertyAssertion, DataPropertyAssertion, AnnotationAssertion), RDF Triple
=Note: RDF Triples do not have identity where as some subtypes of SMIF:Property Type do have identity and are therefor statements.

Direct Supertypes

Thing

Associations

binds : Thing [1]
through association: Bound Individual

The thing bound to a property in a specific situation. E.g. if the weight of truck-XYZ is 4500 LBS, the bound individual would be "4500 LBS".
[FUML] value
[OWL] rdf:object

bound by : Property Type [1] *Redefines*: has type:Type
through association: Bound Property

The property a binding binds a thing to.
[FUML] definingFeature
[OWL] rdf:predicate

bound to : Identifiable Entity [1]
through association: Bound Subject

The subject of a property binding.
[FUML] owningInstance (note that in SMIF the owner and subject may not be the same). Where the are the same, the semantics are the same as FUML.
[OWL] rdf:subject

16.13 Class Property Owner

Property Owner is an abstract element for anything that may own a set of property bindings. This element is abstract and not intended to directly represent domain concepts. Subtypes of property owner provide semantic interpretation.

Direct Supertypes

Thing

Associations

- <<Restriction>> : Owned Property Binding [*] *Subsets*: states:Thing *Redefines*: has binding:Property Binding
- <<Sufficient>><<Restriction>> : Property Owner Type [1..*] *Redefines*: has type:Type

16.14 Class Property Owner Type

A type of Property Owner (See Property Owner for details) which defines a set of "Owned Property Types" which are the types of owned property bindings.
Property owner is abstract and not intended to directly represent semantic elements.

Direct Supertypes

Type

Associations

- <<Sufficient>><<Restriction>> : Property Owner [*] *Redefines*: categorizes:Thing
- <<Restriction>> : Owned Property Type [*] *Subsets*: states:Thing *Redefines*: has property:Property Type

16.15 Class Property Type

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

[Guizzardi] MomentUniversal

[FUML] Parameter where owner is operation. Otherwise Property.

[UML] Property. All typed elements in SMIF are Property Types.

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

[OWL] rdf:Property, ObjectUnionOf(owl:ObjectProperty, oe;DatatypeProperty).

Direct Supertypes

Type

Associations

is of type : Type [*]

A type of instances bound to a property. Also known as the "range" of a property.
If asserted the property rule shall be owned and asserted by the properties <property of> type.

[OWL] Range

<<Restriction>> : Property Constraint [*]   *Subsets*: constrained by:Rule

property of : Type [0..1]

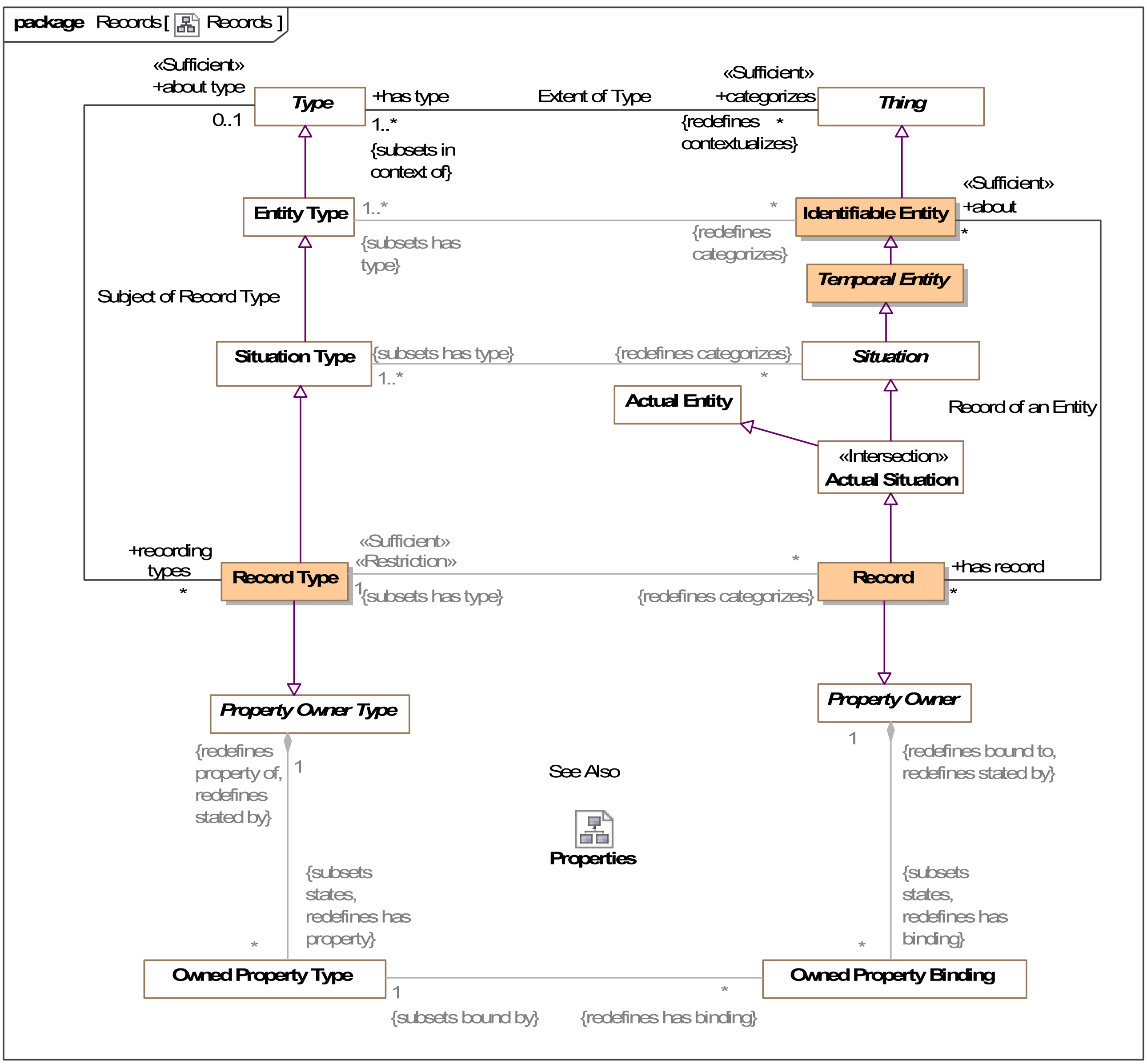
through association: Properties Relationship

Type for which a property is relevant. The domain of the property.
<property of> excludes "Owned Property Type" and ("Association Type" that is not "Relationship Type")
[FUML] featuringClassifier
[OWL] Domain

17 SMIF Conceptual Model::Records

A record of the condition of an entity at a point in time - this includes facts, speech acts and DBMS records. Records are a kind of information.
Records are typically used in data representations, not conceptual models.

17.1 Diagram: Records



bb. Records

17.2 Class Record

A record of the condition of an entity at a point in time - this includes facts, speech acts and DBMS records. Records are typically used in data representations, not conceptual models. Records specialize associations as owners of properties.

[IDEAS] A Representation that describes a Thing

Direct Supertypes

Actual Situation, Property Owner

Associations

about : Identifiable Entity [*]
through association: Record of an Entity
The thing described by a record.

<<Sufficient>><<Restriction>> : Record Type [1] *Subsets*: has type:Type

17.3 Class Record Type

Type of the record of the condition of an entity at a point in time - this includes facts, speech acts and DBMS records. A record type may involve variant and invariant types as variables. Those that are enumerated in a "uniqueness constraint" are invariant (independent variables) uniquely identify the situation which is the subject of the fact type where as the other variables may change over time (dependent variables).
Record types may be grounded in atomic relations by using invariant conditions.
Record types represent typical "data structures".

Direct Supertypes

Property Owner Type, Situation Type

Associations

<<Sufficient>><<Restriction>> : Record [*] *Redefines*: categorizes:Thing

about type : Type [0..1]
through association: Subject of Record Type
Thing for which a record exists

17.4 Association Subject of Record Type

Relationship defining types of records for another type.

Association Ends

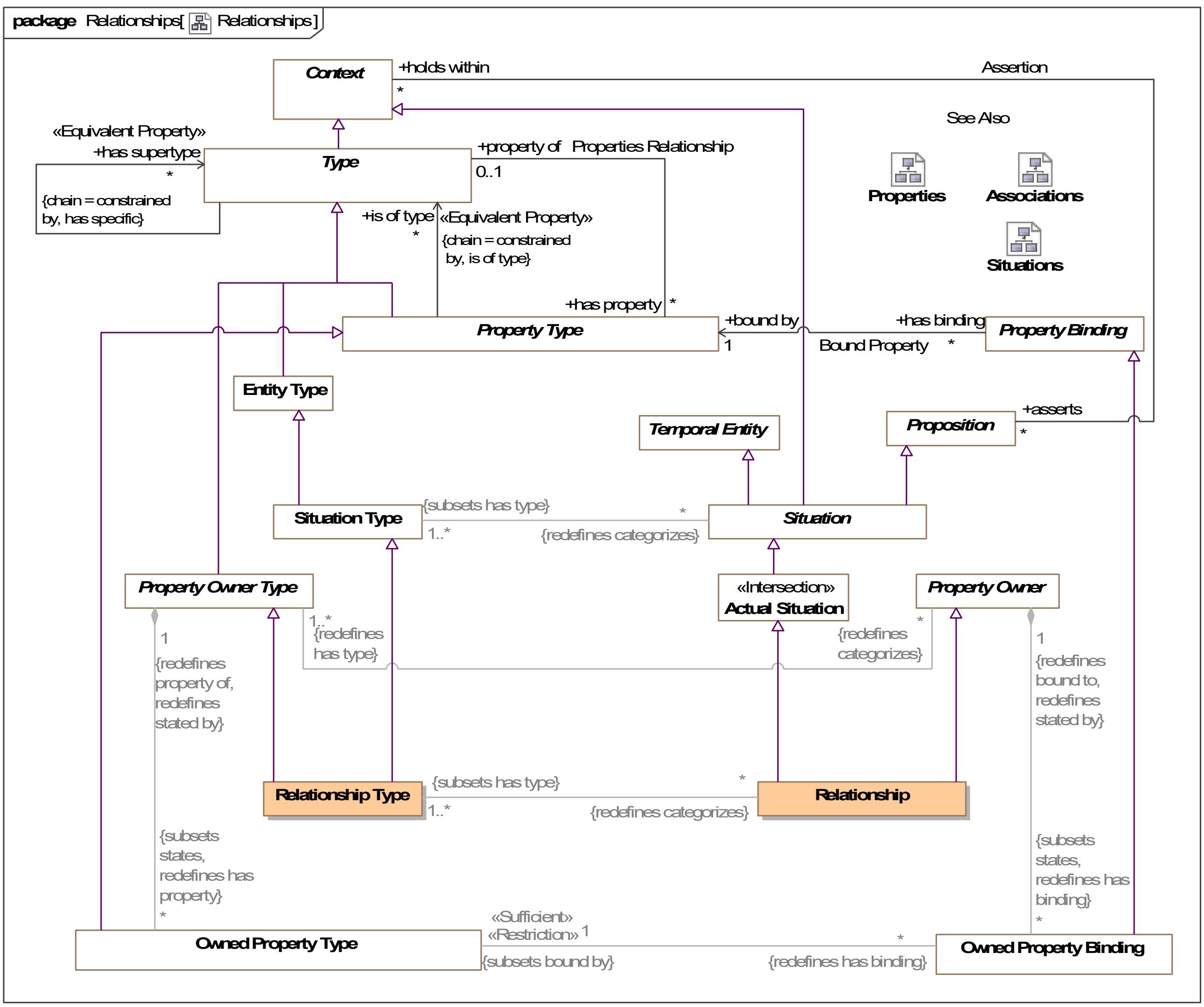
about type : Type [0..1]
Thing for which a record exists

recording types : Record Type [*]
Record for a thing.

18 SMIF Conceptual Model::Relationships

Relationships are primitive but identifiable conditions that relate other entities through properties of the relationships. Relationships have their semantics described by a relationship type. The ends of relationships are defined by "structured property type", a relationship may have any number of "ends".
Relationships are first-class "actual" and "temporal" things that exist in their own right. These are known as "external relations" in much of the theoretical literature.

18.1 Diagram: Relationships



cc. Relationships

Relations are atomic actual situations that bind 2 or more properties as a fact.

18.2 Class Relationship

A relationship defines a situation involving related things. A relationship may be asserted within a context as true or false within that context. Each relationship type has a number of bindings of which do not change for the life of the relationship..
A relationship may be true or false within its context (including a timeframe) but is atomic in its truth value. Relationships may participate in (be bound to) other relationships and as such bindings involving a relationship may change over time. That is, relationships are "first class" objects.

[IDEAS] tuple: A relationship between two or more things.
Note: SMIF allows one end of a relationship.

[OWL] An OWL class that is a subclass of SMIF: Relationship

Direct Supertypes

Actual Situation, Property Owner

Associations

<<Sufficient>><<Restriction>> : Relationship Type [1..*]   *Subsets*: has type:Type

18.3 Class Relationship Type

A relationship type defines a type of condition, the relationship, involving related things. A relationship may be asserted within a context as true or false within that context. Each relationship type has a number of <has property> "structured property type" properties which describe the role of the related things with respect to the relationship, values of which uniquely do not change for the life of the relationship.
A relationship may be true or false within its context (including a timeframe) but is atomic in its truth value. Relationships may participate in (be bound to) other relationships and as such bindings involving a relationship may change over time.
The terms for properties of a relationship in a conceptual model are typically verb phrases, connecting the relationship with the related types.

[FIBO] A kind of Mediating Thing

[IDEAS] TupleType: The Powertype of tuple.

[FUML] Association where memberEnd corresponds with <has property>. Note that SMIF relationships are "first class" and may also be considered to correspond to an association class where there are any properties or other relationships referencing the subject relationship.

[UML] AssociationClass (note that "end ownership" is meaningless in SMIF).

[Guizzardi2015] Relator: endurants of a special kind, with the power of connecting (mediating) other endurants. Note: Guissardi "mediation" corresponds with relationship properties.

Direct Supertypes

Property Owner Type, Situation Type

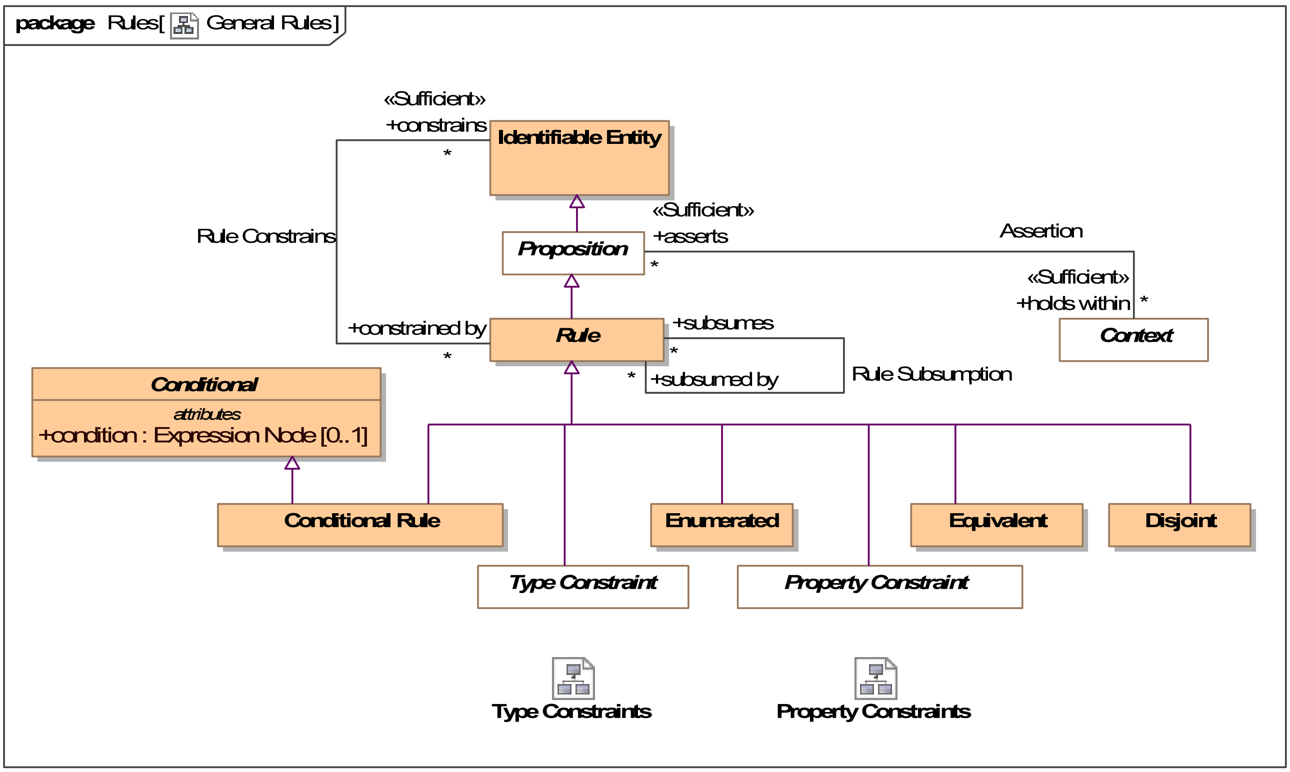
Associations

<<Sufficient>><<Restriction>> : Relationship [*]   *Redefines*: categorizes:Thing

19 SMIF Conceptual Model::Rules

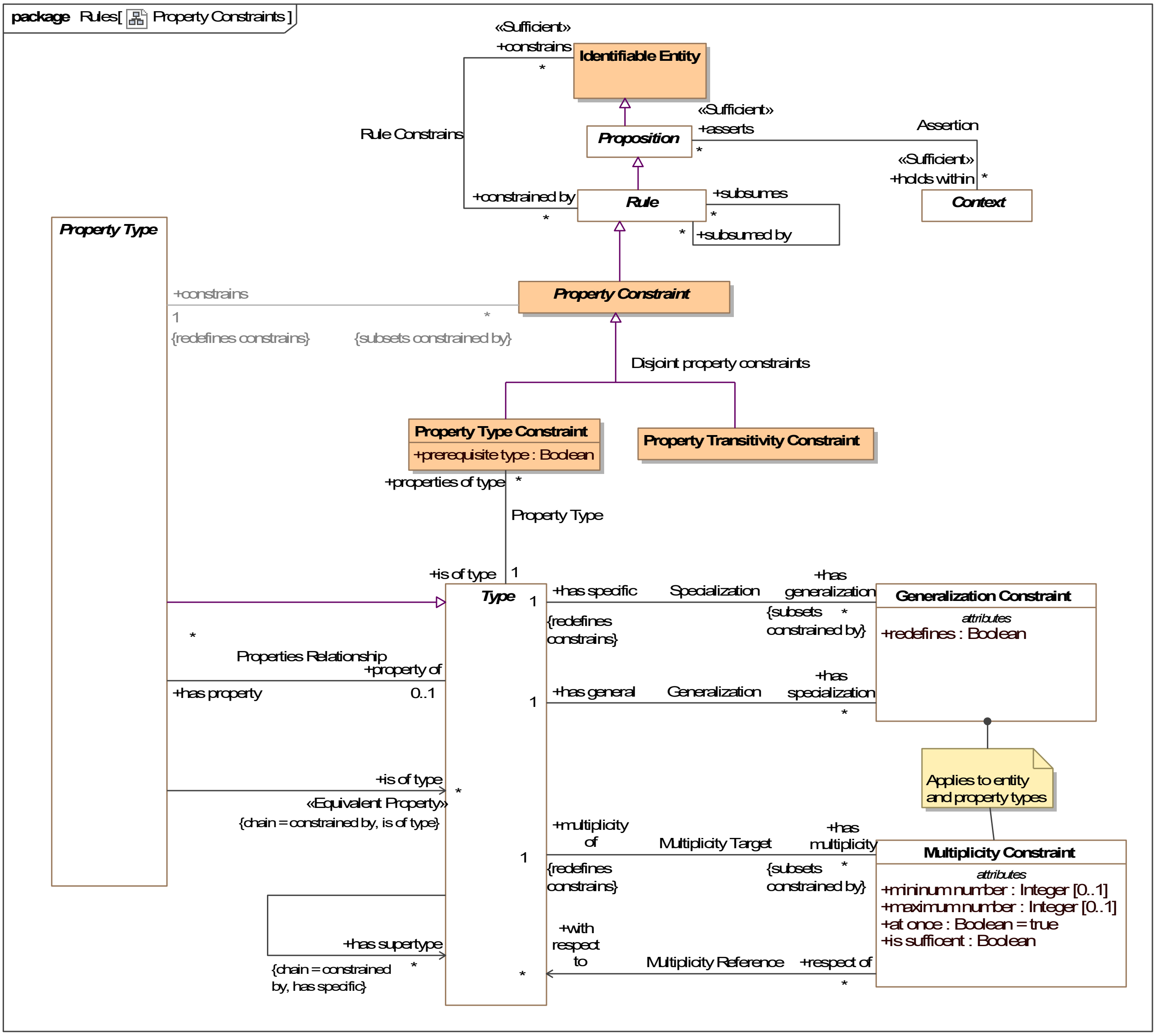
Rules define constraints or behaviors that are asserted in specified context.

19.1 Diagram: General Rules



dd. General Rules

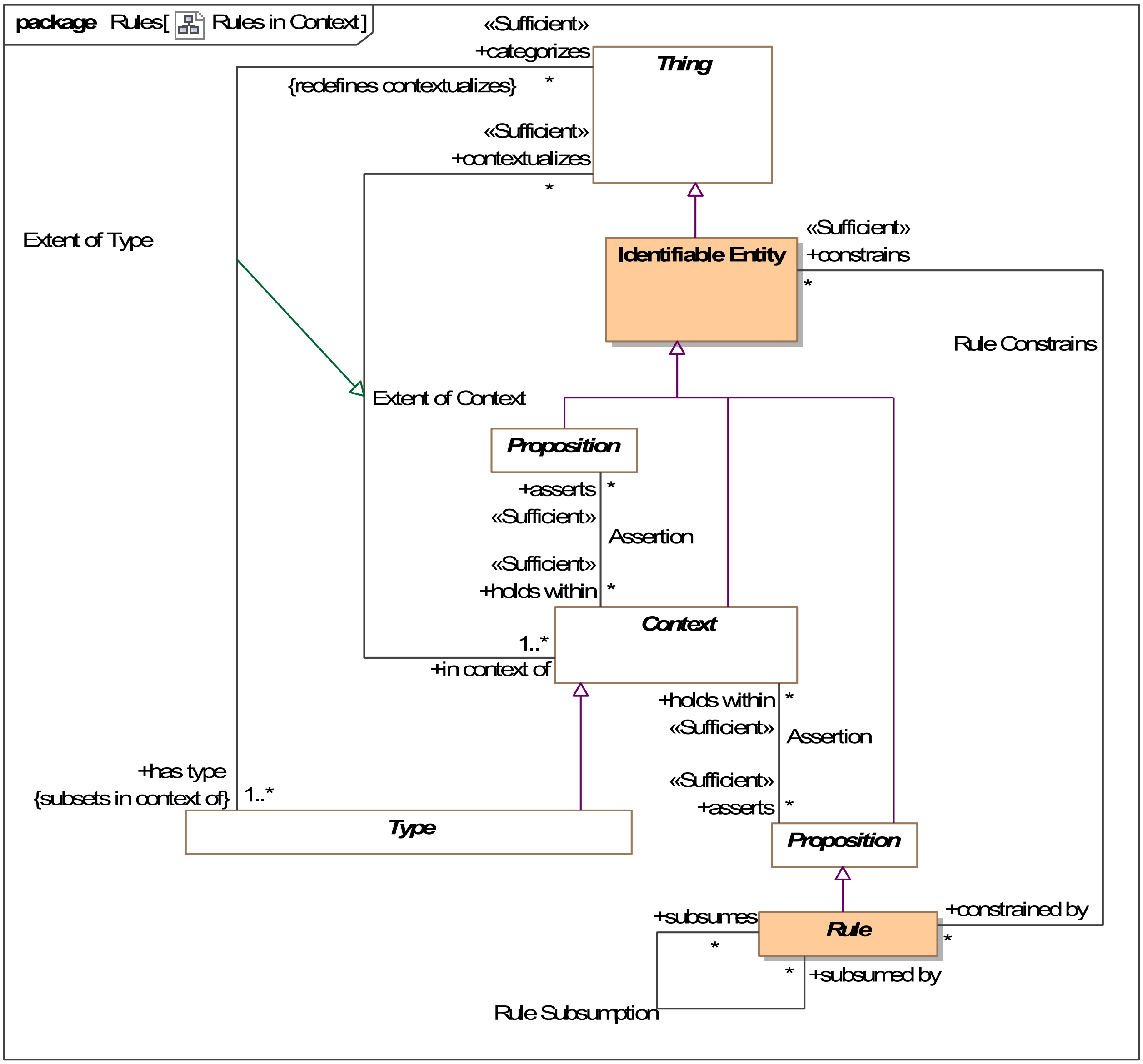
19.2 Diagram: Property Constraints



ee. Property Constraints

This diagram focuses on rules about properties.

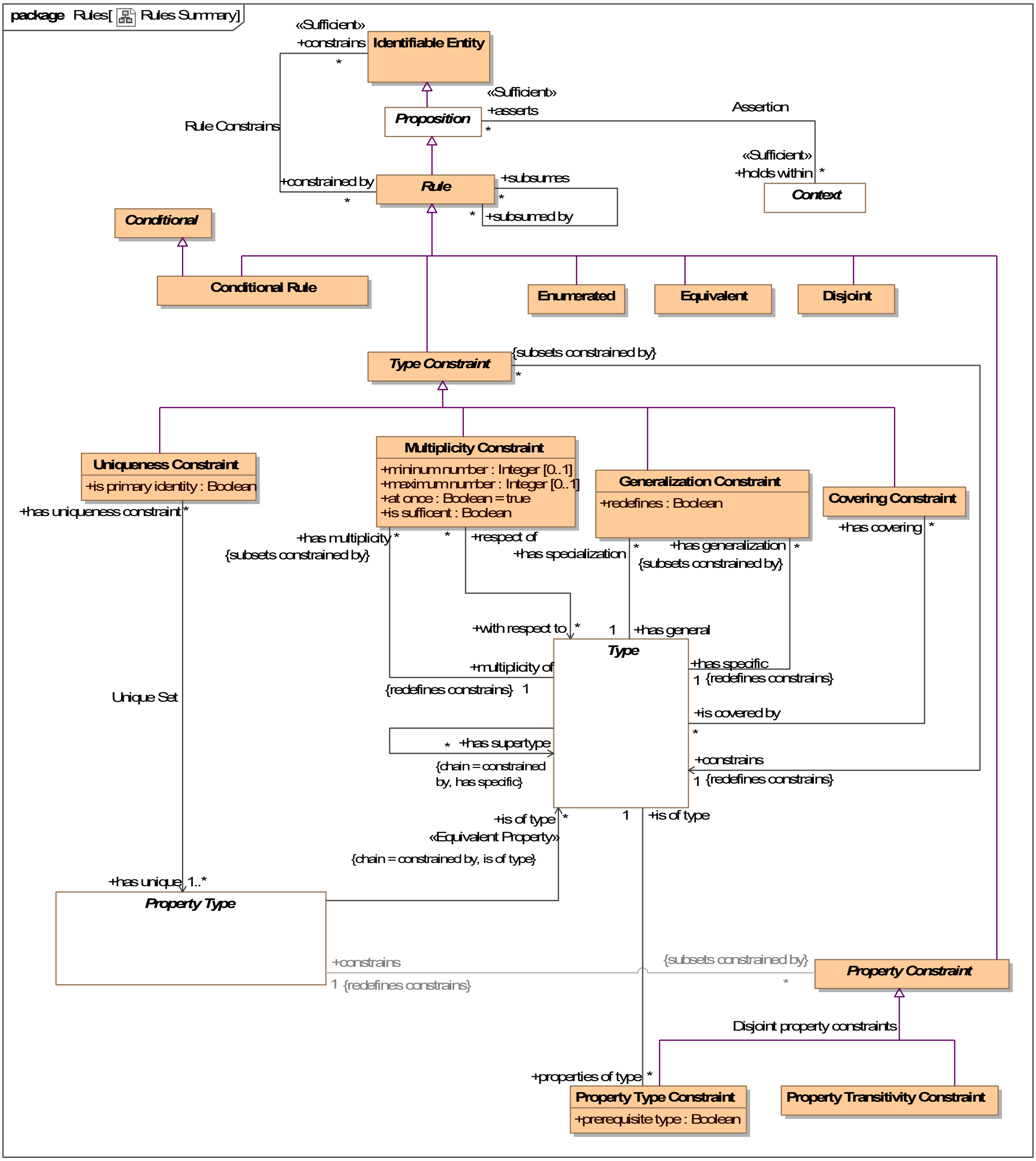
19.3 Diagram: Rules in Context



ff. Rules in Context

This diagram shows how rules are propositions that may be asserted within any context to apply to any other context, thus realizing the "open world assumption".

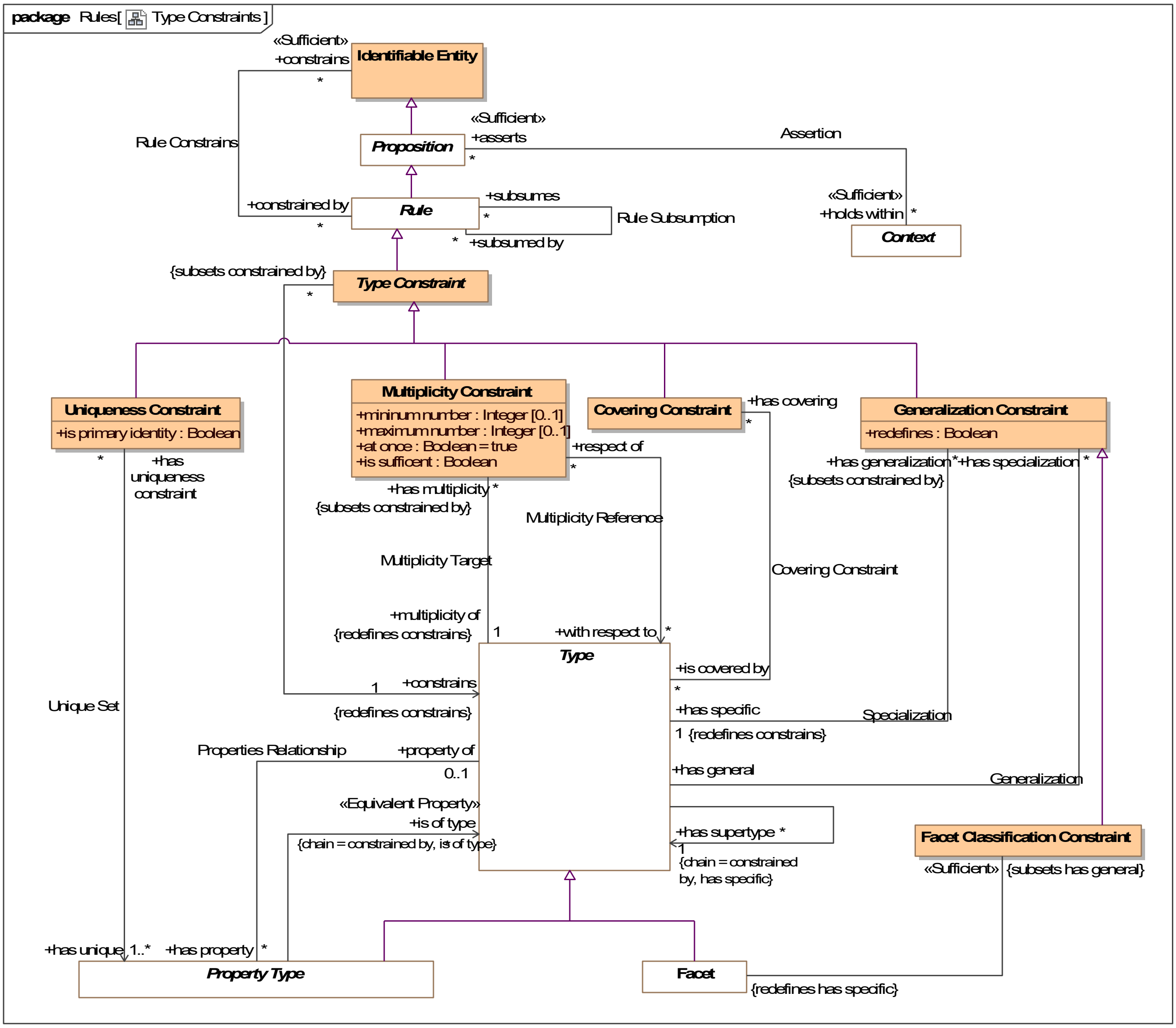
19.4 Diagram: Rules Summary



gg. Rules Summary

This diagram shown a summary of the primary rules.

19.5 Diagram: Type Constraints



hh. Type Constraints

This diagram focuses on rules about types (note that property types are also types).

19.6 Class Conditional

Anything with a condition defined by an expression.

Attributes

condition : Expression Node [0..1]
Condition that must be TRUE for an element to be asserted. All values other than "TRUE" are FALSE.

19.7 Class Conditional Rule

A rule with a general expression as a condition that applies to what the rule <constrains>. Where asserted, the condition must be true.
[UML] Constraint where "context" corresponds with <holds within> and "constrainedElement" corresponds with "constrains". "specification" corresponds with "condition".

Direct Supertypes

Conditional, Rule

19.8 Class Covering Constraint

A constraint that the extent (<categorizes> things) of the <constrains> type is equivalent to the union of the extents of the <is covered by> types.
[UML] GeneralizationSet with isCovering=TRUE. "constrains" corresponds with the common "general" of each Generalization". "is covered by" corresponds with each "special" of each generalization.

Direct Supertypes

Type Constraint

Associations

is covered by : Type [*]
through association: Covering Constraint
A type covered by a covering constraint.

The <constrains> type must be a direct supertype of all <is covered by> types.

19.9 Association Covering Constraint

Relationship defining the types covered by a covering constraint.

Association Ends

is covered by : Type [*]
A type covered by a covering constraint.

The <constrains> type must be a direct supertype of all <is covered by> types.

has covering : Covering Constraint [*]
Covering constraints of a type.

19.10 Class Disjoint

Disjoint is a rule that the things denoted by what the rule <constrains> do not and may not denote any of the same set of things.
When applied to a context (including types) all elements contextualized are included in the set of disjoint individuals.

[FIBO] Mutually Exclusive sets

[IDEAS] PartitionOfSetOfDisjointIndividuals: A FusionOfSetOfIndividuals whose fusioned Type is a

SetOfDisjointIndividuals.

[UML] [UML] GeneralizationSet with isDisjoint=TRUE. "constrains" corresponds with "is covered by" of each "special" of each generalization. Note the SMIF does not require that disjoint elements have a common supertype, one may be inferred for UML mapping.

[OWL: Union(DisjointClasses, DisjointObjectProperties, DisjointDataProperties, DifferentIndividuals)

Direct Supertypes

Rule

19.11 Class Enumerated

The contextualized elements of the <constrains> context is a closed (enumerated) set, it can not be extended. A.K.A. "Closed World Assumption". Elements may not be asserted by any context other than the one specified in <holds within>.

[FIBO] Selections of Things
[FUML] Wen constraining a type, corresponds with [FUML] "Enumeration". SMIF enumerations are not limited to literals. The "ownedLiteral" corresponds with all elements owned by <holds within>.
[ISO11404] Enumerated: enumerated is a family of datatypes, each of which comprises a finite number of distinguished values having an intrinsic order.
[OWL] ObjectUnitionOf(DataOneOf, ObjectOneOf)

Direct Supertypes

Rule

19.12 Class Equivalent

Equivalent is a rule that the things the rule <constraints> denote the same set of things. When applied to a context (including types) each thing the context contextualizes is included in the set of equivalent things.

Related to*: [ISO 1087] synonymy: relation between or among terms (3.4.3) in a given language representing the same concept (3.2.1)

Related to*: [ISO 1087] equivalence: relation between designations (3.4.1) in different languages representing the same concept (3.2.1)

* SMIF relates concepts, not terms. synonymy may also be represented by multiple terms for the same concept.

[OWL] Union(SameIndividual, EquivalentClasses, EquivalentObjectProperties, EquivalentDataProperties)

Direct Supertypes

Rule

19.13 Class Facet Classification Constraint

A Facet Classification Constraint asserts that the specialized type is "non rigid" with respect to the general (rigid) type - that is the <has specific> type may change over the lifetime of instances of the <has general> type. The <has specific> type will be inferred to be a Facet. e.g. "Registered voter" is a facet of a person.

[FIBO] isPlayedBy

Direct Supertypes

Generalization Constraint

Associations

<<Sufficient>> : Facet *Redefines*: has specific:Type

19.14 Association Generalization

Relationship defining the general type of a generalization constraint.

[ISO 1087] generic concept: concept (3.2.1) in a generic relation (3.2.21) having the narrower intension (3.2.9)

Association Ends

has general : Type [1] *Redefines*: has specific: Type

The general type in the Generalization rule.

[ISO 1087] concept (3.2.1) in a generic relation (3.2.21) having the broader intension (3.2.9)

[FUML] General (Where redefines is false or not defined)
[FUML] RedefinableElement.redefinedElement (Where redefines is true)

has specialization : Generalization Constraint [*] *Redefines*: has specific: Type

Specialization rules for a type.

19.15 Class Generalization Constraint

A Type Generalization Constraint is a taxonomic relationship between a more general <has general> type and a more specific <has specific> type. Each instance of the specific type is also an instance of the general type. The specific type inherits the properties and rules of the more general type.
The extent (<categorizes> property) of the specific type is the same as or a subset of the extent of the more general type. Note that "multiple inheritance" is supported.

[IDEAS] superSubtype: A couple relating two Types which asserts that one type is a subset of the other.

[ISO 1087] generic relation: genus-species relation relation between two concepts (3.2.1) where the intension (3.2.9) of one of the concepts includes that of the other concept and at least one additional delimiting characteristic (3.2.7)

[FIBO] Inheritance

[UML] Generalization

[Guizzardi] (Specialization relation): Let F and G be two universals such that F is a specialization of G. Then, for all $w \in W$ we have that $extw(F) \subseteq extw(G)$

[OWL] Union(SubClassOf, SubPropertyOf)

Direct Supertypes

Type Constraint

Attributes

redefines : Boolean

Defines the generalization as a redefinition, subsuming the more general type in the definitional context.

Where <redefines> is true the more specific type subsumes the more general type in the definition context. In this case the more general and more specific sets are equivalent. A type may be redefined multiple times, as long as it is unambiguous which definition applies for a particular instance.

Where <redefines> is false or not defined the more specific type represents a subset of the more general property.

Redefinition is most often used with properties (as defined in UML) but may also be applied to other types.

Associations

has general : Type [1]
through association: Generalization
The general type in the Generalization rule.

[ISO 1087] concept (3.2.1) in a generic relation (3.2.21) having the broader intension (3.2.9)

[FUML] General (Where redefines is false or not defined)
[FUML] RedefinableElement.redefinedElement (Where redefines is true)

has specific : Type [1] *Redefines*: constrains:Identifiable Entity
through association: Specialization
The specific type in a generalization rule.
[ISO 1087] generic concept: concept (3.2.1) in a generic relation (3.2.21) having the narrower intension (3.2.9)
[FUML] specific
[ISO11404] A subtype is a datatype derived from an existing datatype, designated the base datatype, by restricting the value space to a subset of that of the base datatype whilst maintaining all characterizing operations. Subtypes are created by a kind of datatype generator which is unusual in that its only function is to define the relationship between the value spaces of the base datatype and the subtype.
[OWL] Union(rdfs:subClassOf, SubObjectPropertyOf, SubDataPropertyOf)

19.16 Class Multiplicity Constraint

A Multiplicity constraint constrains the number of bindings <multiplicity of> types (including property types) may have in a particular instance of the constrained type.

For a property type, The number of instances bound to a property for the set of instances bound to <with respect to> shall be limited by the minimum and maximum number of the multiplicity.

For non-property types, the multiplicity shall apply to the extent of the type as described by <classifies>.

[IDEAS] superSubType

[FUML] MultiplicityElement: Note: Multiplicity Constraint constraining a type has semantics included in to UML MultiplicityElement.

[OWL] Union(ObjectMaxCardinality, ObjectMinCardinality, ObjectExactCardinality, DataMaxCardinality, DataMinCardinality, DataExactCardinality)

Direct Supertypes

Type Constraint

Attributes

mininum number : Integer [0..1]
Minimum number in a set as constrained by a multiplicity.
[FUML] MultiplicityElement.lowerValue
[OWL] MinCardinality

maximum number : Integer [0..1]
Maximum number in a set as constrained by a multiplicity.
[FUML] MultiplicityElement.upperValue
[OWL] maxCardinality

at once : Boolean = true
When at once is true, the constraint applies for each snapshot in time but not across snapshots (e.g. a car can have at most one driver at a time). When at once is false the constraint applies across all time (e.g. a person has exactly one birth mother across all time).

is sufficent : Boolean
One of the set of sufficient conditions that will infer the type designated in <constrains>.

Associations

with respect to : Type [*]
through association: Multiplicity Reference
One or more types or properties that define the "from" side of a multiplicity.

Where with respect to is undefined and <multiplicity of> is a property, all properties that are <property of> the same structured type as <multiplicity of> shall be considered the set of <with respect to> properties. I.e. all the "other ends" of a relationship.

<with respect to> provides for complex multiplicities across n-ary situations, data structures and relationships.

multiplicity of : Type [1] *Redefines*: constrains:Identifiable Entity
through association: Multiplicity Target
The type or property that is the subject of a multiplicity constraint.

19.17 Association Multiplicity Reference

Multiplicity may be defined between things. E.g. there are 2 wheels on a motorcycle. This is most often required where relationships have more than 2 ends.
Multiplicity reference defines the "from" side of such a multiplicity (e.g. the motorcycle).

Association Ends

with respect to : Type [*] *Redefines*: constrains: Identifiable Entity
One or more types or properties that define the "from" side of a multiplicity.

Where with respect to is undefined and <multiplicity of> is a property, all properties that are <property of> the same structured type as <multiplicity of> shall be considered the set of <with respect to> properties. I.e. all the "other ends" of a relationship.

<with respect to> provides for complex multiplicities across n-ary situations, data structures and relationships.

respect of : Multiplicity Constraint [*] *Redefines*: constrains: Identifiable Entity

Multiplicity constraints using a property or type as a <with respect to> reference.

19.18 Association Multiplicity Target

Relationship defining the type a multiplicity rule applies to. Note that properties are types and may also have multiplicity constraints.

Direct Supertypes

Rule Constrains

Association Ends

multiplicity of : Type [1] *Redefines*: constrains: Identifiable Entity

The type or property that is the subject of a multiplicity constraint.

has multiplicity : Multiplicity Constraint [*] *Redefines*: constrains: Identifiable Entity

Multiplicity constraint of a type or property.

19.19 Class Property Constraint

Abstract supertype for constraints that constrain properties types.

Direct Supertypes

Rule

Associations

<<Restriction>> constrains : Property Type [1] *Redefines*: constrains:Identifiable Entity

19.20 Class Property Transitivity Constraint

A transitive property defined by <constrains> interlinks two individuals A and C whenever it interlinks A with B and B with C for some individual B.
For example "larger than" is transitive in that if Joe is larger than Sue and Sue is Larger then Sam, then Joe is larger than Sam.
[OWL] TransitionObjectProperty

Direct Supertypes

Property Constraint

19.21 Association Property Type

Relationship defining the type of a property.

Association Ends

is of type : Type [1] *Redefines*: constrains: Identifiable Entity

A required type of a thing bound to a property.
Note that the type may be inferred based on the value of <prerequisite type>.
[OWL] rdfs:range,

properties of type : Property Type Constraint [*] *Redefines*: constrains: Identifiable Entity
Properties typed by a type

19.22 Class Property Type Constraint

A property type constraint defines the type(s) of a property.
All elements bound to a property must have the type <is of type>. <is of type> may be pre-existing or inferred based on the value of <prerequisite type>.
Note that Property Type Constraint is a rule independent of the definition of a property to allow for the type of a property to be refined in a more restrictive context.

[FUML] TypedElement.type: Note: A property type constraint applied to a property has the same semantics as a UML TypedElement.

[OWL] Union(AllValuesFrom, SomeValuesFrom, DataPropertyRange, ObjectPropertyRange)). <is of type> corresponds to rdfs:Range. <constrains> corresponds to rdfs:Domain (note that in an association type or relationship type with two property types, the range will be the domain of the "opposite" property, if any).

Direct Supertypes

Property Constraint

Attributes

prerequisite type : Boolean
If true, <is of type> is a prerequisite - the bound thing must be of the given type for the property to be bound. A non prerequisite type will cause a binding to infer <is of type>, provided all prerequisite types have been satisfied.

Associations

is of type : Type [1]
through association: Property Type

A required type of a thing bound to a property.
Note that the type may be inferred based on the value of <prerequisite type>.
[OWL] rdfs:range,

19.23 Class Rule

A rule is a proposition that constrains one or more entities by limiting possible conditions or producing some effect.
Note that rules may or may not be defined in the same context that they hold within or constraint. This support the "open world assumption" that a rule may be asserted outside of the scope of the rule or what the rule is constraining.

Direct Supertypes

Proposition

Associations

constrains : Identifiable Entity [*]
through association: Rule Constrains

The entity or entities constrained by a rule.
Where a rule constrains a context, all things contextualized by the context shall be subject to the rule.
Where there are no <constrains> for a rule, the rule applies globally - to the universal context.

subsumes : Identifiable Entity [*]

through association: Rule Subsumption

When a rule subsumes another the subsumed rule will not apply (fire) if the <subsumed by> rules applies (fires).
Where rules are also patterns, a rule may specialize another which will subsume the specialized rule as well as include the generalized rule parts as parts of the specialized rule.

subsumed by : Identifiable Entity [*]
through association: Rule Subsumption

When rule is <subsumed by> another the subsumed rule will not apply (fire) if the <subsumed by> rules applies (fires).

19.24 Association Rule Constrains

Relationship defining the entity constrained by a rule. Where no constrained entity is specified, all entities are constrained with the scope of <holds within> are constrained.

Association Ends

constrains : Identifiable Entity [*]

The entity or entities constrained by a rule.
Where a rule constrains a context, all things contextualized by the context shall be subject to the rule.
Where there are no <constrains> for a rule, the rule applies globally - to the universal context.

constrained by : Rule [*]

Rules applying to an entity.

19.25 Association Rule Subsumption

Relationship defining rule subsumption. When a rule subsumes another the subsumed rule will not apply (fire) if the <subsumed by> rules applies (fires).

Association Ends

subsumes : Rule [*]

When a rule subsumes another the subsumed rule will not apply (fire) if the <subsumed by> rules applies (fires).
Where rules are also patterns, a rule may specialize another which will subsume the specialized rule as well as include the generalized rule parts as parts of the specialized rule.

subsumed by : Rule [*]

When rule is <subsumed by> another the subsumed rule will not apply (fire) if the <subsumed by> rules applies (fires).

19.26 Association Specialization

Relationship defining the specific type of a generalization constraint.

Direct Supertypes

Rule Constrains

Association Ends

has specific : Type [1]

The specific type in a generalization rule.
[ISO 1087] generic concept: concept (3.2.1) in a generic relation (3.2.21) having the narrower intension (3.2.9)
[FUML] specific
[ISO11404] A subtype is a datatype derived from an existing datatype, designated the base datatype, by restricting the value space to a subset of that of the base datatype whilst maintaining all characterizing operations. Subtypes are created by a kind of datatype generator which is unusual in that its only function is to define the relationship between the value spaces of the base datatype and the subtype.
[OWL] Union(rdfs:subClassOf, SubObjectPropertyOf, SubDataPropertyOf)

has generalization : Generalization Constraint [*]

Generalization rules for a type

19.27 Class Type Constraint

A constraint of a type, including Relationships types.

Direct Supertypes

Rule

Associations

constrains : Type [1] *Redefines*: constrains:Identifiable Entity

19.28 Association Unique Set

Relationship defining the set of properties that uniquely identify an instance of the constrained type.

Association Ends

has unique : Property Type [1..*] *Redefines*: constrains: Identifiable Entity

The set of involved properties within a type that uniquely identify an individual.

has uniqueness constraint : Uniqueness Constraint [*] *Redefines*: constrains: Identifiable Entity

Uniqueness constraints for a property.

19.29 Class Uniqueness Constraint

A constraint that, within the <constrains> type the rule applies to, the set of instances bound to the set of types in the "has unique" relation must be unique and serves to define the "identity" of each individual.
Note: Uniqueness may be used to define a "key".
[OWL] HasKey where CE (subject class expression) is <constrains> and <has unique> is Union(ObjectPropertyExpression, DataPropertyExpression)

Direct Supertypes

Type Constraint

Attributes

is primary identity : Boolean
A uniqueness constraint that can be interpreted as a "primary key", the identity of an entity.

Associations

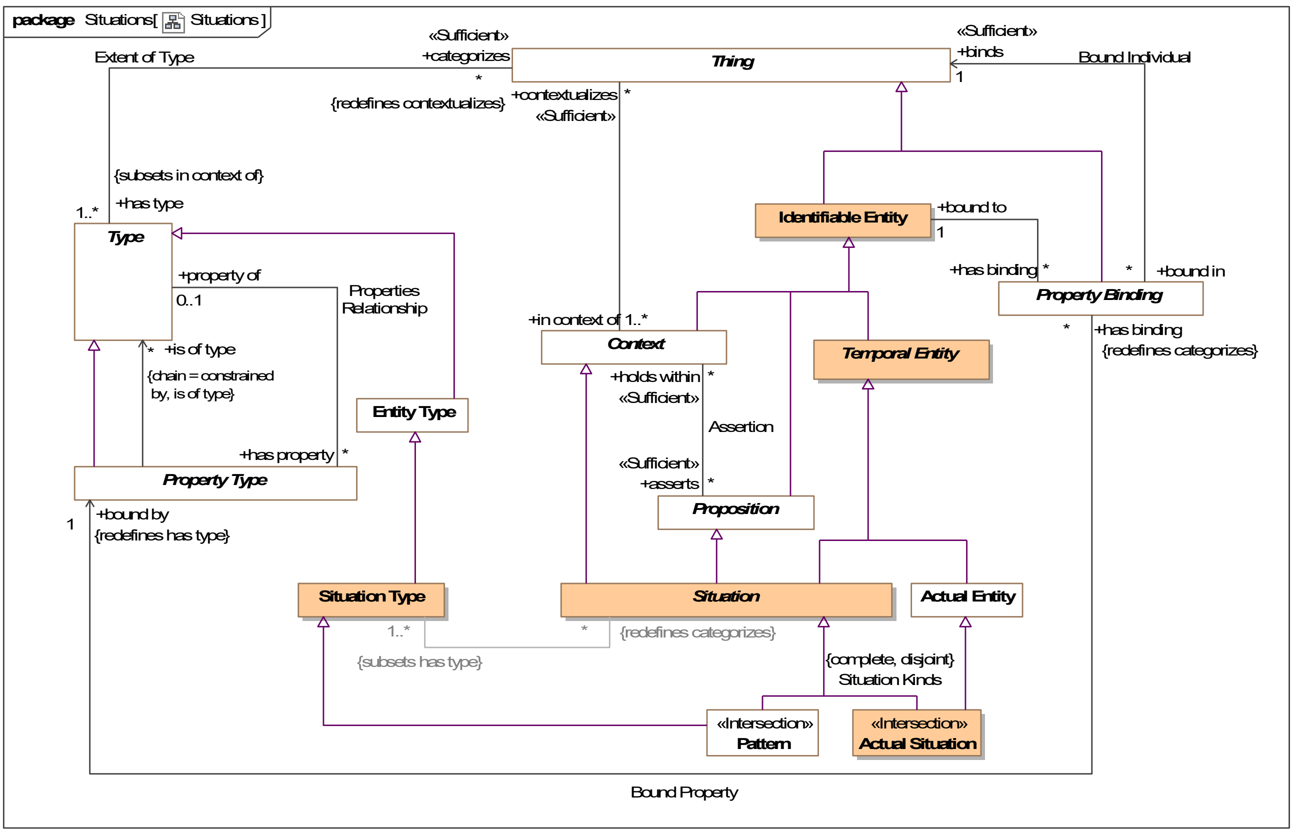
has unique : Property Type [1..*]
through association: Unique Set

The set of involved properties within a type that uniquely identify an individual.

20 SMIF Conceptual Model::Situations

A situation is a particular configuration of things and their relations including spatial, temporal, and logical connections between those things valid over a period of time. Situations form the basis of all complex, time dependent entities.

20.1 Diagram: Situations

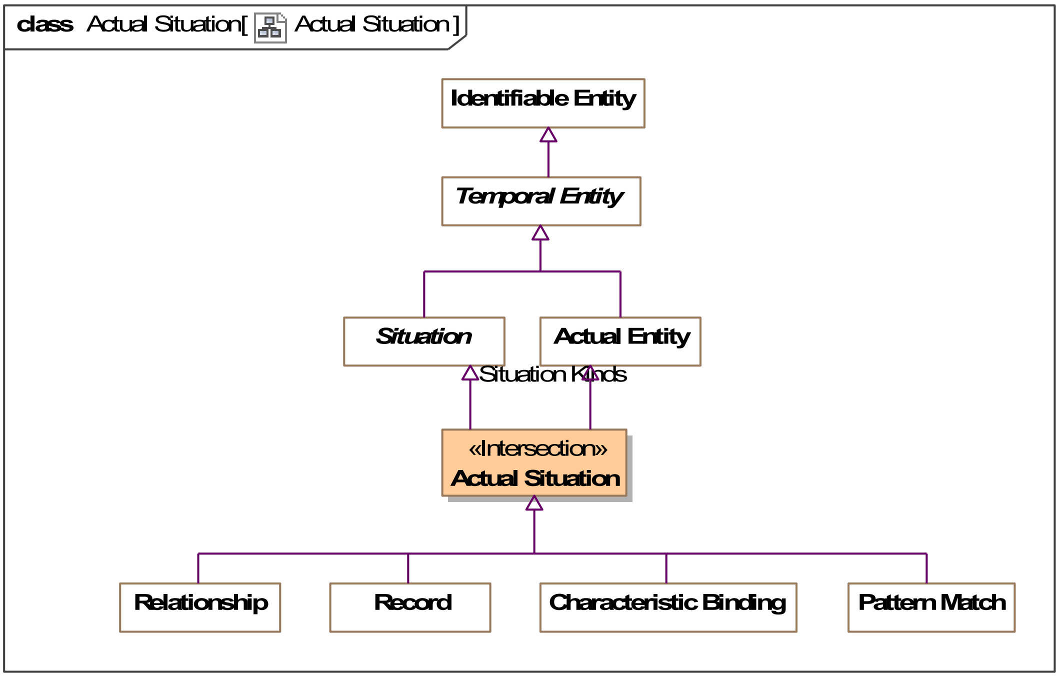


ii. Situations

20.2 Class Actual Situation <<Intersection>>

An actual situation is an individual situation that actually exists, happened in the past or may exist in some possible world, not a template or process definition. Such situations must exist for a time interval, however there are no constraints on such a time interval - from an instant to the life of the universe.

DTV: Occurrence: state of affairs that is a happening in the universe of discourse



- **Actual Situation**

Direct Supertypes

Actual Entity, Situation

20.3 Class Situation

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

[SBVR] "State of affairs"
[SOWA1999] Nexus

Direct Supertypes

Context, Lexical Scope, Proposition, Temporal Entity

Associations

<<Sufficient>><<Restriction>> : Situation Type [1..*] *Subsets*: has type:Type

matched by : Pattern Match [*]
through association: Situation Matches

Pattern matches that match the subject situation.

20.4 Class Situation Type

A situation type defines a kind of identifiable arrangement of individuals, assertions and the relations between them over a timespan. As an identifiable entity, situations may participate in other situations and relationships by being bound to properties of those situations or relationships with bindings, thus situations are “first class” entities in a SMIF model.

The roles or behaviors things (any entity or value) may play in a situation are identified as properties of the situation type.
Entity types and roles may also be situation types.
Syn. Type of a state of affairs.
A situation type may have properties such that instances, may bind things to structures based on properties.
Things may be bound to a structure (i.e. play a role in the structure) via properties. Things bound to properties of a structure may change over time, unless otherwise constrained.

[DTV] situation kind: state of affairs that may or may not happen in some possible world

Direct Supertypes

Entity Type

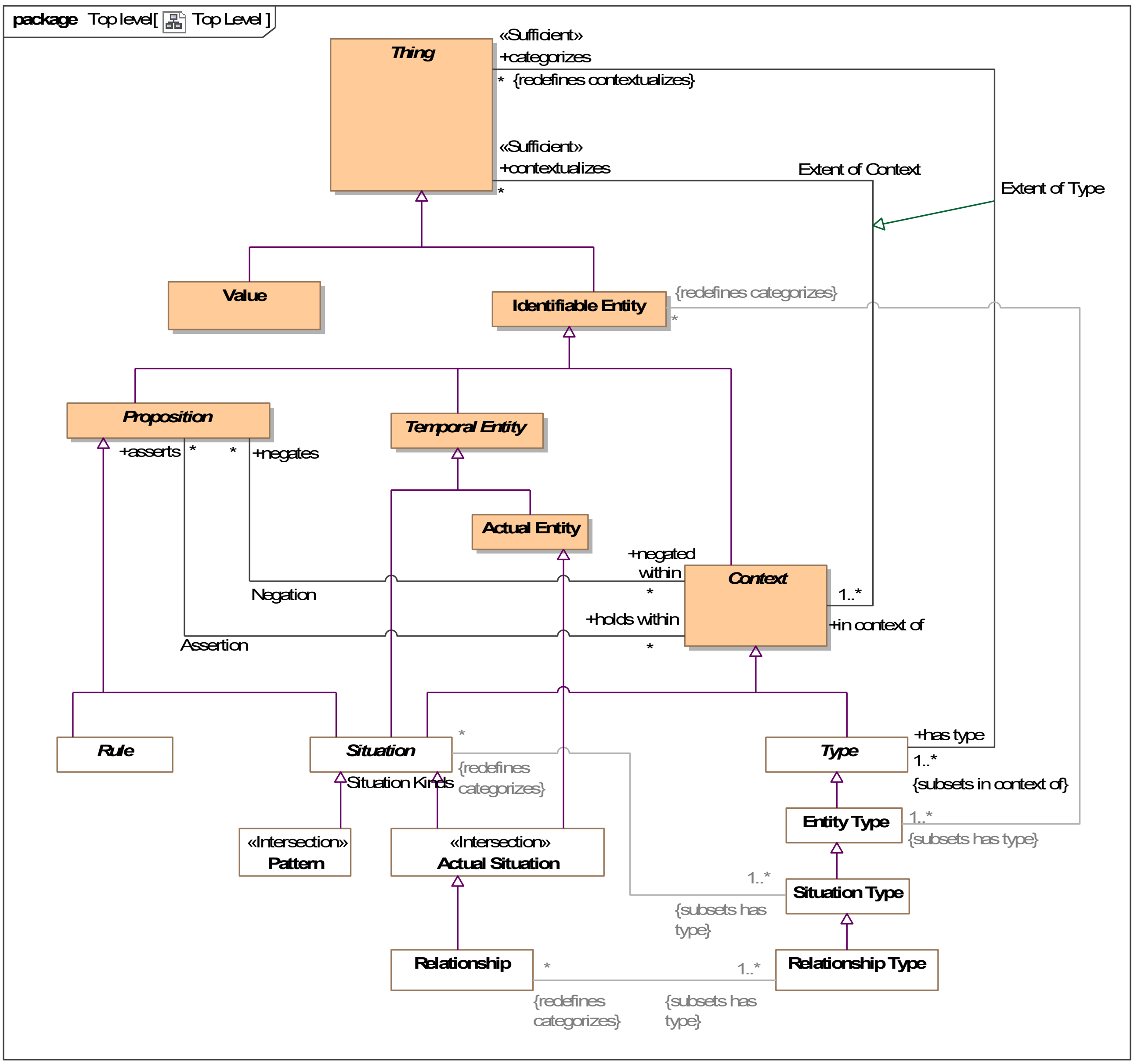
Associations

<<Sufficient>><<Restriction>> : Situation [*]   *Redefines*: categorizes:Thing

21 SMIF Conceptual Model::Top level

The top level objects provide the foundation for all objects in a SMIF model

21.1 Diagram: Top Level



jj. Top Level

Diagram showing summary of top level classes and significant subtypes.

21.2 Class Actual Entity

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

[ISO 1087] individual concept: concept (3.2.1) which corresponds to only one object

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

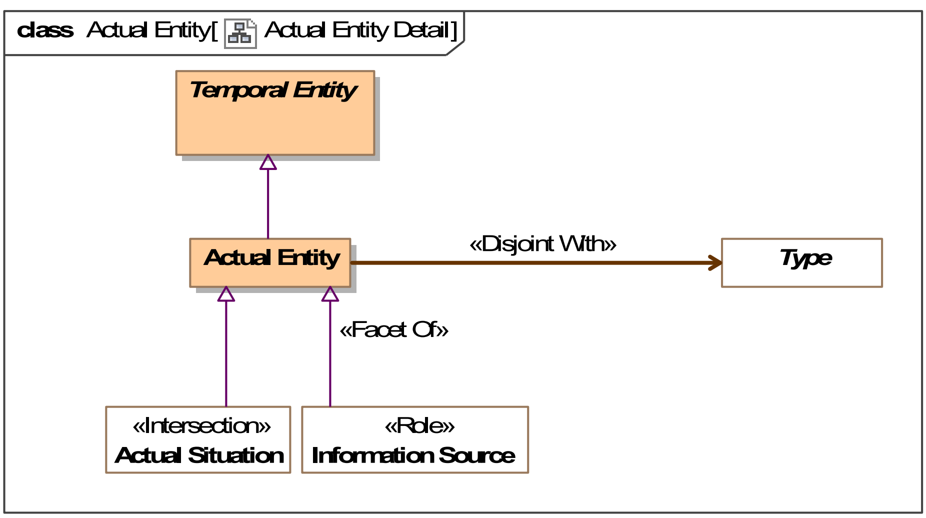
[Guizzardi] (individual concept)

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

[DOLCE] Particular: particulars are entities which have no instances

[SOWA1999] Independent. Can be considered "Actuality" when including social constructs in [SOWA1999] Physical.

[OWL] Individual



- **Actual Entity Detail**

Direct Supertypes

Temporal Entity

21.3 Association Assertion

An assertion relationship between a context and the propositions asserted within that context. The <asserts> proposition is asserted (defined as "true") for all things contextualized by the <holds within> context. Assertion of truth is not absolute, it is relative to the context. For example, something could be asserted within a context where that entire context is asserted to be false.

Assertion is transitive.
[CL] Implication
[OWL] Assertion; Any [OWL] Assertion included in a graph (All assertions in an OWL graph are asserted by the graph)

Association Ends

asserts : Proposition [*] *Redefines*: categorizes: Thing

Proposition that is asserted (must be true) for anything contextualized by a context.
As types are a context, types may assert a proposition for their instances.

holds within : Context [*] *Redefines*: categorizes: Thing

Context in which a proposition is asserted (required to be true). Anything contextualized by the context is subject to the proposition.

21.4 Class Context

A <Context> is a grouping of <contextualizes> things that are related in some way.

A <Context> also <asserts> propositions that hold for all things the context <contextualizes>, thus providing the link between an assertion and the set of things asserted. Likewise a context <negates> propositions that are false within the context.

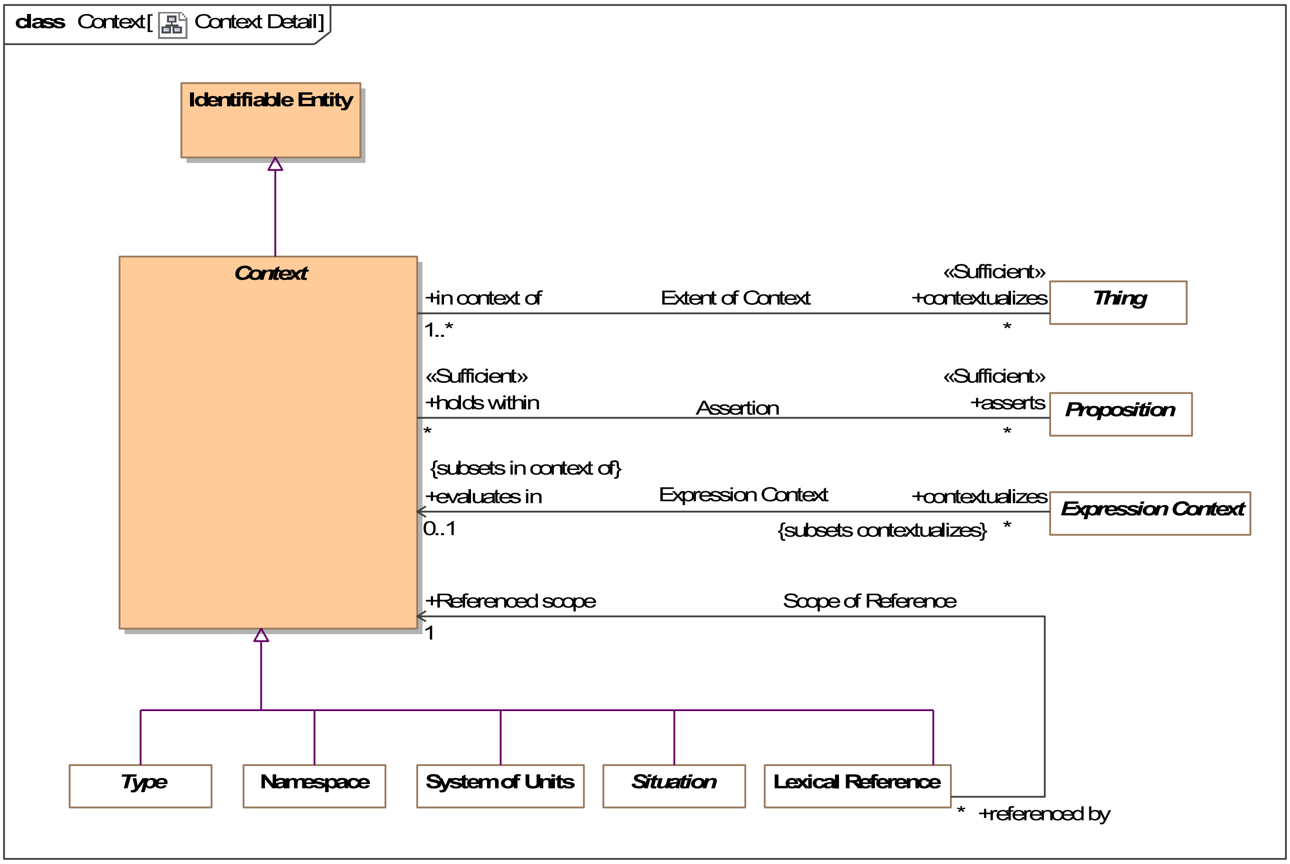
Subtypes of <Context>, such as <Type> ascribe more semantics to the context as well as the things it <contextualizes>.

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

[CL] Sort: any subset of the universe of discourse over which some quantifier is allowed to range

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

[SOWA1999] Mediating



• **Context Detail**

Direct Supertypes

Identifiable Entity

Associations

<<Sufficient>> asserts : Proposition [*]

through association: Assertion

Proposition that is asserted (must be true) for anything contextualized by a context. As types are a context, types may assert a proposition for their instances.

contextualizes : Thing [*]

through association: Extent of Context

The set of things contextualized by a <Context>, or "in" the <Context> and therefor subject to the <asserts> propositions of the <Context>.

negates : Proposition [*]

through association: Negation

Proposition that is negatively asserted (must be FALSE) for anything contextualized by a context. As types are a context, types may assert or negate a proposition for their instances.

21.5 Association Extent of Context

The association between a context and the set of things contextualized by that context, defining the extent of the context, a set.

[ISO 1087] extension: totality of objects (3.1.1) to which a concept (3.2.1)
corresponds

Association Ends

contextualizes : Thing [*]

The set of things contextualized by a <Context>, or "in" the <Context> and therefor subject to the <asserts> propositions of the <Context>.

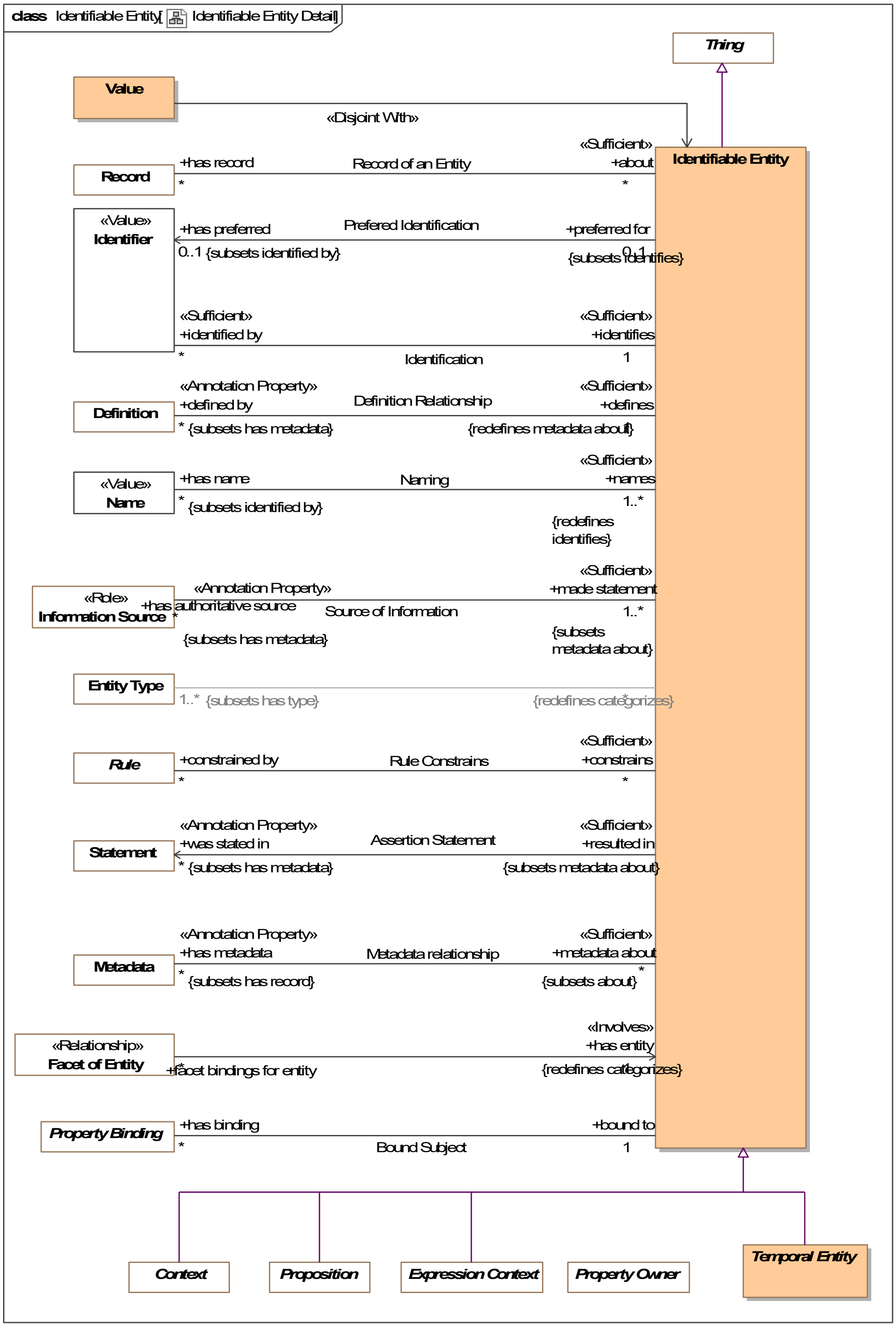
in context of : Context [1..*]

A <Context> that contextualizes a thing making what it <contextualizes> subject to the propositions referenced by <has assertion> of the context.
A thing may be <in context of> one or more context.
[FIBO] hasContext

21.6 Class Identifiable Entity

An identifiable entity is any identifiable thing other than values, this includes individuals, types, axioms, situations, speech acts, information structures, etc.
Identifiable entities always have some kind of identity and may have identifiers. Note that identity is an abstraction that may have representation in models as any number of identifiers, also known as a "sign".

[OWL] Entity type (Implied in section [OWL] 5.8) as an instance of rdfs:Class



- **Identifiable Entity Detail**

Direct Supertypes

Thing

Associations

<<Sufficient>> was stated in : Statement [*] *Subsets*: has metadata:Metadata
through association: Assertion Statement

Metadata representing the speech act, document or other record where a statement captured in a model was made.
[OWL] rdfs:isDefinedBy

has preferred : Identifier [0..1] *Subsets*: identified by:Identifier
through association: Prefered Identification

Default identifier to use for an entity.
Where multiple identifiers are preferred in differing context any method for selecting the most preferred identifier is implementation specific and not specified by this standard.
[FUML] NamedElement.name: Note: An Identifier that is <preferred for> an entity is equivalent to the name of a named element.

<<Restriction>> : Entity Type [1..*] *Subsets*: has type:Type

<<Annotation Property>> defined by : Definition [*] *Subsets*: has metadata:Metadata
through association: Definition Relationship

An informal description of something.
[FIBO] hasDefinition
[UML] comment
[FUML] ownedComment

<<Sufficient>> identified by : Identifier [*]
through association: Identification

An identifier for an <Entity>.
[FIBO] hasDenotation

<<Annotation Property>> has metadata : Metadata [*] *Subsets*: has record:Record
through association: Metadata relationship

Metadata associated with (data about the information concerning) the subject entity.
[OWL] AnnotationProperty, annotationValue of Annotation Assertion

has name : Name [*] *Subsets*: identified by:Identifier
through association: Naming

A human meaningful name for an entity.
[FIBO] hasName: that by which some thing is known; may apply to anything

[OWL] rdfs:label

has record : Record [*]
through association: Record of an Entity

A record about something.

constrained by : Rule [*]
through association: Rule Constrains

Rules applying to an entity.

<<Annotation Property>> has authoritative source : Information Source [*] *Subsets*: has metadata:Metadata
through association: Source of Information

Metadata representing the authority behind a statement - who or what made a statement captured in a model.

has binding : Property Binding [*]
through association: Bound Subject
Bindings asserted for properties within a situation.

21.7 Association Negation

An assertion relationship between a context and the propositions negated (FALSE) within that context. The <negates> proposition is asserted as FALSE for all things contextualized by the <negated within> context. Assertion or negation of truth is not absolute, it is relative to the context.
[CL] Negation+Implication

Association Ends

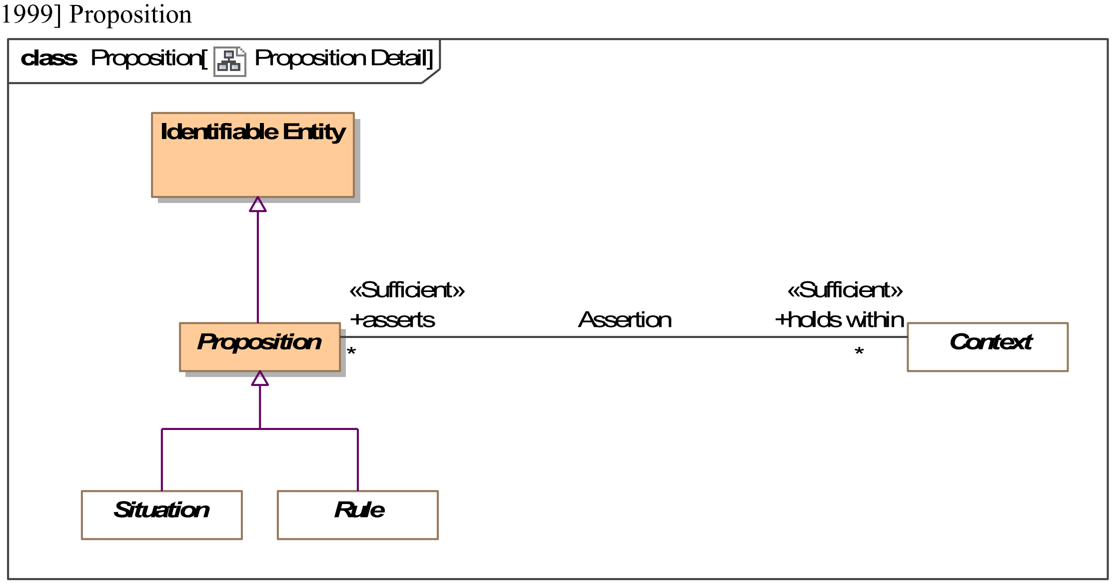
negates : Proposition [*]
Proposition that is negatively asserted (must be FALSE) for anything contextualized by a context. As types are a context, types may assert or negate a proposition for their instances.

negated within : Context [*]
Context in which a proposition is negated (required to be FALSE). Anything contextualized by the context is subject to the proposition.

21.8 Class Proposition

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

[SBVR] the state of affairs is posited by the proposition and if the state of affairs were actual, the proposition would be true
[CL] Sentence: unit of logical text which is true or false, i.e. which is assigned a truth-value in an interpretation
[SOWA1999] Proposition



- **Proposition Detail**

Direct Supertypes

Identifiable Entity

Associations

<<Sufficient>> holds within : Context [*]
through association: Assertion

Context in which a proposition is asserted (required to be true). Anything contextualized by the context is subject to the proposition.

negated within : Context [*]
through association: Negation

Context in which a proposition is negated (required to be FALSE). Anything contextualized by the context is subject to the proposition.

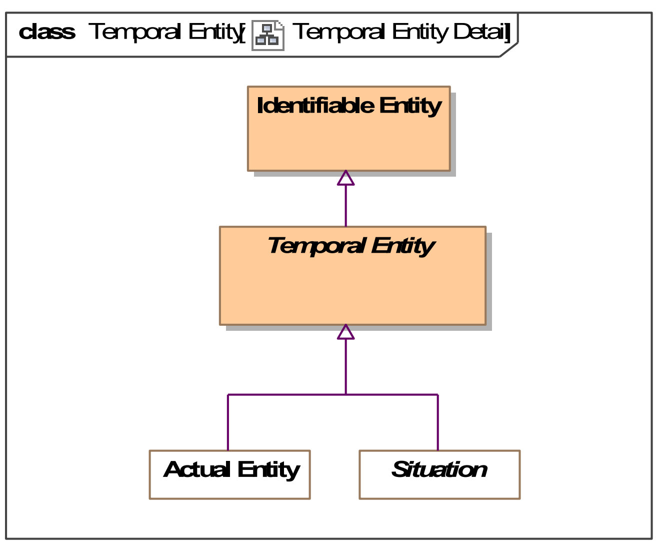
qualified within : Proposition Variable [0..1]
through association: Qualified Proposition

21.9 Class Temporal Entity

A temporal is anything that has a timespan. Temporal things may have temporal relationships with other temporal things.

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

[SOWA1999] Continuant



- **Temporal Entity Detail**

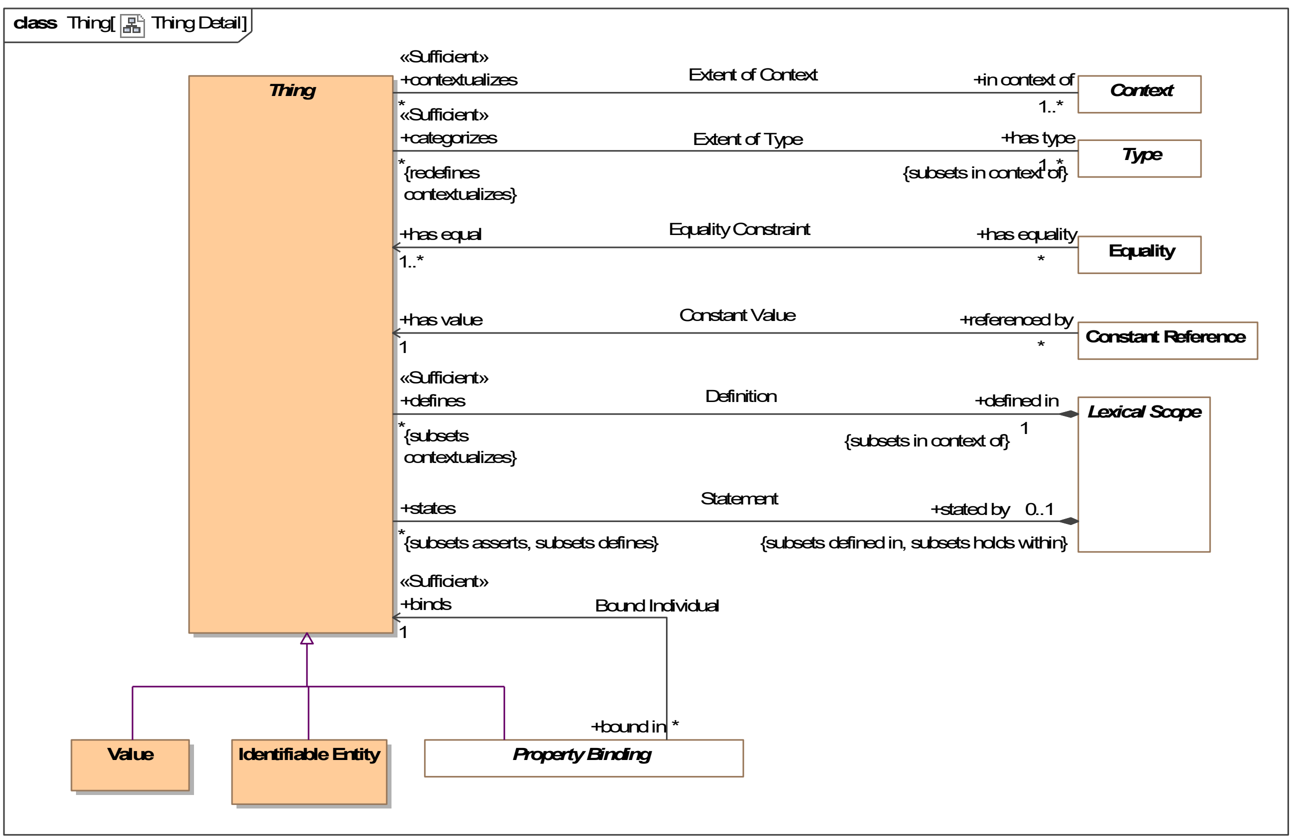
Direct Supertypes

Identifiable Entity

21.10 Class Thing

Any thing or value that does or may exist in any possible world. Thing is the supertype of all types and may therefore participate in unbounded relations.
Instances of Thing are referred to as "a thing" in this model.

[IDEAS] Thing
[OWL] Thing
[ISO 1087] object: anything perceivable or conceivable
[FIBO] Thing
[Guizzardi] Thing
[FUML] Element
[SOWA1999] "T"
[OWL] rdfs:Resource



• **Thing Detail**

Associations

defined in : Lexical Scope [1] *Subsets*: in context of:Context
through association: Definition

Lexical scope defining model elements.
[UML]owner

in context of : Context [1..*]
through association: Extent of Context

A <Context> that contextualizes a thing making what it <contextualizes> subject to the propositions referenced by <has assertion> of the context.

A thing may be <in context of> one or more context.
[FIBO] hasContext

has type : Type [1..*] *Subsets*: in context of:Context
through association: Extent of Type

A type that holds for something.
Things may have multiple types and these types may change over time.
The <categorized> thing satisfies the constraints of the <has type> type.
[FIBO] isClassifiedBy
[OWL] rdf:type

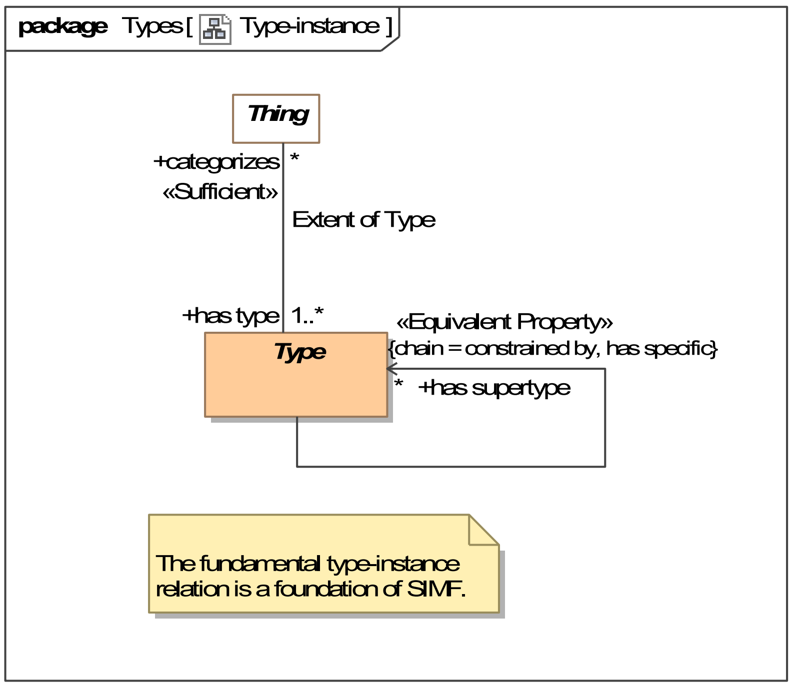
stated by : Lexical Scope [0..1] *Subsets*: defined in:Lexical Scope holds within:Context
through association: Statement

<stated by> is a lexical scope that both defines and asserts a model element.

22 SMIF Conceptual Model::Types

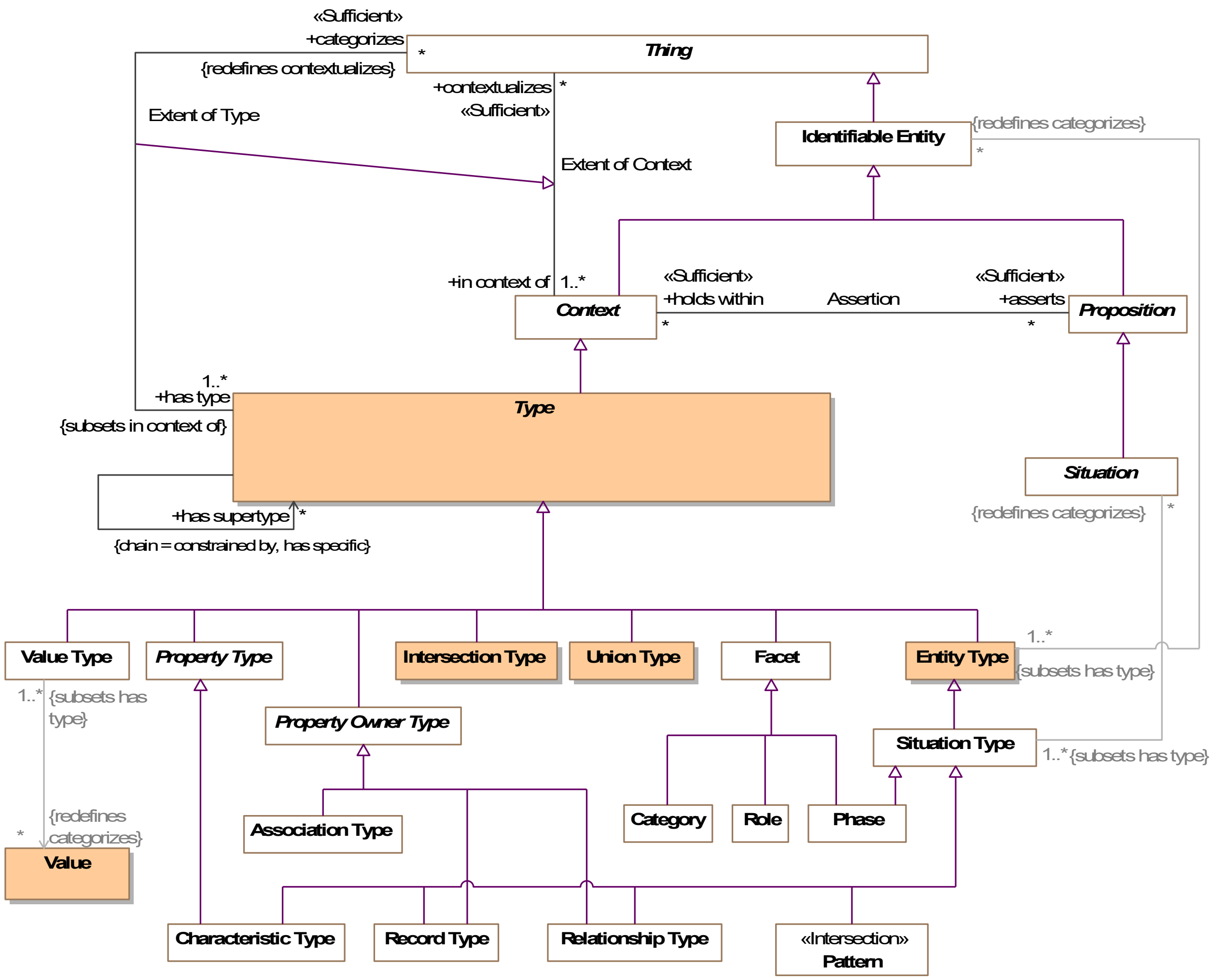
Types provide for ways to categorize anything based on what it is, the roles it plays or the phases it may be in. Something may be categorized by any number of types (multiple classification assumption).

22.1 Diagram: Type-instance



kk. Type-instance

22.2 Diagram: Types



II. Types

22.3 Class Entity Type

A type of an identifiable entity. All concrete entity instances must have at least one entity type. Entity type may be mixed with other types to fully define an entity.

[FUML] Classifier

[Guarino1994] Substantial or Pseudo-Sortal (Substantial being concrete)
[Guizzardi] A Rigid Universal.
(Rigid Universal): A universal G is rigid (or modally constant) iff for any w,w′∈W 3. extw(G) = extw′(G) Putting definitions 4.1 and 4.3 together, we have that for any rigid universal G the following is true 4. ext(G) = extw(G), for all w ∈W A rigid universal is one that applies to its instances necessarily, i.e., in every possible world. Every substance sortal G is a rigid universal.

[OWL] rdfs:Class (as Entity Type does not include values). However, non=primitive values are typically represented as rdfs:Class

Direct Supertypes

Type

Associations

<<Restriction>> : Identifiable Entity [*]   *Redefines*: categorizes:Thing

22.4 Association Extent of Type

The relation between a type and the things that type categorizes, the instances which defines the extent of the type, a set.

[IDEAS] typeInstance: A couple that asserts that a Thing is a member of a Type.

[Guizzardi] (Extension functions): Let W be a non-empty set of
possible worlds and let w ∈W be a specific world. The extension function extw(G) maps a universal G to the set of its instances in world w. The extension function ext(G) provides a mapping to the set of instances of the
universal G that exist in all possible worlds, such that ext(G) = U w∈W w ext (G)

[OWL] ClassAssertion

Direct Supertypes

Extent of Context

Association Ends

categorizes : Thing [*]   *Redefines*: categorizes: Thing

The set of things described by a type, the "extent" of the type.
The thing a type <categorizes> is subject to the <has assertion> propositions of the type.
[FIBO] classifies

has type : Type [1..*]   *Redefines*: categorizes: Thing

A type that holds for something.
Things may have multiple types and these types may change over time.
The <categorized> thing satisfies the constraints of the <has type> type.
[FIBO] isClassifiedBy
[OWL] rdf:type

22.5 Class Intersection Type

An intersection is a type that has an extent which is the complete intersection of the extents of all supertypes. Intersection is a stronger statement than a subtype as a subtype may not be a complete intersection.

[MathWorld] The intersection of two sets A and B is the set of elements common to A and B. This is written A intersection B, and is pronounced "A intersection B" or "A cap B."

Direct Supertypes

Type

22.6 Class Type

A <Type> is a categorization of any thing based on specific criteria. The specific criteria may or may not be formalized in a model.
A <Type> <categorizes> a set of <Thing>s which comprises the "extent" of the type.
A <Type> is a <Context> where the things it <categorizes> are <in the context> of the <Type>.

[IDEAS] Type: A set (or class) of Things.
[ISO 1087] general concept: concept (3.2.1) which corresponds to two or more objects (3.1.1) which form a group by reason of common properties
[FIBO] Classifier: a standardized classification or delineation for something, per some scheme for such delineation, within a specified context
[FUML] Type
[CL] Type:: logical framework in which expressions in the logic are classified into syntactic or lexical categories (types) and restricted to apply only to arguments of a fixed type
[Guarino1994] Discriminating Predicate
[OWL] Union(rdfs:Class, rdfs:Datatype)

Direct Supertypes

Context, Lexical Scope

Associations

has supertype : Identifiable Entity [*] *Redefines*: categorizes:Thing

Supertypes(s) of a type as defined by generalization rules.

All statements made about the supertype are true for the subtype. The extent (categorizes) of the subtype is a subset of the extent of the supertype.

Has supertype is a a derived association based on generalization rules.

categorizes : Thing [*] *Redefines*: contextualizes:Expression Context
through association: Extent of Type

The set of things described by a type, the "extent" of the type.
The thing a type <categorizes> is subject to the <has assertion> propositions of the type.
[FIBO] classifies

has property : Property Type [*]
through association: Properties Relationship

A property of a structured type such that there may be bindings of a thing to instances of the structured type with reference to the property which defines the semantics of the bound thing withing the context of the structure.
[FUML] feature
[UML] memberEnd. attribute (of classifier).

<<Sufficient>> asserts pattern : Pattern of Type [0..*] *Subsets*: asserts:Proposition
through association: Subject of Pattern Relationship

A pattern asserted for all instances of a type. Where the pattern includes parts, the type defines a composition.

has covering : Covering Constraint [*]
through association: Covering Constraint

Covering constraints of a type.

has specialization : Generalization Constraint [*]
through association: Generalization

Specialization rules for a type.

has multiplicity : Multiplicity Constraint [*]   *Subsets*: constrained by:Rule
through association: Multiplicity Target

Multiplicity constraint of a type or property.

properties of type : Property Type Constraint [*]
through association: Property Type

Properties typed by a type

recording types : Record Type [*]
through association: Subject of Record Type

Record for a thing.

has generalization : Generalization Constraint [*]   *Subsets*: constrained by:Rule
through association: Specialization

Generalization rules for a type

22.7 Class Union Type

A Union is a type that has an extent which is the complete union of the extents of all types that specialize the Union.

[FIBO] Logical Unions

[MathWorld] Given two sets A and B, the union is the set that contains elements or objects that belong to either A or to B or to both. We write A È B

[OWL] ObjectUnionOf(ObjectUnionOf, DataUnionOf)

Direct Supertypes

Type

23 SMIF Conceptual Model::Values

The values package defines the concepts of values and quantities expressed in units.

Values may be differentiated from entities in that values have no independent lifetime or "identity" other than the value its self. E.g. the number 5 "just is" and can't be changed. Properties and relations referencing values can, of course, change but the values are constant.

The failure to properly express units in data models often results in errors, inefficiencies and risk. Translation and federations between models, schema and data sources that is not cognizant of the units used would be even more error prone and risky. For example, what does “Speed limit 50” mean? For these reasons the SMIF language provides specific support for specifying quantity kinds and unit types in conceptual, logical and physical models. The SMIF mapping rules may then perform the appropriate unit conversions.

The foundation of information specification in SMIF at all levels is the type system. Types specified for all properties and relations involving values must match the types of the related values. The concepts of units and values as defined in "VIM" [JCGM 200-2008] is used as the basis for defining the types used in SMIF to guarantee type safety of quantities across different representations. Since many existing models and schema do not include well defined units some effort may be required to find and then specify the implicit units based on documentation, SME interviews or inspection of data or source code. It is recommended that the units used by external models and schema be determined prior to attempting federation and integration of information based on those models or schema.

VIM [JCGM 200-2008] concepts of quantities and units

VIM defines

- quantity: property of a phenomenon, body, or substance, where the property has a magnitude that can be expressed as a number and a reference [ed. to a unit]
- kind of quantity (kind): aspect common to mutually comparable quantities
- measurement unit (unit): real scalar quantity, defined and adopted by convention, with which any other quantity of the same kind can be compared to express the ratio of the two quantities as a number

SMIF concepts of quantities and units

SMIF uses the VIM concepts to define "quantity values" and types to capture the quantity kind and unit. Types are defined for each Unit. The goals for this type based approach are:

- That it is clearly grounded in semantics as defined in VIM
- That a type may be used to specify the range of a property or relation involving unit based values.
- That a quantity value (e.g. 5 grams) be representable as a simple number with a type.
- That there is a clear type hierarchy starting with a representationally independent type in a conceptual model (e.g. mass) that can be further specialized to a specific unit in a logical model (e.g. grams) and further specialized to be represented by a physical data type (e.g. “double”).
- That external models and schema may have unit specifications asserted without changing the schema.
- That a quantity of an entity be able to be referenced without a specific quantity value being known (e.g. John’s weight).
- That systems of units such as [ISO-80000] or [OMG QUDV] (A part of SysML) be able to be directly referenced as the definition of a unit.

SMIF defines three types to realize the above goals: Quantity Kind, Unit Type, Base Unit Type. SMIF also defines Quantity Values, which are instances of unit types.

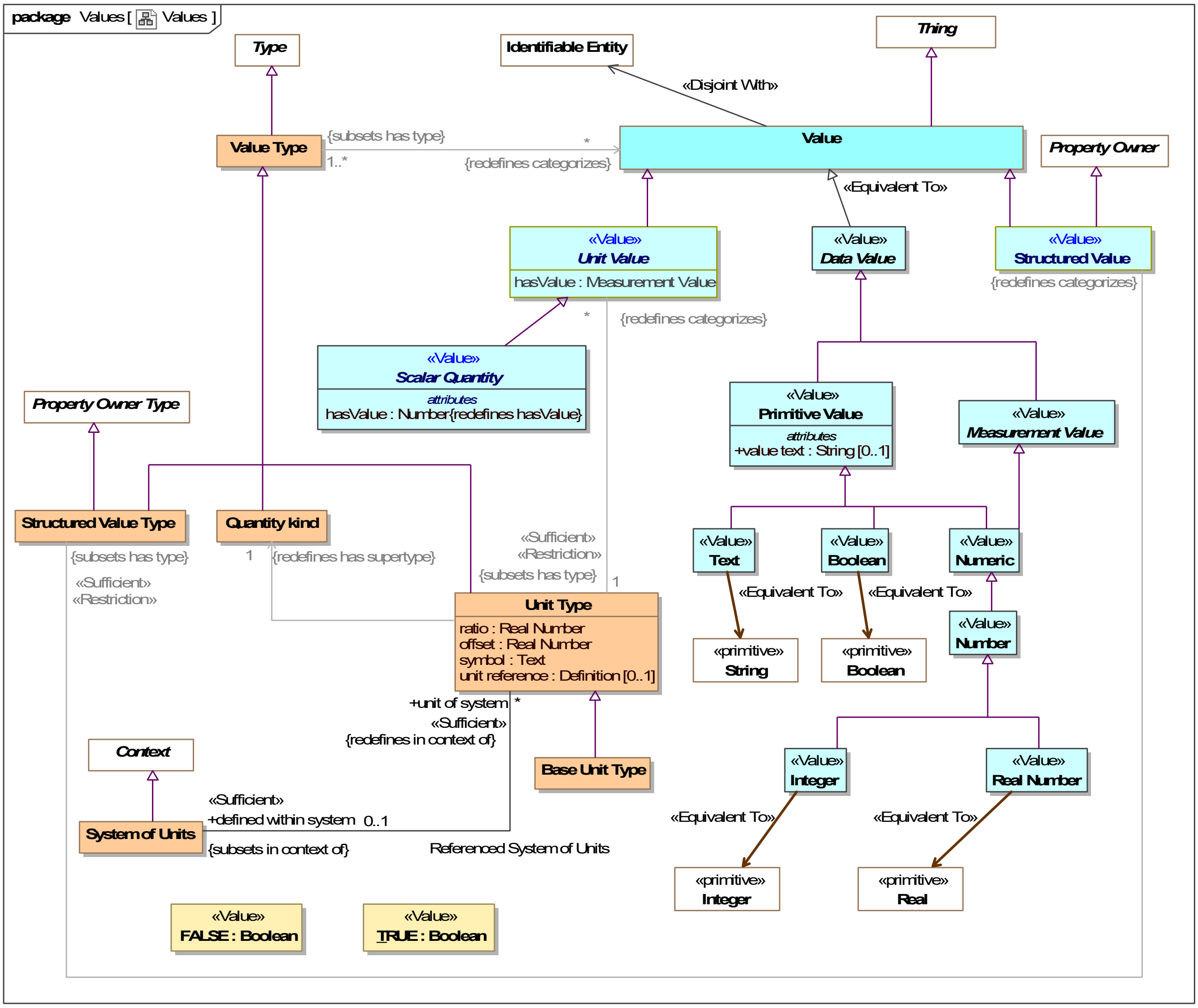
In VIM a quantity has a magnitude that is expressed as a number and a reference. The SMIF quantity value is the numeric value of such a quantity where the reference is specified by the “unit reference” property of the quantity value’s type. The quantity value’s type is a “Unit Type”. The Unit type has attributes for converting a unit to a base unit, a symbol and a unit reference. Based on VIM the unit reference may be “a measurement unit, a measurement procedure, a reference material, or a combination of such” and is specified with a description that contains reference information. In summary, the reference of a SMIF quantity value is determined indirectly through its unit type. A quantity value has

exactly one unit type and exactly one Quantity Kind. A quantity value expressed in any unit of the same quantity kind may be converted to any other unit of the same quantity kind.
This type-based sapproach allows specification of a property at the conceptual (quantity kind) logical (unit type) or physical (unit type with a numeric type) levels. Such specifications use the same type-based approach used for other aspects of the models. Given this information a SMIF implementation may correctly and reliably convert between compatible types regardless of representation. Please see the specification of the value types, attributes and relationships for more detail.

Example:

5. A specification for a road segment has a property “Speed limit”.
6. The type of this property in a reference conceptual model is “Speed:Quantity Kind”.
7. A unit “Kilometer per Hour:Unit Type” is defined as a subtype of “Speed:Quantity Kind” with a “unit reference” of “[ISO-80000.4] Kilometer per Hour”. Note that quantity kinds and unit types would normally be defined in reference models that correspond to a “system of units”.
8. Miles per hour is also defined as a subtype of Speed.
9. A physical schema defines “Speed-KPH: Integer”.
10. A SMIF mapping rule maps “Speed limit” to “Speed-KPH” and asserts a type of “Kilometer per Hour” on the “Speed-KPH” end.
11. A data file defines a road “Route One” with a speed limit of 100:KPH-Int.
12. When converted to a U.S. application this speed limit of route one can be viewed as 62:MPH-Int.

23.1 Diagram: Values



mm. Values

23.2 Class Base Unit Type

One unit type of a quantity kind may be marked as the base unit within a system of units. The base unit provides the basis for conversions between units of the same quantity kind. The base unit always has a ratio of one and an offset of zero.

Type of a [JCGM 200:2008] measurement unit that is adopted by convention for a base quantity

[FIBO] (type of) Base Unit: a measurement unit that is defined by a system of units to be the reference measurement unit for a base quantity

There ma be at most one base unit for a quantity kind within a system of units.

Direct Supertypes
Unit Type

23.3 Class Quantity kind

[JCGM 200:2008] A Quantity Kind is an aspect common to mutually comparable quantities represented by one or more units. Units with a common quantity kind may be algorithmically converted to any other unit of that quantity kind. e.g. temperature.

Quantity kinds are a supertype of unit types which are then a type of all quantity values, Quantity values are mutually comparable with all other quantity values categorized by the same quantity kind.

[FIBO] QuantityKind: a categorization type for “quantity” that characterizes quantities as being mutually comparable

[DOLCE] Quality Space

Direct Supertypes
Value Type

23.4 Association Referenced System of Units

Relationship between a system of units and the set of unit types defined within that system.

Direct Supertypes
Extent of Context

Association Ends
defined within system : System of Units [0..1] *Subsets*: constrained by:Rule

The system of units in which a unit is defined and is the basis for ratio and offset.

By default the system of units is "si": http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=30669

unit of system : Unit Type [*] *Subsets*: constrained by:Rule

Unit type defined within a system of units

23.5 Class Scalar Quantity <<Value>>

Direct Supertypes
Unit Value

Attributes
hasValue : Number

The value of a quantity that, when multiplied by the unit defined in a subtype of quantity kind, specifies a measurement value such as 3 Meters.

23.6 Class Structured Value <<Value>>

A value that may have sub-elements (owned properties) defined as "structure property type".

Direct Supertypes

Property Owner, Value

Associations

<<Sufficient>><<Restriction>> : Structured Value Type *Subsets*: has type:Type

23.7 Class Structured Value Type

A structured value type is a type of value that has parts represented as properties - also used for "data types" and forms.

Direct Supertypes

Property Owner Type, Value Type

Associations

<<Sufficient>><<Restriction>> : Structured Value *Redefines*: categorizes:Thing

23.8 Class System of Units

[JCGM 200:2008] A set of base units and derived units, together with their multiples and submultiples, defined in accordance with given rules, for a given system of quantities.

[FIBO] SystemOfUnits: a set of measurement units associated with a system of quantities, together with a set of rules that assign one measurement unit to be the base unit for each base quantity in the system of quantities and a set of rules for the derivation of other units from the base units

Direct Supertypes

Context

Associations

<<Sufficient>> unit of system : Unit Type [*] *Redefines*: in context of:Context
through association: Referenced System of Units

Unit type defined within a system of units

23.9 Class Unit Type

A Unit type is a type of a quantity value referencing a specific unit. A Unit Type a required type of a property representing a quantity.

Each quantity value has a reference as defined by the "unit reference" property of the quantity value's type.

[JCGM 200:2008] A Unit is a real scalar quantity, defined and adopted by convention, with which any other quantity of the same quantity kind can be compared to express the ratio of the two quantities as a number. e.g. Degrees Centigrade, Miles.

Each unit type represents refinement of a quantity kind using generalization and is thus substitutable for that quantity kind. Typically quantity kinds are used in conceptual models and unit types in physical or logical models.

Unit types may only subtype quantity kinds or other units.

Note that unit types are not units, but the type of quantity values expressed with respect to a common unit as defined in [JCGM 200:2008].

[IDEAS] MeasureCategory: A MeasureType whose members are recognized types of MeasureInstance.

Direct Supertypes

Value Type

Attributes

ratio : Real Number
The multiplier by which to multiple the referenced unit to convert to the base unit within a system of units.

offset : Real Number
The difference between zero in the referenced unit and zero in the base unit after the ratio is applied within a system of units.

symbol : Text
The accepted symbol for the unit referenced by the unit type

unit reference : Definition [0..1]
The unit reference is the reference to a unit shared by all quantities values that are instances of a unit type.

[JCGM 200:2008] A reference can be a measurement unit, a measurement procedure, a reference material, or a combination of such. For magnitude of a quantity.

Typical references include ISO 8000 and OMG QUDV.

Associations

: Quantity kind [1] *Redefines*: has supertype:Type

<<Sufficient>><<Restriction>> : Unit Value [*] *Redefines*: categorizes:Thing

<<Sufficient>> defined within system : System of Units [0..1] *Subsets*: in context of:Context
through association: Referenced System of Units
The system of units in which a unit is defined and is the basis for ratio and offset.

By default the system of units is "si": http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=30669

23.10 Class Unit Value <<Value>>

A unit value is a numeric magnitude with a unit type that may be used as the value of a quantity property as defined by [JCGM 200:2008]. The reference of the quantity is defined by the "unit reference" property of the Unit Type.

e.g. 5cm is an instance of the unit type "Centimeter"

Each unit value has exactly one UNit Type as a type.

In a physical model a quantity value must have a type that specifies its unit (e.g. "Gram"). The magnitude shall be expressed using "hasValue"

[JCGM 200:2008] A quantity is a property of a phenomenon, body, or substance, where the property has a magnitude that can be expressed as a number and a reference.

Note: A quantity as defined here is a scalar. However, a vector or a tensor, the components of which are quantities, is also considered to be a quantity.

[IDEAS] ScaleMapping: A CoupleType whose members are all the couples linking MeasurePoints to RealNumbers. The CoupleType (i.e. the set of couples) represents the scale.

[FIBO] QuantityValue: number and measurement unit together giving magnitude of a quan-tity

[Guizzardi] (quale): A point in a n-dimensional quality domain

Direct Supertypes

Value

Attributes

hasValue : Measurement Value

The value of a quantity that, when multiplied by the unit defined in a subtype of quantity kind, specifies a measurement value such as 3 Meters.

[OWL] rdf:value restricted to abstract quantity

Associations

<<Sufficient>><<Restriction>> : Unit Type [1] *Subsets*: has type:Type

23.11 Class Value

A Value is an atomic. immutable piece of information without a specific lifetime or identity independent of the value. Values include numbers, strings and other atomic "primitive" data. Values also include structured values, which are immutable.

In UML values may be defined by the name of an instance specification with a value type.

[IDEAS] Representation: A SignType where all the individual Signs are intended to signify the same Thing.

[ISO11404] The identification of members of a datatype family, subtypes of a datatype, and the resulting datatypes of datatype generators may require the syntactic designation of specific values of a datatype.

[OWL] data values

Direct Supertypes

Thing

23.12 Class Value Type

A type categorizing values where a value is an atomic piece of information without a specific lifetime or identity independent of that value. Values include numbers, strings and other atomic "primitive" data.

[IDEAS] RepresentationType: A Type that is the Powertype of Representation.

[FUML] DataType

[ISO11404] datatype: set of distinct values, characterized by properties of those values, and by operations on those values

[OWL] rdfs:Datatype (Note that some values are represented as OWL classes)

Direct Supertypes

Type

Associations

<<Restriction>> : Value [*]   *Redefines*: categorizes:Thing

7 SMIF UML Profile (Normative)

This section defines the UML profile for conceptual modeling and mapping. In order to improve UML's suitability for modeling real-world concepts, this profile interprets standard UML with semantic extensions, as detailed below:

7.1 Concept Modeling Profile Semantics

A concept[CC1] model can be expressed in UML with the concept modeling profile. The profile defines the interpretation of UML concepts used, extends UML concepts with "stereotypes" and makes some UML semantics more specific to concept modeling. While there are some extensions, every effort is made to use "generic UML" class diagrams, as they are well understood and supported. [CC2] It only provides stereotypes to extend UML to make concept models more expressive. For example, without complex OCL constraints, UML normally has no way to express that, in the context of some class, some values must be of some type, all values must be of some type, or that the property chain *has father* • *has brother* is equivalent to the property *has uncle*. Other examples of extensions include *Roles* and *Phases* to describe how entities may be classified in different context or over time. These extended notions are introduced here, in subsequent sections. Readers are referred to the UML specification and the many books and courses on UML for an in-depth treatment of generic UML.

This section is intended to define the semantics of UML used in this specification to represent concept models. [CC3] The subset of UML used for concept modeling is primarily that known as "Class models", the most commonly used part of UML. Our scope further narrows what we utilize to exclude behaviors and methods – elements used for object oriented design. Those elements may be present, but they are ignored for the purposes of concept modeling.

The[CC4] goal of a concept model is to unambiguously define durable conceptualizations of the real or an imaginary [CC5] world. One can think of a concept model as describing a "subject area", which can be as small or large as desired (e.g., the concepts across the entire financial industry, or merely the concepts within one organization). Concepts are, of course, the foundation of a concept model. Concepts are how we think about the world. They are modeled as combinations of classes, datatypes, enumerations, associations, and properties. A related goal of a concept model is to be as non-technical and business-friendly as possible. That means that names for concepts should contain spaces rather than what's called "CamelCasedWords" or "Underscore_Separated_Words". It is the job of the transformations to convert those names into lexemes that are acceptable to more technical tooling.

A concept model owned by subject matter experts is more durable than a data model or logical information model designed with a particular system in mind. Thus, one definition of concepts and properties can be represented by multiple logical information models, each optimizing for different technical goals. Note that there are multiple interpretations of "logical model" that span from almost conceptual to near a data schema. Conceptual models can be used to help federate any of these levels of abstraction.

A concept model is not an information or data model. When someone think about concepts, they think about real-world things, not data structures or even natural language text about those things. These real-world concepts become the pivot points around which we define and relate the many terms, languages and data structures that describe those things. For example, every Person *has biological mother* one Person, which is essential to being a Person. Such concepts provide criteria that narrow the definition of *what* a Person concept *is*, it does not specify that a system should store every person's mother. What is contextual is our knowledge of that or our need to know it, which is the subject of an information model. For another example, it would be reasonable for a concept model to assert that an eye has a measurable visual acuity, but not to define how visual acuity will be represented within a computer as bits and bytes, or how often visual acuity will be stored within a database. Such technical concerns should be elaborated in a data model, which has elements that can be well defined by a concept model. Note, however, that things such as tables and columns *are* valid concepts in their own right – as "data things", but they are different from the real-world concepts they might represent.

Concept models can be modular. A concept model may refer to things in a number of other concept models. This is useful for refining another organization's concept model, separately maintaining overlapping concepts between organizations or disciplines, or more easily managing smaller subject areas.

A concept model consists of a network of concepts with a simple essential structure. That structure is the definition of classes, relations between them and their characteristics. Classes represent abstractions of "things" in our world – including physical things like trees or people and "made up" things like agreements.

Other concepts connect those things - the relations between things are UML associations that have properties. Things have characteristics such as weight or color. Things can also have properties that are attributes of a class. This basic network of classes, associations, and properties forms the foundation of the concept model and defines the conceptual framework and vocabulary of a domain. Each of these concepts may have names, which form the vocabulary of a domain of interest. Various assertions are then made about these concepts and their connections that further refine the semantics of those concepts – multiplicities of relationships, specializations between concepts, essential properties of things, etc.

One of the fundamental ways we understand and organize concepts is their arrangement into hierarchies, where general concepts are specialized to form more specific concepts within a specific context or with more specific characteristics. A concept model can arrange all the fundamental elements into conceptual hierarchies using generalization relationships. In contrast, another kind of hierarchy is a structural data hierarchy – where data elements contain other data elements. As the purpose of a concept model is not representing data, data hierarchies are not part of a concept model, they are typically part of logical information models that represent concepts. To allow for the many viewpoints that can exist for any concept, a concept can be in many generalization hierarchies at the same time.

The following section defines how UML with conceptual extensions is used to represent the foundational network of concepts using classes, associations, and properties. Additional constraints, expressed as rules, are then attached to this basic framework to enhance semantic expression and the ability of automation to federate and analyze information about those concepts.

7.1.1 Classes

Classes specify, or classify, a set of things, according to some set of rules or understanding. Classification is the essential mechanism of conceptualization we use. Classes specify a set of things belonging to that class – this is called the class's *extent*. Each element of the class is an *instance* of that class –it is something the class classifies. Classifications may be arranged in hierarchies.

In the UML concept model, a class is diagrammed as a box with a name at the top. In some cases, a definition is also shown next to the box in a "note" form.

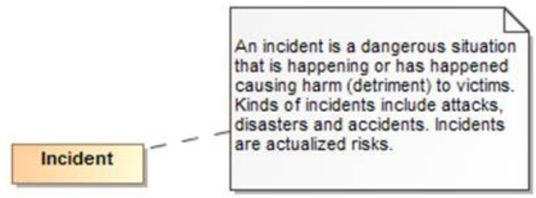


Figure 5 Example of a Class

The above example shows the class “Incident” and its definition. It should be noted that a class is *a way* to classify something. It is natural to classify something multiple ways. For example, we may classify a situation as also being a danger or, to someone else, an opportunity to do harm. This is different from many technology models (e.g. Java) that only allow something to be classified in one way and the classification is fixed. *The basic assumption of a concept model is that unless specified otherwise, something may be classified in any number of ways and those classifications may change over time*.

7.1.2 Instances

While not usually used in the definition of the concept model, instances can also be shown in UML and are utilized to illustrate examples or to define well known instances, like the “United States of America”. Since the model is conceptual, instances of classes are proxies, or “signs”, for the “real thing” in the world – not data about them or other technology artifacts.

Instances are also shown as a box, but have a “:” separating the name of the instance from its classes.

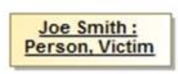


Figure 6 Instance Example

The above example shows a information about an instance named “Joe Smith” that is classified as a “Person” and a “Victim”.

7.1.3 Class Generalization

Since Aristotle, classes have been arranged in hierarchies – from most general concepts to more specific ones. In UML this is shown as a Generalization – an arrow with a solid line from the more specific concept to the more general. The more general class is known as the *Superclass* (or *Supertype*) and the more specific the *Subclass* (or *Subtype*). Generalization has some specific semantic rules:

- Everything that is true about the superclass must be true about all its subclasses
- The extent of the subclass is a subset of the extent of the superclass
- All properties and associations that apply to instances of a class also apply to instances of all its subtypes

In a concept model, a class may have any number of superclasses or subclasses. In contrast, some technologies (Like XML Schema) limit the number of superclasses to one.

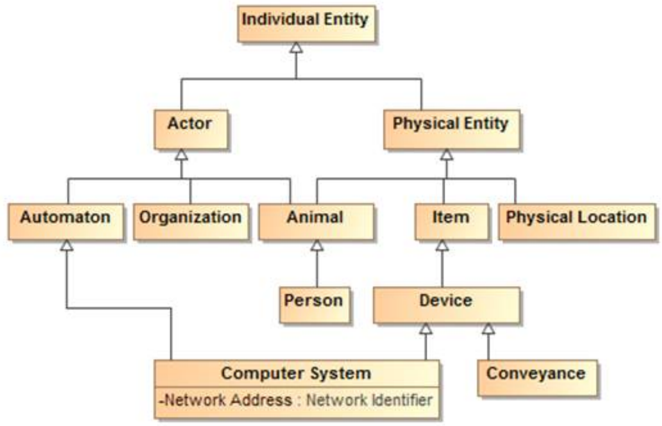


Figure 7 Class Hierarchy Example

The above example shows a class hierarchy with multiple levels.

Note that all properties and associations defined for all superclasses of a class apply to that class. For that reason a complete understanding of a class and its potential properties must include such superclasses.

A generalization is a subsumption relationship between a more general class and a more specific class. Every instance of the specific class is also an instance of the subsuming general class. Because of this subsumption relationship, the specific class inherits all of the necessary conditions of the more general classifier.

For a simple example, if we define "Futsal Team" as a subclass of "Soccer Team", then the set of individuals in "Futsal Team" must be a subset of the set of individuals in "Soccer Team".

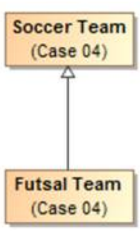


Figure 8 Simple Generalization Example

There are four variations on generalization described in the following subsections. The first variation corresponds to the example above: overlapping and incomplete subclasses. That variation is the default in both UML and concept modeling.

7.1.3.1 Overlapping and Incomplete Subclasses

This variation is the default in both UML and in concept modeling. In this variation, an instance can be a member of the superclass and / or any number of subclasses. In this sense, the classification of instances is "incomplete"—sometimes an instance is classified by one or more specific subclasses, and sometimes it is not.

For example, the diagram below shows four instances (represented by white diamonds). One is an instance of "Manufacturer", one is an instance of "Windshield Manufacturer", one is an instance of "Car Manufacturer", and one is an instance of both "Windshield Manufacturer" and "Car Manufacturer".

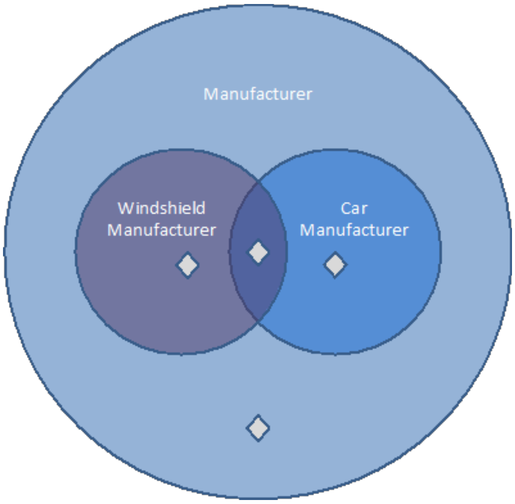


Figure 9 An example of incomplete subclasses

In both standard UML and in concept modeling, incomplete and overlapping subclasses are shown with either no notation, or with the equivalent notation {incomplete, overlapping} near the generalization arrow.

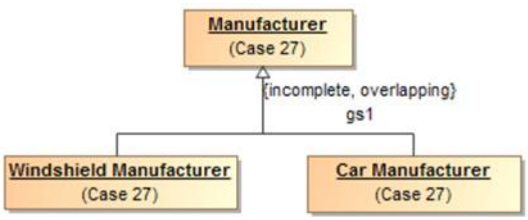


Figure 10 Incomplete and overlapping subclasses in standard UML notation

7.1.3.2 Disjoint and Incomplete Subclasses

This variation means that an instance can only be classified by at most one of the disjoint classes. Disjoint classes cannot have any overlap in their instances.

The diagram below shows three instances. One is an instance of "Cat", one is an instance of "Dog", and one is an instance of "Animal". An instance classified as both "Cat" and "Dog" is impossible because there is no overlap between the two classes. In the most basic terms, an instance of a "Cat" cannot be an instance of a "Dog", and vice versa.

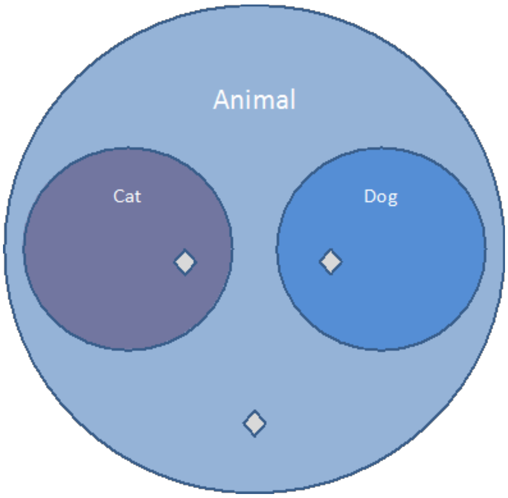


Figure 11 Disjoint Subclasses

The following diagram shows an example of disjoint subclasses in standard UML notation. It shows that "Dog", "Cat", and "Mouse" are all subclasses of "Animal". In addition, the standard UML {incomplete, disjoint} notation declares all of the subclasses to be incomplete and disjoint. Intuitively, an instance of the subclass "Dog" is an instance of the superclass "Animal", but it cannot also be an instance of the "Cat" or "Mouse" subclasses. It is incomplete because there can be many more kinds of animals.

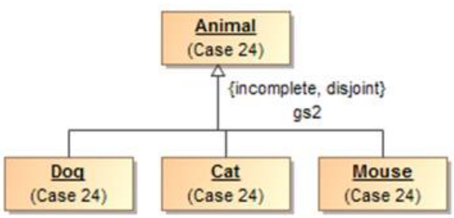
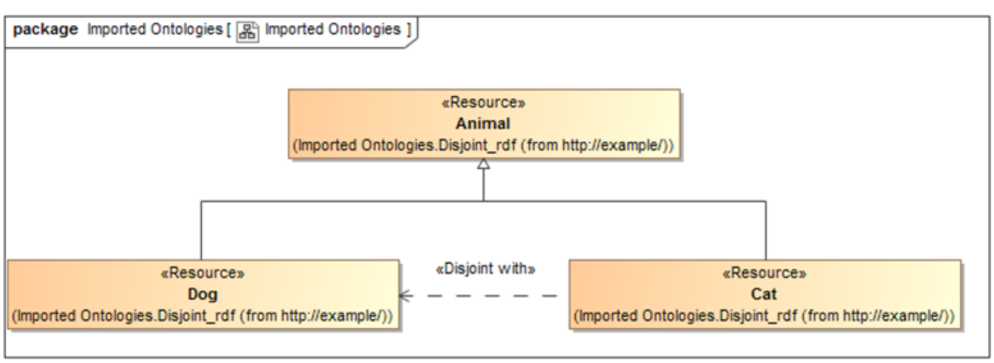


Figure 12 Incomplete and disjoint subclasses in standard UML notation

The profile also supports a dependency stereotyped as «Disjoint With» to specify that anything can be disjoint, even if they are not subclasses of a common super type. disjoint subclasses. For example, the class Animal has three disjoint subclasses, Cat and Dog. [CC6]



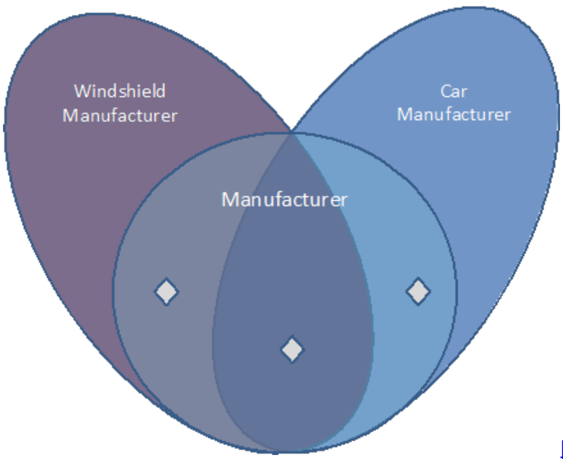
[CC7]

Figure 13 Alternative «Disjoint With» Stereotype

7.1.3.3 Complete and Overlapping Subclasses

This variation means that an instance can only be classified by at least one of the subclasses; it cannot be classified by only the superclass. Keep in mind that an instance of a subclass is indirectly an instance of a superclass at the same time.

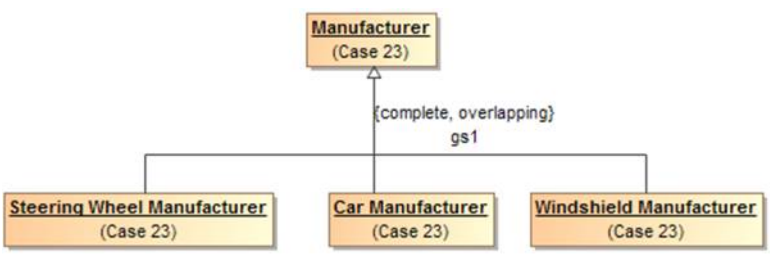
For example, the following diagram shows three instances. One is an instance of "Windshield Manufacturer", one is an instance of "Car Manufacturer", and one is an instance of both "Car Manufacturer" and "Windshield Manufacturer". Note that there can be no instance of "Manufacturer" that is not also an instance of one of the subclasses.



[CC8]

Figure 14 An example of complete subclasses

The diagram below shows an example of complete and overlapping subclasses in standard UML notation. The diagram shows that "Steering Wheel Manufacturer", "Car Manufacturer", and "Windshield Manufacturer" are all subclasses of "Manufacturer". In addition, the standard UML {complete, overlapping} notation declares that the subclasses are complete and overlapping.



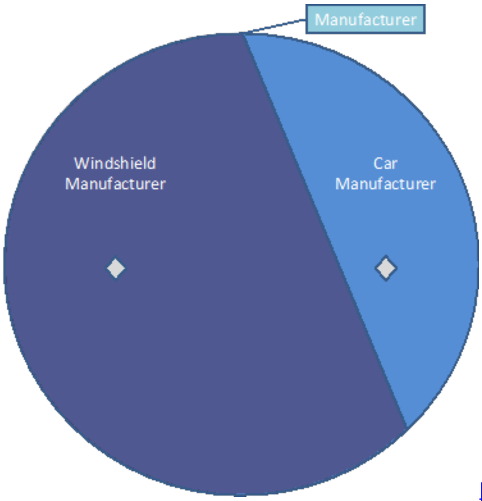
[CC9]

Figure 15 Complete subclasses in standard UML notation

7.1.3.4 Disjoint and Complete Subclasses

This variation means that an instance can only be classified by one of the subclasses. The instance cannot be classified as only the superclass, and it cannot be classified by two subclasses at the same time.

For example, in the subsequent diagram, two instances are shown. One is an instance of "Windshield Manufacturer", and one is an instance of "Car Manufacturer". There can be no instance of "Manufacturer" that is not also an instance of one of the subclasses, and there can be no instance that is classified as both a "Windshield Manufacturer" and a "Car Manufacturer" at the same time.



[CC10]

Figure 16   Disjoint and complete instances

The diagram below shows an example of disjoint and complete subclasses in standard UML notation. The diagram shows that “Steering Wheel Manufacturer”, “Car Manufacturer”, and “Windshield Manufacturer” are all subclasses of “Manufacturer”. In addition, the standard UML {complete, disjoint} notation declares that the subclasses are complete and disjoint.

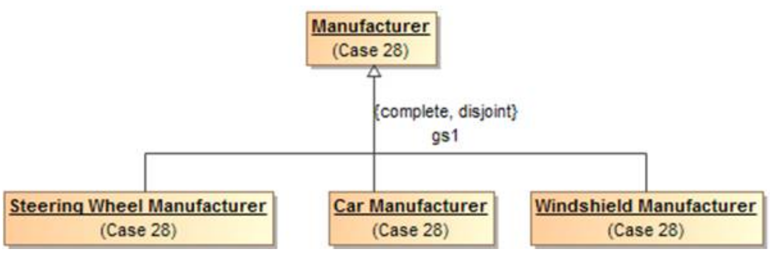


Figure 17   Disjoint and complete subclasses in standard UML notation

7.1.4 Properties

Properties represent qualities inherent in something, such as size, weight or a time. Each property has a “type” for the kind of value that represents that quality. A property is a characteristic that an individual can have, or, as explained in a subsequent section, an individual *must* have to qualify as a particular concept.

Most properties are relations between concepts, usually expressed as a verb phrase, such as "Heart *comprised of* Chamber" or "Geographic Region *identified by* Address". This kind of property is generally drawn as a UML association end, as part of a UML association.

Some properties are relations with data types, such as a standard UML Date, usually expressed as a prepositional phrase, such as "Person *born on* Date" or a noun phrase, such as “Person *birth date* Time Point”. This kind of property is generally drawn as a UML attribute, within an attribute compartment of the most general classifier that can have that quality.

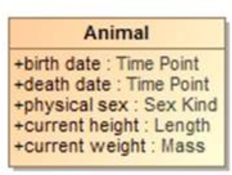


Figure 18 Example of Properties

The above example shows that an animal has the qualities of birthdate, death date, physical sex, height and weight. Note that these is no assumtion that these qualities may be known, required or that different data sources may or may not agree on them – just that an animal has these qualties. Instances of properties are facts about the entity they describe. In concept models, attributes are only used for qualities, never to relate different entities.

A much smaller number of properties represent metadata, usually expressed as a noun phrase, such as "Anything *description* String" or "Anything *see also* URI". To represent metadata, this profile provides a stereotype called «Annotation Property» that can be applied to a standard UML property in a concept model.

Note that because every class ultimately specializes the special class «Anything», when that <<Anything>> has properties, those properties can be used by instances of any class. Moreover, classes or subclasses can have constraints on the values of properties that only hold from that class and below in the generalization hierarchy. See subsequent sections for further explanation.

7.1.5 Associations

Associations describe facts about how entities are related. Associations are shown as lines between the classes that have related instances. At each end of the line is an "association end" property – the association end describes how the instances of the class on the far end relate to those of the near end. If there are limits to how many instances may be related, these are also shown. Since an association has at least two ends, the association may be read in any direction, but is the same "fact". The properties involved are considered "inverse properties". [CC11] The association end properties are typically verbs or verb phrases, but in some cases, such as when an association is reified as a class, the association ends can become noun phrases[CC12] . In either case the name denotes the intent of the class *at the other end of the line*.

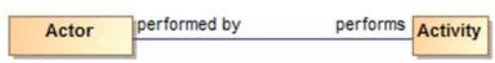


Figure 19 Association Example

The above example says that there are relations between actors and activities such that the *actor performs the activity* and the activity is performed by the actor. These are considered two ways to "read" the same fact. Like any fact, relations may be true for some period of time or in some specific situation.

As can be seen in the example the ends of associations are typically verb phrases which can then be read as <the actor> performs <the activity>. In other cases the ends are nouns in which case they represent a role being played. If a role were used above instead of "performed by" it could read: <activity> has performer <actor> (the *has* in this sentence being implied by english grammar).

This combination of classes and associations with ends forms the basis for nouns and verbs common to human language. The terms used for the nouns and verbs should be both consistent with their semantics and resonate with stakeholders – sometimes this is a bit of a challenge.

In some cases the ends of the relation are sufficent to define it, in other cases it makes more sense to give the association a name and its own definition. Associations and association ends, like classes, can be part of a hierarchy.

Note that unspecified multiplicities are interpreted as unconstrained: having a minimum cardinality of 0 and a maximum cardinality of "*".

7.1.6 Property and association end hierarchies

Like class hierarchies, attributes and association ends (we will just call both properties from now on) can also be arranged in hierarchies of more or less specific properties. In UML, property hierarchies are represented using either "Subsets" or "Redefines". [CC13] What a property subsets or redefines is shown next to its name in in the diagram (Note that by convention this is not shown on summary diagrams, only the primary definition of the property). If a property completely subsumes the other in a particular context it uses a "Redefines" – that is the redefining and redefined properties have the same set of values. If the more general concept can also be used in the context a "Subsets" is used.

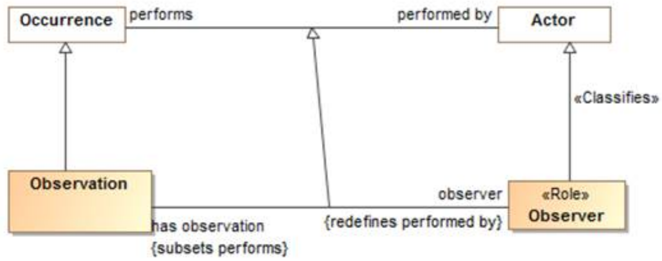


Figure 20 Example of Association End Hierarchy

The above example shows that the “has observation” and “observer” properties are specializations of the “performs” and “performed by” concepts. The property “observer” redefines “performed by” – that is, an Observation always has an observer, never a “performed by” any other kind of actor. Likewise “has observation” specializes “performs” but an instance of Observer can perform other activities as well. Note the generalization between the associations is implied, but is shown in this example for clarity.

Where a redefined or subset property has no name, it is an indication that the property type and/or multiplicity is merely constrained in some way. No new properties or associations are actually defined for a constraint (more on this below).

7.1.7 Association Classes

In a concept model any “fact” may have properties. Of particular importance is the “provenance” of the fact – where the fact came from and thus how much it can be trusted. Facts can also be time-bound, true for some period or only valid within some context. Where an association may have additional specific properties or may participate in other relationships, an “association class” is used. As implied by its name, an association class has both the properties of an association and the properties of a class. More complex associations between things use association classes. An association class is diagrammed as an association line and a class box with a dashed line between the association line and its class. While the association line and box may seem somewhat visually distinct – they are the “same concept”.

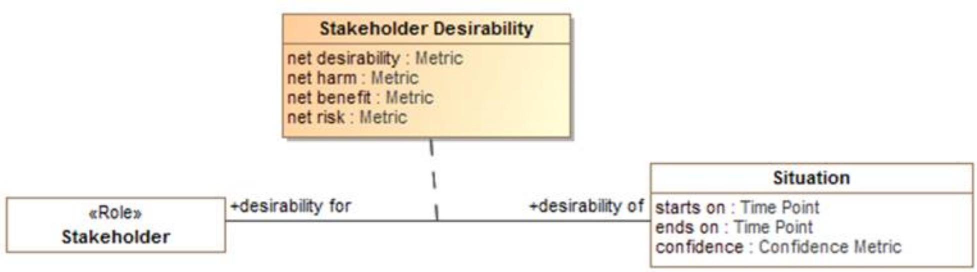


Figure 21 Association Class Example

The above example shows the “Stakeholder Desirability” relation. Between any situation and any stakeholder there can be some metrics as to how much that stakeholder desires or wants to avoid that situation. The Stakeholder Desirability association class represents these as properties of the association: net desirability, net harm, net benefit and net risk – which can all be poitive or negative reflecting a benefit or harm, respectivly.

7.1.8 Annotation

This profile provides a way to comment on any element using *annotations*. One can annotate classes, properties, and models using an open-ended system of *annotation properties*. An annotation property defines information about the model (metadata), not about the subject domain. A property can be made an *annotation property* using the «Annotation Property» stereotype on a UML property[CC14] .

Every «Annotation» is a textual value for an «Annotation Property». An annotation describes some subject using an annotation property and a (usually textual) value. An annotation should specify a tagged value called "value for" that refers to an «Annotation Property».

For example, the following diagram illustrates several UML comments stereotyped with «Annotation»

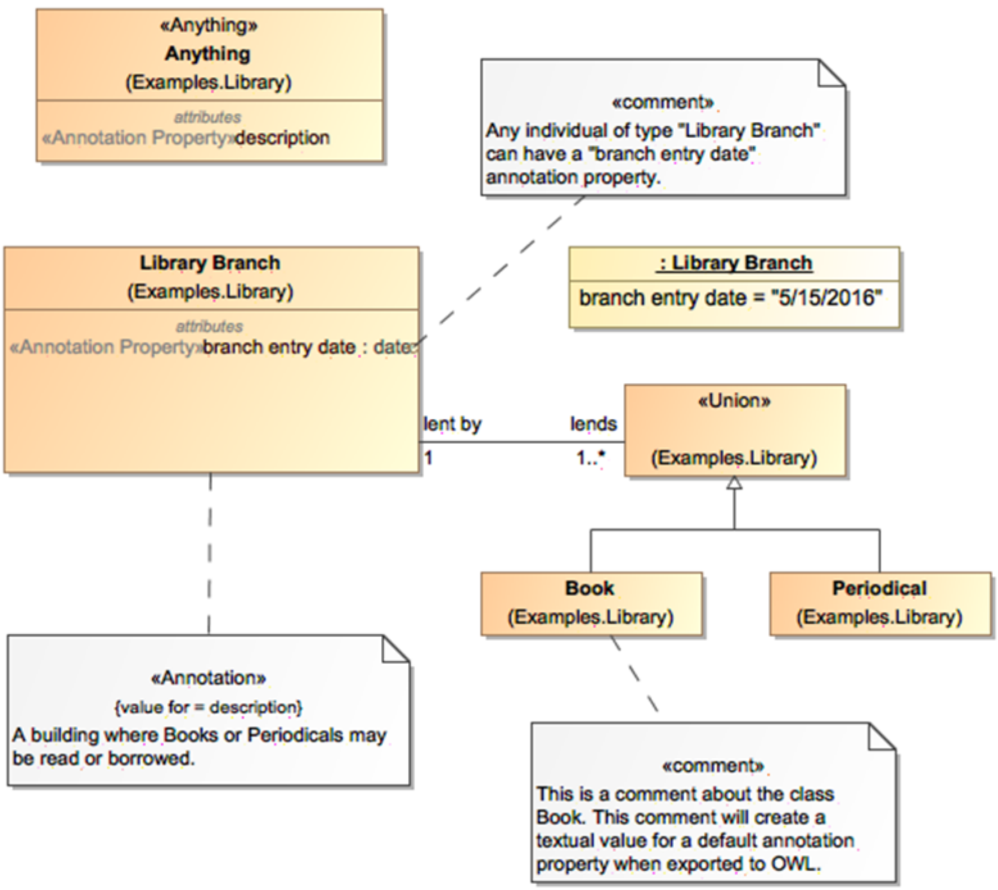


Figure 22 Annotation Examples[CC15]

7.1.9 Specific kinds of classes

There are additional concept modeling specific stereotypes documented in the reference section that further define the semantics of a class. Some of these stareotypes are very important for understanding the concept model and are further explained here. These are roles, phases and quantity kinds.

7.1.9.1 Anything

The stereotype «Anything» can be applied to any class to make it special[CC16] . Every such special class is equivalent to one topmost class (⊤) of which all other classes are subclasses. Thus, a property of a class marked as «Anything» is inherited by all subclasses. In addition, while the name of a such a marked class is irrelevant, consistently naming such classes “Anything” in all concept models avoids any confusion with normal classes.

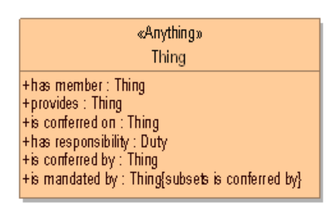


Figure 22 «Anything» Example

7.1.9.2 Union

A «Union» is a class that has an extent (set of instances) which is equivalent to the union of the extents of all types that specialize the Union (Subclasses). Specializing types shall include subtypes and types that realize the union. The union can be either named or unnamed. When it is unnamed, it can only be used at the domain or range of a property.[CC17]

Note: UML realizations are included to support unions across external models because UML generalization cannot be used across external models due to the ownership of generalization.

An anonymous union class always implies a complete subclass generalization. [CC18]

The following diagram states that an instance of a Person may have a value of type Cat or Dog for the *cares for* property. The diagram also states that an instance of a Cat or a Dog may have a value of type Person for the *cared for by* property.

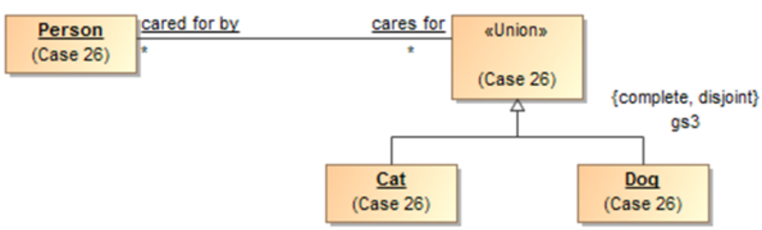


Figure 23 A union class

7.1.9.3 Intersection

An «Intersection» is a class that has an extent (set of instances) equivalent to the intersection of the extents of all supertypes. Intersection is a stronger statement than a subtype, as a subtype may be a subset of the intersection. An instance of all the supertypes implies an instance is also an instance of the intersection type.

For intersection, The SMIF profile considers UML generalization and UML realization equivalent. This is due to ownership and legacy considerations in UML. Generalization is the preferred representation.

Note: Realizations are included to support unions across external models. UML generalization can not be used across external models due to the ownership of generalization.[CC19]

7.1.9.4 Facets, Roles, Phases and <<Facet Of>>

Some types may be considered the “fundamental” type of something that is essential to its being and identity for its entire lifetime; this is the default assumption of most classes. Other types classify something in a specific context or for a period of time, SMIF calls these “Facets”. Examples of facets are “Roles” and “Phases” that something may have over its lifetime. The facets an instance is classified with may change over time and may be only valid within a particular context or viewpoint. Facets are defined with a <<Facet Of>> generalization to another type, the type of thing that can be so classified. For example, “Policeman” can classify a “Person”.

Context specific types such as Roles and Phases are classifications and expected to be used in this more contextual and dynamic fashion; these types may be assigned to or removed from an instance over time or in a context.

For an instance to be classified with a classification, it must also have the type of what the classification <<Facet Of>>. To use the example above, a "Policeman" can't classify a Toaster since the toaster is not a person. Please see the "Role" and "Phase" discussion for more usage scenarios of <<Facet Of>>.

> **Implementation note**: most programming languages do not allow for direct representation of multiple classifications, multiple inheritance or context. A common implementation pattern is to represent classifications, roles and phases as independent objects related to the object they classify. An example of this is the iUnkown pattern in .NET.

The following stereotypes define additional classification semantics.

7.1.9.5 Roles

Roles are facet classes that are expected to be dynamic and contextual, such as teacher, victim or president. A role is defined using a class with the <<Role>> stereotype and, optionally, a <<Facet Of>> generalization. Implementation technologies should interpret roles as classifications that may be added to or removed from an instance over time and may be defined in a particular context. A role is usually required to be a role of some particular other class, for example a teacher is expected to be a role of a person (at least until a computer takes her job). The constraint of what a role must be a role of is defined using a <<Facet Of>> stereotype of a generalization. A role will also frequently have a relationship with something where the multiplicity is at least one. For example, a person is a parent if they have at least one child.

Many implementation languages don't have the capacity to represent roles, so roles are sometimes implemented is the single and unchangeable "type" of a class or DBMS table. The problem with this is that the same individual may not be connected across all their roles. Specifically representing roles allows the same individual to play multiple roles and for these roles to change – this better reflects the reality of the world and the way we think about it.

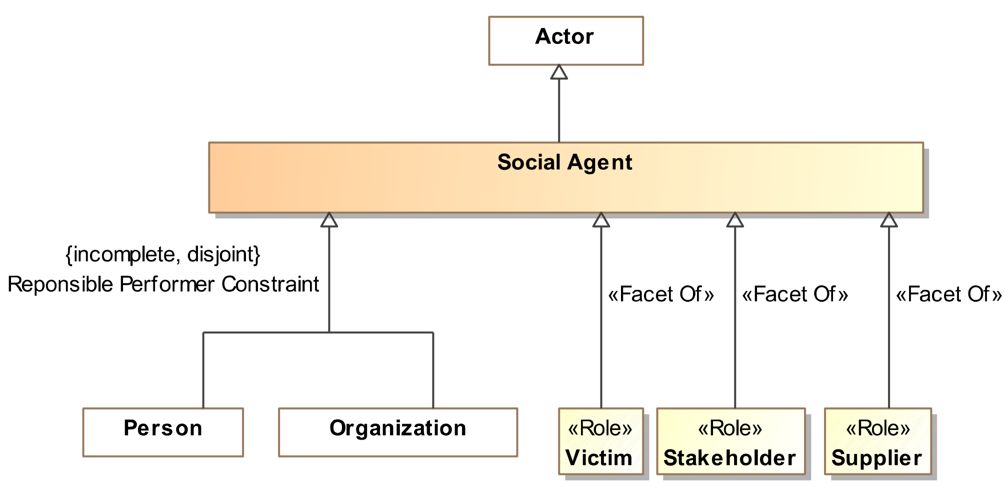


Figure 22 Role Example

The above example shows that a "social agent" can be a person or organization and that either could be classified as being able to play the Supplier, Stakeholder and/or a Victim roles.

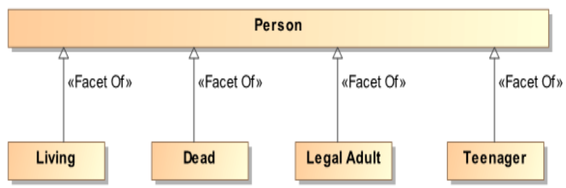
Roles help to decouple concepts in models and specifically allow an instance to "play" multiple roles at the same time or over time. Roles, when combined with quantification constraints, clearly define the semantics of roles. For example, we could say that a victim must be a victim of some incident and an owner must own something.

There are various implementation patterns for roles; SMIF does not define a specific implementation pattern, such choices will be based on the target technology and application requirements. Examples of such implementation patterns include defining a separate technology object for each instance of an entity playing a role, defining roles in separate graphs or using multiple classification.

Phases

Phases are facet classes that are expected to classify an instance over a specific span of time, such as a teenager, "legal adult" or "Paid Invoice". A teenager is a person between the ages of 13 and 19 (inclusive) – perhaps "legal adult" is of age 19 or older – we may also want to consider people living or dead, thus "alive" and "dead" would be phases of a lifeform. Phase may be considered a synonym for the "State" of something.

A phase is defined as a class with the <<Phase>> stereotype. Like roles, phases use the <<Facet Of>> stereotype of a generalization to define what a phase must be a phase of.

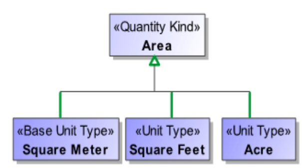


[CC20]

Figure 23 Phases of a person

Also like roles, phases help to decouple concepts in models and specifically allow an instance to "be in" multiple phases (or multiple roles) at the same time or over time. If an instance cannot be in two phases at the same time or be in a role and a phase a "disjoint with" constraint can be used to state that restriction. For example, "Dead" is disjoint with "Legal Adult" and "Living". Only a "Legal adult" can commit to a contract.

7.1.9.6 Quantity kinds and units



[CC21] Fundamental to understanding and describing something is physical and other qualities such as temperature, length and color. Many data models fail to capture units of measure explicitly which can and has[1] resulted in dramatic systems failures. A concept for somethings weight should properly be typed by a measure of weight, not an "int" or "real" – which are just ways to represent numbers without knowing what they mean. Of course there needs to be numbers, but in relation to their units.

In that there are different units that can represent the same kind of measure, such as degrees Celsius and degrees Fahrenheit can represent the same temperature – an abstraction is used above like units. The abstraction for a measurable unit is called a <<Quantity Kind>>. Examples of quantity kinds include Length, mass, temperature, frequency, etc.

As any element of measurement data must be specific to a specific unit in a specific data exchange, the <<UnitType>> stereotype is used to define a unit for a quantity kind. A <<Represents>> stereotype of generalization (Diagrammed as a green arrow) is used to say that the unit represents the quantity kind.

In the example above, the "Area" quantity kind (indicated by a black shaded class) can be represented by (the green lines) "Square Meter", "Square Feet" or an "Acre". One unit may be nominated as the "Base Unit" and will be used to express conversion factors between the units. As per SI specifications, the Square Meter is the base unit.

By convention quantity kinds are used in fully conceptual models whereas units are used in data models. The "Animal" example shows quantity kinds being used to define properties of animals.

7.1.10 Assertions about concepts

Above we defined the network of essential concepts as classes, relationships and properties. Additional assertions may be made about those concepts using both UML foundational and extended profile capabilities. The following define the kinds of assertions that can be made. Note that the term "property" applies to both simple properties and the ends of associations.

7.1.10.1 Property Ownership

The concept modeling profile of UML interprets the owner (defining class) of a property *definition* as the subject of that property (its domain) and the context in which that property must conform to certain constraints.

Constraints may be placed on a property. These constraints can include multiplicity, which includes a minimum cardinality and a maximum cardinality, a type for the property, existential quantification, and universal quantification. [CC22] When an instance is a member of a class, all of that class' constraints must be met.

Property ownership is not interpreted as "slots" in an object. Property values may or may not be independent of the instance that defined them, thus supporting an OWL/RDF, or "open world", interpretation of properties and associations.

7.1.10.2 Cardinality

Cardinality defines how many value of a property may exist for a particular subject instance. For example, how many ages can a person have? The obvious answer is that a person can have at most one age at any one point in time. Thus cardinalities represent the number of instances at any one time – regardless of how it is represented.

UML allows the cardinality of a property to be left unspecified, in which case it defaults to 1..1. The concept modeling profile interprets unspecified cardinalities as 1 (one) based on UML defaults. Note that conceptual models do define what you may or must know or what the requirements of a data model are – they define what must be true about the world as it is conceived.

7.1.11 Constraining properties and associations

A cardinality of one or more defined for a property requires that an instance of the related element must exist for an instance of the domain (owning class) of that property or association end to be valid. For example, a living person must have exactly one living brain. This is known as an *existential quantification* (∃) or qualified constraint in first order logic. Existential quantification is defined using UML cardinality and, potentially, *subsets*.

An existential quantification can be stated for a newly defined property or an existing one. For a newly defined property this is done by simply stating cardinality greater than one. For example, a phone must have at least one button with a "has buttons" association end and a cardinality of "1..*". When a new property is being defined it is given a name. If an existing property is being constrained (without a new property being defined) it subsets or redefines [CC23] the existing property and does not need a name. In the concept modeling profile of UML, any cardinality requiring one or more creates an existential quantification constraint.

A property is not limited to a minimum and a maximum cardinality (known as multiplicity) for just one type. A property can have a multiplicity for a superclass, while at the same time having a more specific multiplicity for one or more subclasses of that superclass. This type of constraint is an assertion that, among other possible values, the number of values of one of these subclasses is between some minimum and maximum cardinality.

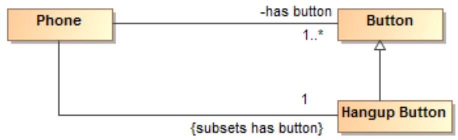


Figure 27 Phone constraint: A phone must have a hangup button

For example, we may say a phone must have one or more buttons with a "has button" property but exactly one of those buttons must be the "hang up button". We would then define an unnamed property with the type "hang up button" that subsets the "has button" property with a cardinality of 1. If we wanted the Hangup Button to also define a new property, we would give that property a name.

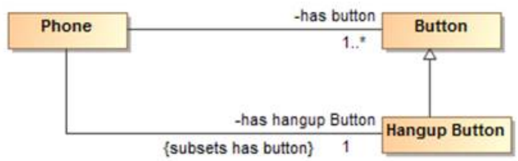


Figure 28 Hangup button with new property

In the concept modeling interpretation of UML, subsetting or redefining a property without giving the new property a different name (or leaving off the new property name altogether) creates a constraint without defining a new property.

As {subsets} or {redefines} with an omitted name is not well defined in UML, in the concept modeling profile it is used to state that a subset of values must meet the stated cardinality and type constraints of the subsetting property. It does not define an instantiable property of the domain, although it does indicate a context in which this constraint holds: the owning class and its subclasses.

The diagram below shows an existential quantification constraint on the global property "is conferred by" (from the Anything "Thing). The multiplicity is such that at least one of the instances of the property constraint must be one of the types in the union.

Note that the property adding the constraint is unnamed. This is equivalent, in this case, to naming this property the same as the property being constrained ("is conferred by" from the Anything "Thing").

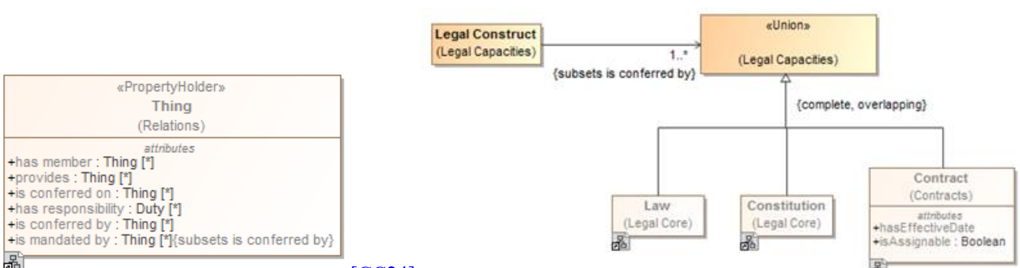


Figure 29 Constraining a global property

7.1.12 Tightening a property's type

Sometimes it is necessary, in the context of some class, to constrain *all* the values of a property to a particular type. When defining a new property, the type of that property asserts that all values of that property must be of the given type. This is known as a *universal* quantification or *for-all* constraint (∀) in first order logic. This kind of constraint is an assertion that only values of the specified type are valid, and the number of values must be between some minimum and maximum multiplicity.

Where all values of a property must be of a given type in a specialized property, UML *{redefines}* is used as part of the definition of the property. [CC25] If the redefined property is given a name, a new property with the quantification is defined. If the redefined property does not have a name, the existing property is constrained in the more specialized context (usually a subclass).

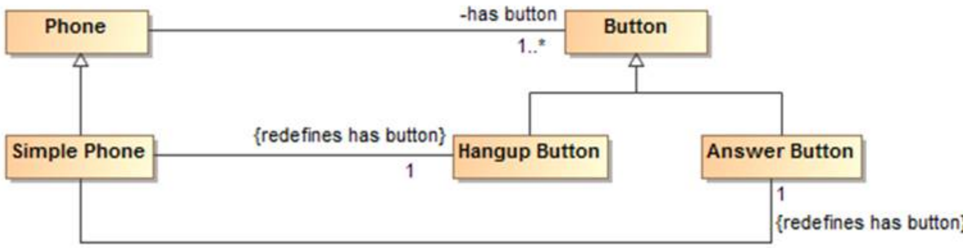


Figure 30 Example of redefines

The example above shows a "simple phone" that has exactly two buttons and they must be an answer button and a hangup button. Since redefines is used, no other buttons are allowed.

The diagram below shows the introduction of a new property "consists of", defining a universal quantification constraint on the property. The constraint states that, in the context of Soccer Team and any of its subclasses, all values of this property must be of the type "Soccer Player" and that there must be between 5 and 11 values of this property.

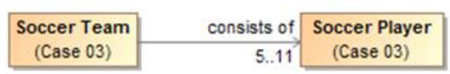
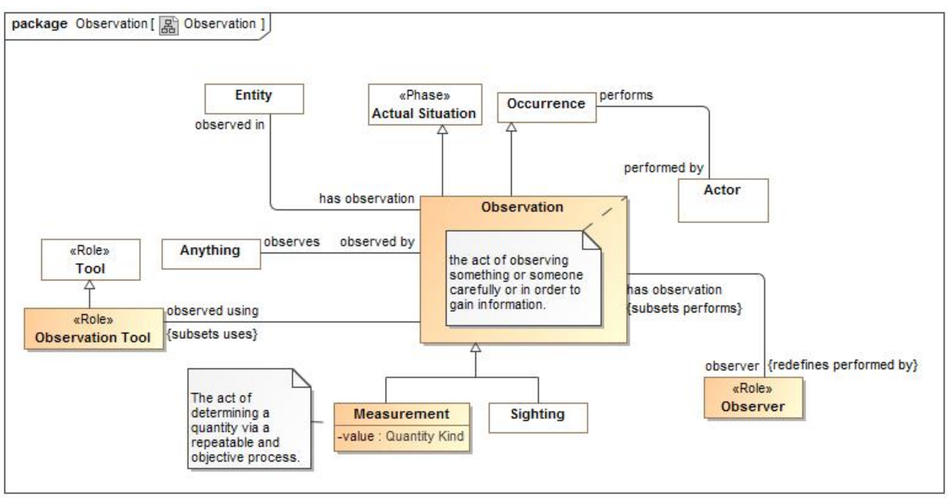


Figure 31 Example of Cardinality range

The diagram below shows a universal quantification constraint on the property "observer". Where any occurrence can be performed by any actor, an observation must be performed by an entity in the role of observer.



[CC26]

Figure 32 Observation Example

7.1.13 Inferring a type from its properties

A property's multiplicity or type is declared in the context of an owning class or a special «Anything» class. These declarations are always *necessary* conditions for an instance to be a member of the owning class[CC27] , or, in the case of «Anything», for an instance to be valid at all.

Another kind of condition is known as *necessary and sufficient*. A class with at least one necessary and sufficient condition is known as a *defined* class[CC28] , which means the differentiating characteristics of the class that make it distinguishable from its parent and sibling classes are defined. Note that using a necessary and sufficient condition on a property with a minimum cardinality of zero is not meaningful.[CC29]

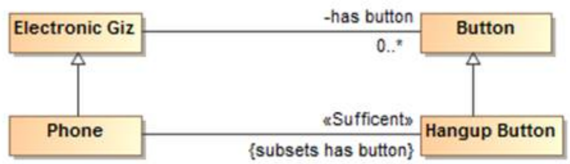


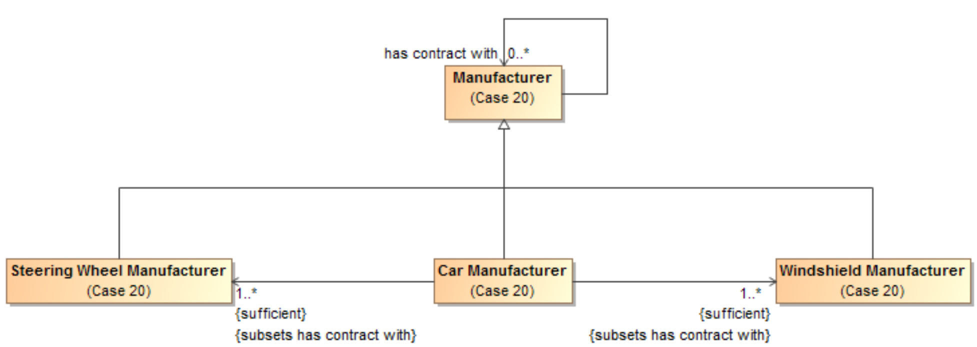
Figure 33 Phone example for sufficient

The diagram above defines a phone as *any "electronic giz" that has a hangup button*. The existence of a hangup button is sufficient to know something is a phone.[CC30]

In the concept modeling interpretation of UML, a property that has the «Sufficient» stereotype applied to it indicates that when an instance satisfies the multiplicity and type constraints for all the sufficient property's' values, not only is a *necessary* condition for being an instance of the class met, it is a *sufficient* condition.. This necessary and sufficient condition could allow an inferencing engine to classify that instance as a member of the class that owns the property. All <<sufficient>> constrains of the class and all superclasses must be met for an instance's type to be inferred.[CC31]

In the concept modeling interpretation of UML, a property that has the «Sufficient» stereotype applied to it indicates that when an instance satisfies the multiplicity and type constraints for all the sufficient property's' values, not only is a *necessary* condition for being an instance of the class met, a *sufficient* condition is also met. This necessary and sufficient condition allows an inferencing engine [CC32] to classify that instance as a member of the class with that condition. Once an instance is classified automatically, the conditions on any other properties that have the «Sufficient» stereotype, including those inherited from superclasses, merely become *necessary* conditions the instance must meet to be a *valid* member of the owning class. An instance satisfying the constraints of all the «Sufficient» properties is enough for an inferencing engine to automatically classify an instance.

The diagram below shows that when an instance with the property "has contract with" satisfies specific multiplicity ("1..*") and type constraints (of type 'Steering Wheel Manufacturer" and "Windshield Manufacturer") for the property's values, the instance meets necessary and sufficient conditions to be a member of the class "Car Manufacturer". Therefore, an inferencing engine [CC33] would classify this as an instance of the class "Car Manufacturer". As discussed above, an instance meeting all of these necessary and sufficient conditions is enough to classify the instance. The conditions on the values of these properties become necessary conditions on an instance for it to be a valid member of class "Car Manufacturer."[CC34] Also, an instance meeting all of the necessary and sufficient conditions is enough to distinguish instances of the class "Car Manufacturer' from its parent class "Manufacturer."



[CC35]

Figure 35 An example of necessary and sufficient condition

7.1.14 Property Chain

A property chain is useful for composing a property from two or more other properties that are put together in a chain[CC36] . It defines the property with reference to the other properties. The property chain allows you to navigate from a starting class (the one with the stereotype «Equivalent Property») [CC37] through a chain of properties that take a path through the same or multiple other classes.

A property chain is an ordered list of linked properties, therefore, it should have two or more "chain" tagged values[CC38] .

The following example describes a Person class that has two subclasses "Female Person" and "Male Person", and four properties "has parent", "has father", "has uncle", and "has brother". The stereotype of the property "has uncle" will be **«Equivalent Property»,** and the tagged value is **chain = has father, has brother**. (Note that the «Equivalent Property» stereotype is suppressed in this diagram, but the tagged values are not.)[CC40]

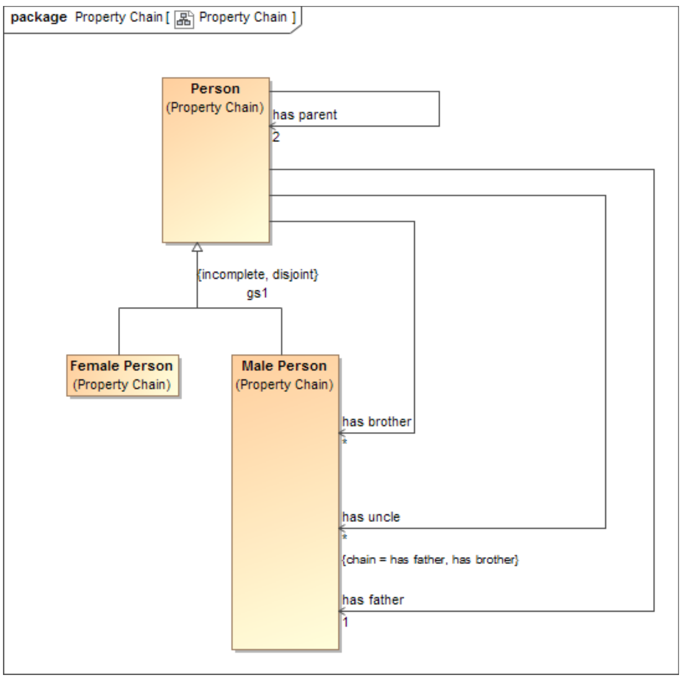


Figure 34 Property Chain Example

7.1.15 Equivalent Property

An «Equivalent Property» [CC41] allows you to represent equivalent properties [CC42] in a model. You can make a property equivalent to two or more other properties by applying the stereotype **«Equivalent Property»** to the referenced properties and the tagged value "**equivalent to**" the equivalent properties.

The following figure shows the equivalent properties in a diagram[CC44] .

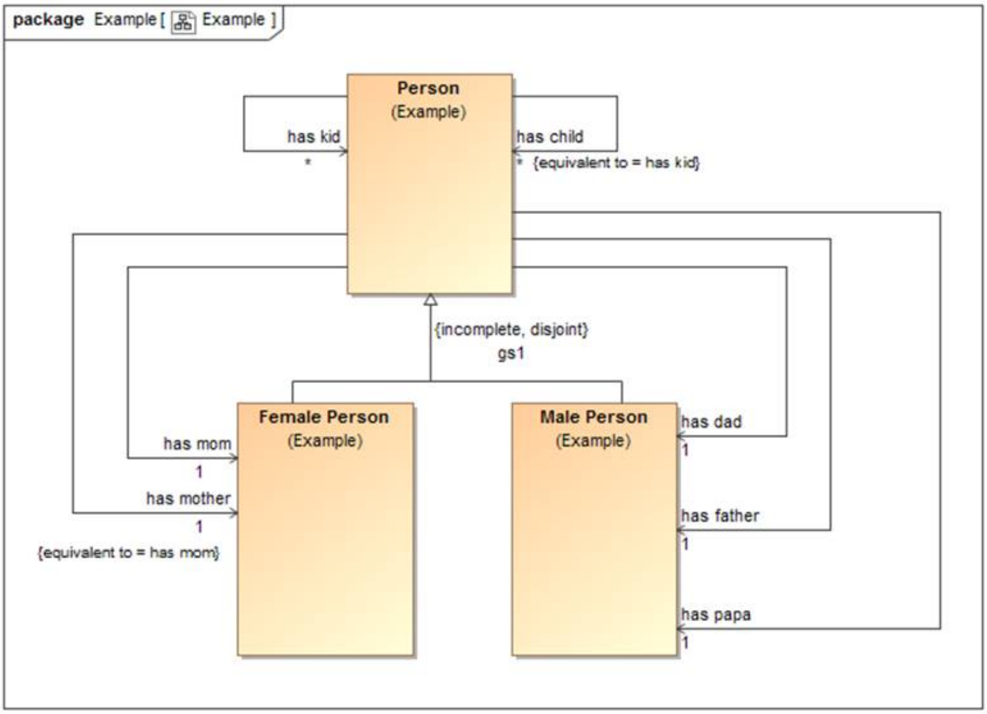


Figure 35 Equivalent properties Example

In the example, the property “has mother” is equivalent to the property “has mom”.

7.1.16 Equivalent Class

An «Equivalent Class» stereotype applied to a generalization can specify equivalence between two classes. Class equivalence expresses a generalization relationship stereotyped as **«Equivalent Class»**. Tools may draw this with a double-headed arrow.[CC45]

The following figure shows two equivalent classes in a diagram.

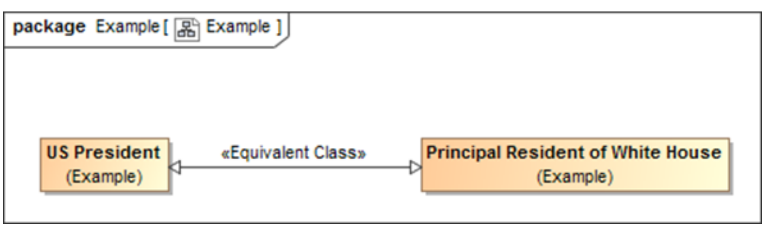


Figure 36 Two Equivalent Classes in the Concept Modeler

In the example, the equivalence class arrow defines that the two classes are semantically equivalent to each other.

7.2 SMIF Profile::SMIF Concept Modeling Profile Reference

The conceptual modeling profile defines the conceptual modeling capabilities of SMIF in UML.

7.2.1 Diagram SMIF Conceptual Modeling Profile

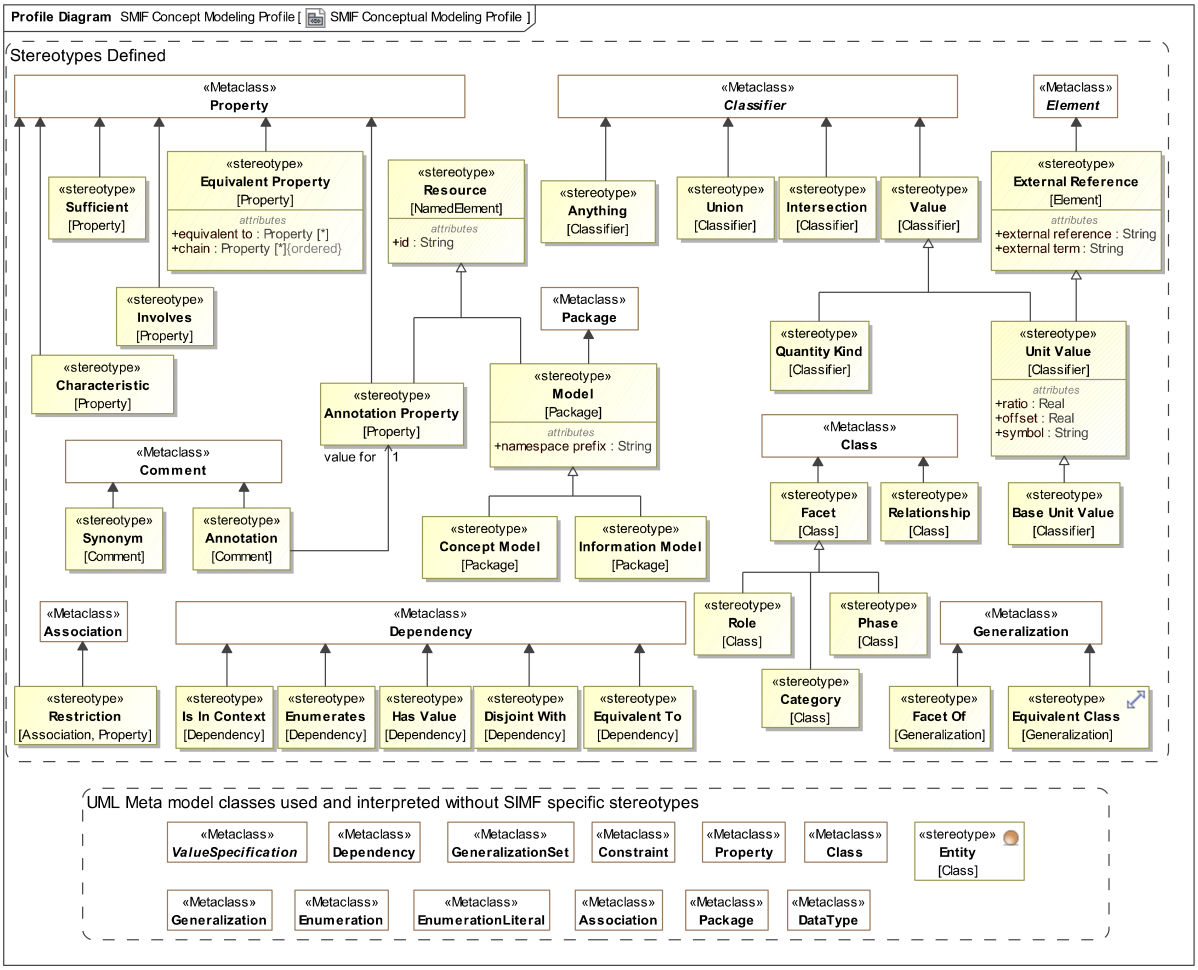


Figure 1 SMIF Conceptual Modeling Profile

7.2.2 Stereotype Annotation

An <<Annotation>> comment provides a textual "body" as a "value for" one <<Annotation Property>> describing the annotatedElement(s).

Base Classes

- Comment

Tag Definitions

○ value for : Annotation Property [1]

<value for> is the property for which the <<Annotation>> is providing a value.

7.2.3 1.2.3 Stereotype Annotation Property

An <<Annotation Property>> is a kind of <<Resource>> that asserts a property represents metadata rather than assertions about the subject domain.

Base Classes

- Property

Direct Supertypes

- Resource

7.2.4 1.2.4 Stereotype Anything

<<Anything>> is a class that represents anything and is equivalent to all other classes of anything in any other model or logic. The defined class is equivalent to SMIF:Anything, OWL:Thing and other "top level" classes.

Because of this equivalence, every class in every model virtually inherits from Anything, just as all OWL classes virtually inherit from owl:Thing.

<<Anything>> classes may be used to define "global properties".[CC46]

Base Classes

- Classifier

7.2.5 Stereotype Base Unit Value

<<Base Unit Value>> is a kind of <<Unit Value>> that marks one Unit Value of a quantity kind as the base Unit Value within a model. The base Unit Value provides the basis for conversions between units of the same quantity kind. The base unit always has a ratio of one and an offset of zero.

Base Classes

- Classifier

Direct Supertypes

- Unit Value

7.2.6 Stereotype Category

A category is a facet that is a classification or division of people, events or things regarded as having particular shared characteristics. Categorization is typically contextual, potentially transient and may or may not be formally defined.

As with all facets, categories are non-rigid. Something classified by a category must also be classified by an entity type. An entity may be classified by any number of categories and those categories may change over time.

Base Classes

- Class

Direct Supertypes

- Facet

7.2.7 Stereotype Characteristic

A kind of characteristic a type of thing may have, e.g. paint may have a color. Characteristics are the type of characteristic bindings which are "first class" elements and may participate in relationships and have other characteristics.

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

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

[UML] Property

Base Classes

- Property

7.2.8 Stereotype Concept Model

A <<Concept Model>> is a kind of <<Model>> that represents concepts in a real or possible world. Instances of elements in a concept model are "real world" things, not data about those things.

Base Classes

- Package

Direct Supertypes

- Model

7.2.9 Stereotype Disjoint With

A <<Disjoint With>> dependency is an assertion that two model elements do not and may not denote any of the same set of entities.

When applied to a classifier, every element of the classifier's extent (set of instances) is included in the set of disjoint things.

Base Classes

- Dependency

7.2.10 Stereotype Enumerates

An <<Enumerates>> dependency asserts that the supplier of the dependency is a type and the client of the dependency is a package containing a complete set of possible instance specifications. In this way, <<Enumerates>> is more general than a UML Enumeration because it can enumerate more than just UML data types.

Base Classes

- Dependency

7.2.11 Stereotype Equivalent Class

A <<Equivalent Class>> generalization is an assertion that two classes have the same extents (set of instances). Unlike ontological languages it is not assumed that the two elements are consistent, as statements from different context may or may not agree.

Base Classes

- Generalization

7.2.12 Stereotype Equivalent Property

<<Equivalent Property>> is a declaration that a property is equivalent to one or more other properties (using "equivalent to") or is equivalent to a chain of other properties (using "chain"). <<Equivalent Property>> with at least one value for the "equivalent to" property is an alternative way of expressing <<Equivalent To>>, without introducing additional lines on a diagram.

Either "equivalent to" or "chain" must have a value.

Base Classes

- Property

Tag Definitions

chain : Property [*]

An ordered list of properties forming a "property composition" expressing a traversal path that is equivalent to the stereotyped property. This is similar to a "property chain".

Due to potential "missing information" in creating a chain, a chain may or may not be able to be determined from asserting the chained property. Such a determination is defined in the mapping rules for that property in a particular context.

Note that a chain may also be defined with mapping rules.

equivalent to : Property [*]

A set of properties that the <<Equivalent Property>> is equivalent to. Note that equivalence can also be declared with a <<Equivalent To>> dependency.

1.2.12 Stereotype Equivalent To

An <<Equivalent To>> dependency is an assertion that two model elements represent the same thing or the same set of things. Unlike ontological languages it is not assumed that the two elements are consistent, as statements from different contexts may or may not agree.

Base Classes

- Dependency

7.2.13 1.2.13 Stereotype External Reference

<<External Reference>> provides traceability to the source of a "fact" in a model based on some external information resource. This references helps to facilitate provenance. Reference is a statement about the model data and has no semantic implication. Source reference may impact the trust in a statement but the evaluation of trust is outside of this specification.

External reference is combined with the owned comment(s) to create SMIF descriptions as defined in the SMIF meta model..

Base Classes

- Element

Tag Definitions

external reference : String

Specifies the location URL of the external resource. The format must comply with [RFC3987].

external term : String

The external term or location of the information in the source. The form of expression of the term or term path is dependent on the referenced technology.

Stereotype Facet Of

A <<Facet Of>> generalization is a "mix in" or "non rigid" classification of an entity beyond any fundamental (rigid) entity type.

An instance must be typed by the classifies supertype for it to also be classified as the classifies subtype. A classification may be contextual, such as within a relation, situation and/or time frame. Instances may have any number of types and facets may change over time.

<<Facet Of>> is used in defining what a <<Role>> may be a role of, and for phases, what a <<Phase>> is a phase of. For a <<Category>>, <<Facet Of>> defines the kind of thing classified.

<<Facet Of>> Implies that specialized type is a <<Facet>>

Facets may be added to or removed from an individual over time and in different context.

Base Classes

- Generalization

7.2.14 Stereotype Has Value

A <<Has Value>> dependency asserts that the client of the dependency is a type and the supplier of the dependency is an instance specification that defines acceptable values for one or more properties of that type. Each slot of the instance specification is a possible value for a corresponding property in the type.

<<Has Value>> corresponds to one or more OWL property restrictions containing a "hasValue" constraint.

Base Classes

- Dependency

7.2.15 Stereotype Information Model[CC52]

An <<Information Model>> is a kind of <<Model>> that represents

a model for some purpose, independent of technical implementation. An information model may contain logical models or data models, as well as other logical viewpoints.

Base Classes

- Package

Direct Supertypes

- Model

7.2.16 Stereotype Intersection

An <<Intersection>> is a class that has an extent (set of instances) equivalent to the intersection of the extents of all supertypes. Intersection is a stronger statement than a subtype, as a subtype may be a subset of the intersection. An instance of all the supertypes implies an instance is also an instance of the intersection type.

For intersection, The SMIF profile considers UML generalization and UML realization equivalent. This is due to ownership and legacy considerations in UML. Generalization is the preferred representation.

Note: Realizations are included to support unions across external models. UML generalization cannot be used across external models due to the ownership of generalization.

Base Classes

- Classifier

7.2.17 Stereotype Involves

<<Involves>> defines a property of a class as the "end" of a relationship - the way in which instances of a a relationship participate in (or, are involved in) instances of another type (including other relationships). Sometimes called a variable, argument or role.

In a conceptual model the terms associated with an <<Involves>> property are typically "verb phrases" defining how instances of the involved type participate in the relationship.

Base Classes

- Proeprty

7.2.18 Stereotype Is In Context

<<Is In context>> is an assertion that the client of the dependency is in the context of the supplier of the dependency. All assertions and rules defined in the supplier context apply to the client and everything in the context of the client (i.e., it is transitive). Packages, classes, situations and instances are typical contexts. Note that <<Is In Context>> is the default interpretation of a dependency, if no stereotype is specified it will be interpreted as <<Is In Context>>.

Base Classes

- Dependency

7.2.19 Stereotype Model[CC53]

<<Model>> is stereotype of package that may have an id (see <<Resource>>) and/or a namespace prefix (like the "dc" in "dc:title").

Base Classes

- Package

Tag Definitions

namespace prefix : String

A hint as to an appropriate abbreviation for a model that may be used in some technology mappings, such as XML. The prefix should be short and contain only letters and numbers and must start with a letter. e.g., "dc" in "dc:title".

Direct Supertypes

- Resource

7.2.20 Stereotype Phase

A <<Phase>> (a.k.a. "State") is a classification of an entity based on change of that entity over time. A <<Phase>> <<Facet Of>> the types that may have that phase (e.g., "Teenager").

A phase is a [DOLCE] "non rigid sortal", a type that may change over the lifetime of an entity.

Base Classes

- Class

7.2.21 Stereotype Quantity Kind

<<Quantity Kind>> is an aspect common to mutually comparable quantities represented by one or more units. Units with a common quantity kind may be algorithmically converted to any other unit of that quantity kind. e.g. temperature. [JCGM 200:2008].

Units with a common quantity kind may be algorithmically converted to any other unit of that quantity kind. e.g. temperature. SMIF takes a wider view of quantity kinds to include conversions that may be contextual and time dependent, such as currencies.

Base Classes

- Classifier

Direct Supertypes

- Value

7.2.22 Stereotype Relationship

A relationship defines a condition involving related things. A relationship may be asserted within a context as true or false within that context. Each instance of a relationship has a number of bindings to the "ends" of the relationship which do not change for the life of the relationship..

A relationship may be true or false within its context (including a time frame) but is atomic in its truth value.

Relationships may participate in (be bound to) other relationships and as such bindings involving a relationship may change over time. That is, relationships are "first class" objects.

The relationship stereotype may be used with association classes or classes. All associations are implicitly relationships. Classes stereotyped as relationships should stereotype the relationship ends as <<Involves>>.

Base Classes

- Class (including association class)

7.2.23 Stereotype Resource

A <<Resource>> is anything that can be referenced by an identifier in a model, ontology or vocabulary. The resource identifier is often an IRI.[CC54]

Base Classes

- NamedElement

Tag Definitions

id : String

A unique identifier for any resource.

When defined for a Package, id has the format defined in [RFC3987]. In this case, it is equivalent to UML:URI, and setting one will set the other.

Stereotype Restriction

A restriction is a property or association that constrains an existing property or association rather than defining a new concept in a domain.

Properties with no name or the same name as a property they subset or redefine are implicitly restrictions. Associations with a restriction as an end are implicitly restrictions.

Base Classes

- Association
- Property

Stereotype Role

A <<Role>> is a classification of an entity based on that entity's behavior, participation in a situation, or capabilities. A <<Role>> <<Facet Of>> the types that may play that role. e.g., "Teacher" <<Facet Of>> “Person”.

A role is a [DOLCE] "non rigid sortal", [CC55] a type that may change over the lifetime of an entity.

Base Classes

- Class

7.2.24 Stereotype Sufficient

Specifying <<Sufficient>> for one or more of a classes or association’s properties means that an instance having an acceptable cardinality of values for all of those properties implies that the instance is an instance of that type.[CC56]

Base Classes

- Property

7.2.25 Stereotype Synonym

<<Synonym>> defines an alternate name for the annotated elements of the comment. The alternate name is the body of the comment.

The alternate name will not be the "preferred name" of the element.

Base Classes

- Comment

7.2.26 Stereotype Union

A <<Union>> is a class that has an extent (set of instances) which is equivalent to the union of the extents of all types that specialize the Union (Subclasses). Specializing types shall include subtypes and types that realize the union.

Note: UML realizations are included to support unions across external models because UML generalization can not be used across external models due to the ownership of generalization.

[MathWorld] Given two sets A and B, the union is the set that contains elements or objects that belong to either A or to B or to both.

Base Classes

- Classifier

7.2.27 Stereotype Unit Value

A <<Unit Value>> is a <<Value>> with an <<External Reference>> that represents a type of a quantity value referencing a specific unit. A Unit Value is a required type of a property representing a quantity.

[JCGM 200:2008] A Unit is a real scalar quantity, defined and adopted by convention, with which any other quantity of the same quantity kind can be compared to express the ratio of the two quantities as a number. e.g. Degrees Centigrade, Miles.

Each Unit Value represents refinement of a quantity kind using generalization and is thus substitutable for that quantity kind. Typically, quantity kinds are used in conceptual models and Unit Values in physical or logical models.

Unit Values may only subtype quantity kinds and numbers.

Note that Unit Values are not units, but the type of quantity values expressed with reference to a common unit as defined in [JCGM 200:2008].

Each instance of a Unit Value shares a common unit (as defined by standards) with a reference defined by "external reference" and "external term".

Classifiers defined as <<Unit Value>> shall semantically subclass the SMIF model "Unit Value" class.

Base Classes

- Classifier

Tag Definitions

offset : Real

The difference between zero in the unit and zero in the base unit after the ratio is applied to the base unit as defined within the same model.

ratio : Real

The multiplier by which to multiply the unit to convert to the base unit as defined within the same model.

symbol : String

The accepted symbol for a unit. e.g. "g" for "Gram".

Direct Supertypes

- External Reference
- Value

7.2.28 Stereotype Value

A <<Value>> is a type representing an atomic unit of information without independent identity. Values include numbers, strings and enumerations. In some cases values may have internal structure. Values do not change over time.

Quantity kinds and units are also values. Values may stereotype any classifier. UML data types, including primitives and enumerations, are implicitly values.

Classifiers defined as <<Value>> shall semantically subclass the SMIF model "Value" class.

Base Classes

- Classifier

7.3 UML Profile – SMIF Patterns & Model Mapping Profile

This section is still undergoing edits.

Pattern based rules provide a general framework for stating the consistency of and between SMIF models and elements. The primary use of patterns is for mappings between data models and conceptual models using a <<Mapping Rule>> however a <<Pattern Rule>> may be used to assert consistency within a model, for example to represent generic assertions such as "all birds have feathers". Pattern based rules are declarative in nature.

Mapping rules define how a particular data model or schema <<*Represents*>> information about the concepts defined in conceptual reference models. This facilitates an "n-way" mapping of information represented using different data models. Since conceptual models are not data models they do not have any particular representation for "data instances" of that model. Instances of a conceptual model would be the real things in the real world os a possible world based on real world concepts[CC57] . The real-world concepts are the "pivot points" between the data representations. Of course implementations may automate data models that correspond closely to conceptual reference model, but that is outside of this specification.

Due to the various ways to represent information, mappings can become complex. The UML representation of mappings simplifies these mappings as much as possible. Note that details of the mapping relations are defined in the profile specification.

7.3.1 Structure of Rule Specifications

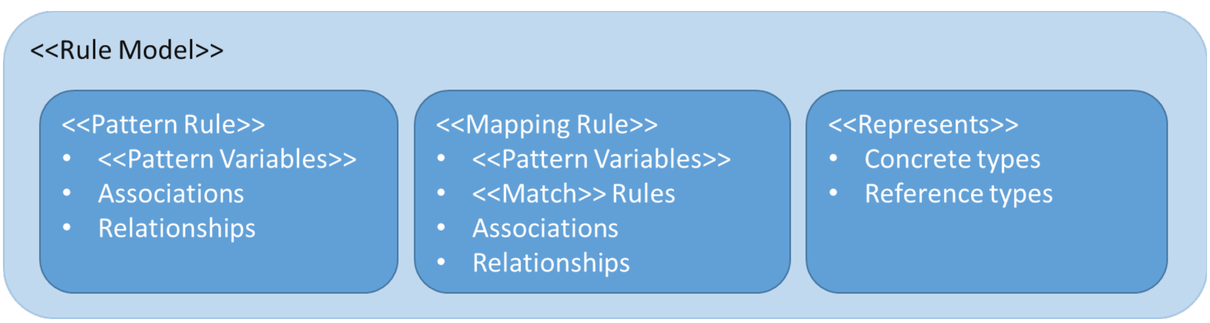


Figure 24. Structure of Rule Specifications

There is an expected structure for defining rules. This normally starts with a <<Rule Model>> package that contains pattern and mapping rules. Note that any "namespace" can contain rules, including classes. By default, rules will hold within the namespace they are defined in but another namespace may be specified by setting the <holds within> tag of a rule. Packages stereotyped as <<Rule Model>> are considered to hold universally within any model in which they are in context. Within a rule context, such as a <<Rule Model>> there may be generic rules marked as <<Pattern Rule>>, <<Represents>> rules or <<Mapping Rule>>s.

<<Pattern Rule>>s and <<Mapping Rule>>s contain <<Pattern Variable>>s that define the pattern of the rule. Pattern Variables can be UML "Parts" (which are properties), connectors and connector ends. There are various stereotypes and tags for Pattern Variables to further define their effect on the pattern. A <<Mapping Rule>> may also contain <<Match>> rules which specify how different Pattern Variables may represent the same informaiton. Mapping rules are bi-directional and may reflect changes between "either side" of the mapping.

The difference between a <<Pattern Rule>> and a <<Mapping Rule>> is that a <<Pattern Rule>> simply states something that must hold (be true) within a model. For example, that fish can swim. A <<Mapping Rule>> creates a correspondence between different representations of the same facts using <<Match>> rules.[CC58]

7.3.2 Rule Model

<<Rule model>> is a stereotype of Package to indicate that the contents should be asserted as rules.

Figure 25 Example Rule Model

The package SMIFProfileToModelMapping is a rule model and the enclosed rules will hold within any model in which it is included.

7.3.3 Representations

The foundation of mapping is the <<Represents>> dependency between classes. Represents asserts that a particular type found in a concrete (logical or physical) model represents information about a real or abstract concept in a conceptual reference model. By default <<Represents>> does not implement a mapping, it defines what elements can be mapped and thus restricts mappings. For simple "one-one" mappings there is an optional tag for <<Represents>> to <<map-all>> known instances of one Class to another.

Example

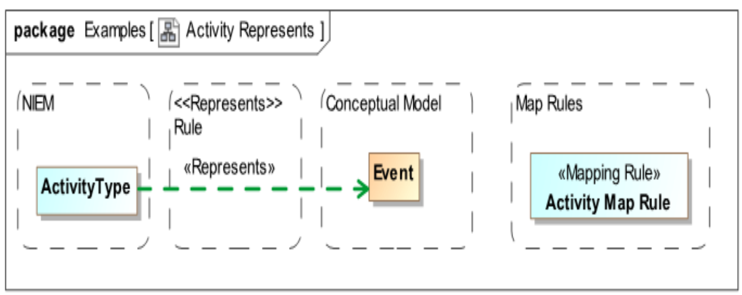


Figure 26 Activity Mapping Summary Example

The above example shows that an "ActivityType" from NIEM-Core represents an "Event" as defined in the threat/risk conceptual model. By convention we show the represents dependency as a green dashed. Representations provide the most abstract level of mapping. This diagram also shows that that there is a more detailed activity Match Rule for the same types which will map the properties and relationships between these types.

What this means is that *some* ActivityType instances represent *some* information about events in "real world" activities. Note that ActivityType may also represent other things, but that not shown in this example. Based on the SMIF mapping riles, this <<Represents>> also implies that relationships involving an event can be validly mapped to relationships involving an activity and that properties of an occurrence can validly be mapped to properties of an activity, *<<Represents>> relations provide type-safety for mappings*.

What this does not say is that ActivityType and Event are equivalent and can necessarily be mapped 1..1. How they are mapped is detailed in mapping rules. However, if the <map-all> tag of <<Represents>> is set true then ActivityType and Occurrence will be asserted as being mapped 1..1, bidirectionally (mapping of types and properties is considered independent, each property must also be mapped). Note that <map-all> implies nothing about the properties and relationships, only the mapped types (each type, property and relationship is an independent concept that is mapped independently).

Figure 27 {map all} Example

In the figure above, all UML classes stereotyped as <<Role>> will be mapped to the Role class in the SMIF model.

7.3.4 Mapping Rules

The detail of mappings happens in classifiers stereotyped as <<Mapping Rule>>s. Mapping Rules define patterns of concrete types and patterns of reference concepts that have corresponding <<Match>> rules. The <<Match>> correspondence rules do the real work, mapping element by element.

Mapping rules are, externally, not that interesting. They are just a classes or components stereotyped as <<Mapping Rule>>. However, note that Mapping Rules may specialize other rules – in which case they include the more general rule but may restrict the <<Match>> variables.

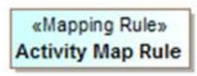


Figure 28 Representation Rule External Example

The above defines a mapping rule for activities that is an assertion that the enclosed pattern must hold and provides a context (in this case the enclosing package) where the Match Rules are asserted. If we look inside the Activity Match Rule we see the structure.

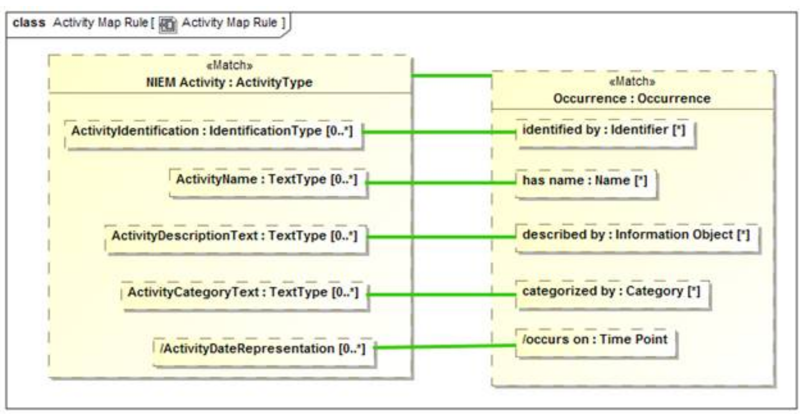


Figure 29 Representation Rule Internal Structure

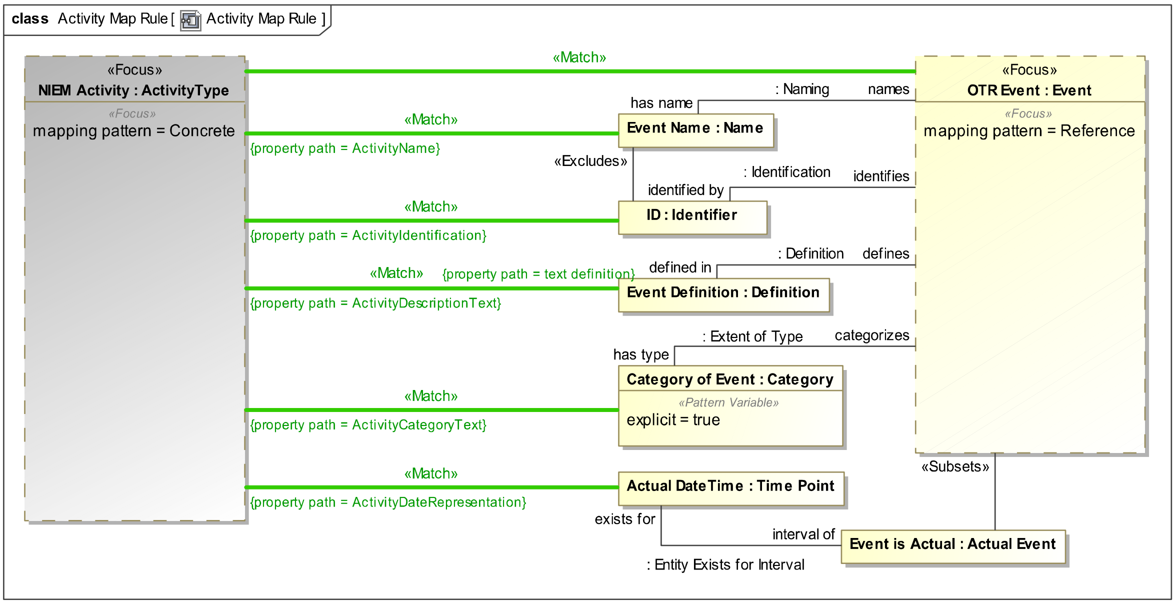


Figure 7.1: Representation Rule Internal Structure

The above example is the internal “structure” of the Activity Match Rule. In this case the mapping is very 1..1 and simple. Inside of the rule we see “parts” that represent “ActivityType” named “NIEM Activity” and “Event” named “OTR Event”. The green line between them is a “Match” rule, represented as a UML connector stereotyped as <<Match>>. This states that in this pattern NIEM Activities and TR Events map 1..1. We could also have put filter constraints on that mapping, but in this case did not.

We also see “<<Focus>> on “OTR Event” and “NIEM Activity”. Focus defines the “starting points” for the for each side of the mapping pattern where one focus is the “Concrete” information model and one is the “Reference” conceptual model. A SMIF conformant mapping engine will find all instances of “Event” (in any mapped data format) and map those to NIEM Activity. It will also find all NIEM Activities and map them to Events. All other parts of this mapping become relative to the <<Focus>> elements.

Related to both NIEM Activity and OTR Event we see other variables. The black lines show how instances of each variable will follow associations and relationships to populate other variables. Each black line is an instance of a SMIF relationship and as such also a pattern variable. Relationships between variables can be followed in the same way that relationships between actual entities could be followed.

The green lines create mapping <<Match>> assertions between those variables *within the context of this rule*. The <<Match>> assertions may connect directly to a variable or to a property within that variable, denoted as a “property path”. (Note that in UML it is not possible to show the connection directly to the contained variable – this limitation has been addressed in SysML)

Thus within this rule NIEM “ActivityName: maps to “has name” based on the <<Match>> between “property path=ActivityName” and its connection to “Event Name”. Event name has a “Naming” relationship with “OTR Event”.

The important point to remember is that mapping any fact requires that the types are compatible. That type compatibility is defined by <<Represents>> rules between the types. The requirement for type matching may be overridden by setting the <coerce> tag of the <<Match>> rule, but in most cases type safety of <<Match>> rules is desirable.

To allow for type compatibility, a <<Match>> correspondence is conditional, the types of the mapped elements must either match or have a <<Represents>> rule that allows them to be mapped. If, for example, an event had an identifier that was an image and NIEM did not allow for image identifiers, that “fact” would not be mapped. Of course, other rules could be constructed to allow for some mapping convention in this case, but SMIF will not force a <<Match>> where the types are not compatible (there is a way to override this with “coerce”, which will be explained below).

Mapping for primitive data types, such as strings and numbers, may be provided by the mapping engine implementation based on each mapped technology. This allows, for example, an identifier that is represented as an integer to be mapped to a string.

In that there may be multiple <<Match>> rules between the same thing, one can be marked as the <<Default>>. A default rule will be applied only if no other rules have fired. In the example above the "ActivityIdentification"/"ID" <<Match>> is a default. This is because a name is also a kind of identifier so ID will not include names.

7.3.5 <<Select>> Variables

The foundation of SMIF rules is patterns. When a rule is asserted a SMIF implementation attempts to "select" the pattern based on existing information and then "assert" that the pattern is "true". The <<Focus> and <<Select>> elements are those that must *pre-exist* for the pattern to even be considered. Relating this to a SQL Query, the <<Focus>> would be similar to the "FROM" clause and the <<Select>> elements would be in the "Where" clause.

If there is more than one related <<Select>> element, they must all be "true" for the pattern to hold (be asserted). If there are any constraints for the <<Focus>> elements they must also hold. Constraints include condition expressions, the type(s) of the Pattern Variables and multiplicities.

Once a pattern is selected, all properties, relationships and subsets from the <<Focus>> and <<Select>> elements are "filled in" from existing information by following the associations, relationships and properties defined in the pattern.

What happens if, as these other elements are being filled in, some other constraint is violated? This depends on the kind of rule. For a general pattern rule the constraint will be asserted – made to be true by attempting to create (assert) each required element. In the case of a mapping rule the rule is in an error state, the behavior of an implementation in response to an error state is implementation specific.

In a mapping rule, after the <<Focus>> and <<Select>> elements have been validated and any relationships followed, the <<Match>> rules for the pattern are applied, asserting the information in the corresponding concrete or reference model.

Note that for a mapping rule there will be two "sides" that are matched – the "Reference" side and the "Concrete" side. Each "side" is considered a separately selected ssub-patter. Sides are determined by the "mapping pattern" property of the focus variables.

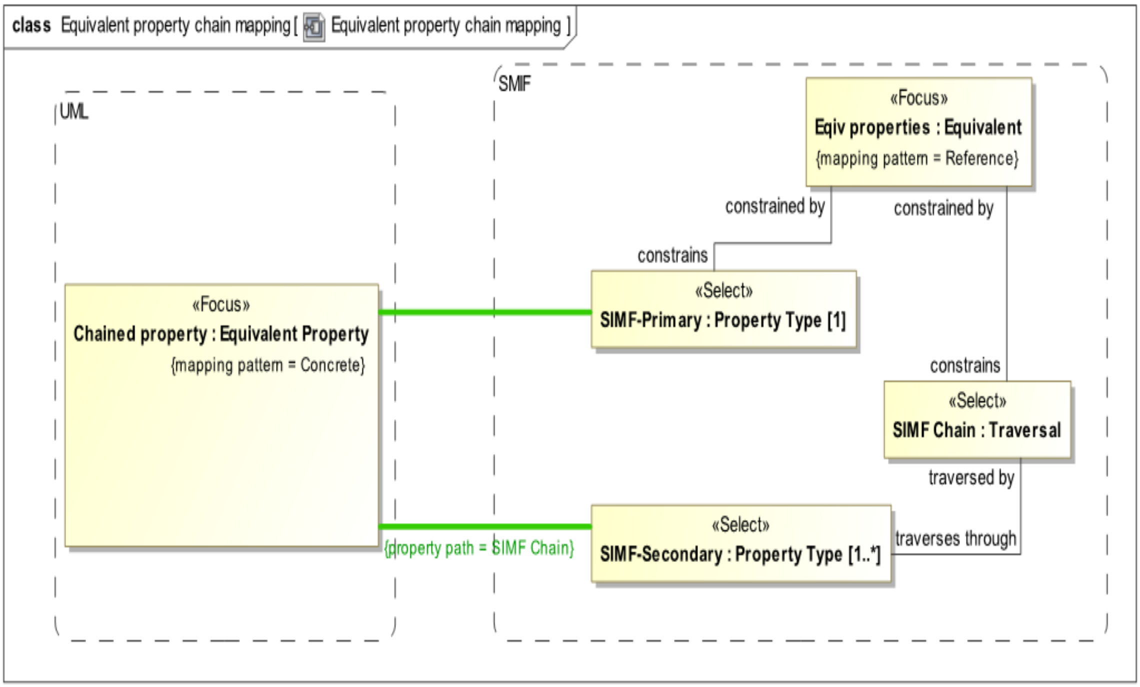


Figure 7.2: <<Select> Example

The example above shows vary specific match pattern on the “Reference” SMIF model side, there must be a pattern of an “Equivalent” constraint that constrains exactly one “Property” and also constraints exactly one “Traversal”. These patterns are very specific because there are very specific ways to represent general concepts (like equivalence) in the UMLprofile.

Once a pattern on one side is matched, the UML side is “asserted”, creating the required elements. In the opposite direction, only an Equivalent Property with a “chain” will be asserted on SMIF.

7.3.6 Multiplicity constraints in patterns

Pattern variables may be bound to zero, one or more elements, each one describes a set. It is sometimes necessary to constraint pattern variables to be bound to a specific number of values. This may occur either in matching the pattern or as the result of following various paths (associations, relationships and characteristics). The same multiplicity constraint that is used to constrain other properties, such as on the ends of relationships, may be used to constraint pattern variables. Multiplicity constraints may also be used on the “ends” of connectors between pattern properties, to constraint the number of relationships (actual ground facts) that must exist between the pattern properties.

Setting the multiplicity constraint of a pattern property constrains it to have the specified set of values. If a <<Select>> is constrained, the pattern must match the constraint. If not a match, the pattern multiplicity will be satisfied by the SMIF implementation creating the required elements. If, for any reason, this or other constraints cannot be satisfied the pattern is a violation The method for handling constraint violations is not specified.

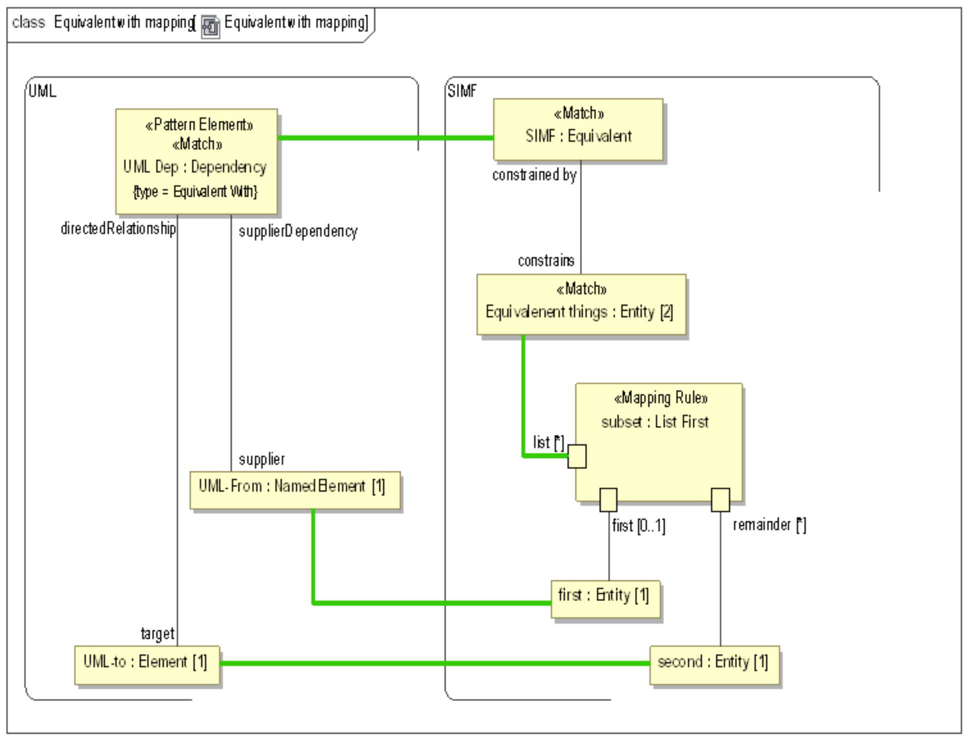


Figure 37. Example of setting multiplicity constraints on <<Match>>

In figure 218 the above example a SMIF Equivalent has exactly 2 <constrains> entities. This is the condition for matching the pattern. This matching pattern then maps the Equivalent constraint to a UML dependency stereotyped as <<Equivalent with>>.

Once these base patterns are mapped the two mapped entities will be mapped to the <supplier> and <target> of the UML dependency via a “List First” rule. The List First rule separates a list into its first and remaining elements.

7.3.7 Subsets of Pattern Variables

Conceptual models use sub classing, multiple inheritance, roles and phases to more accurately and intuitively represent the domain of interest. Many data technologies do not support these concepts and even if they did, would probably structure implementation classes differently. In other cases, there may be restrictions on the “extant” of what maps to what that require calculations or other constraints. To provide for these cases we use <<Subsets>> in mapping patterns. A subset defines another part (property) that holds a subset of the instances of the superset part, based on the type, relationship values and other constraints of the subset part.

To understand this feature we will first look at models for “Entity” and “Actor” in NIEM and the threat conceptual model, respectively.

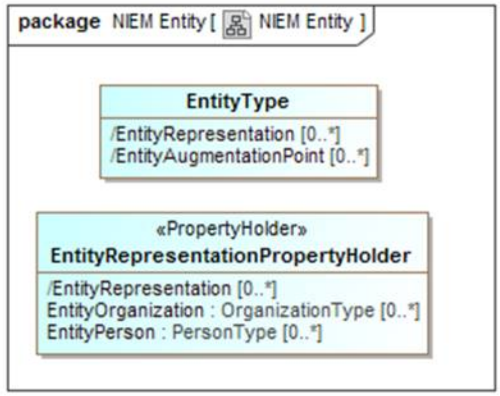


Figure 33 NIEM Entity Example

In NIEM, an “EntityType” has a “substitution group” property with properties that can be “EntityOrganization” or “EntityPerson” to allow the entity to represent one or the other. The general rules for mapping NIEM state that substitution groups are considered subtypes of the primary type.

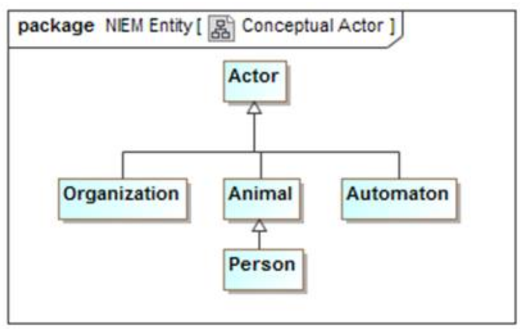


Figure 34 Conceptual Actor Example

In the Threat conceptual model “Actor” is a Supertype of Organization and, indirectly person. It is also a Supertype of “Automaton”. An Automaton can’t be an actor in NIEM so it will not be mapped (However we could define a NIEM extension to allow this).

We want to map actors to NIEM entities, but see that they are very different “shapes”.

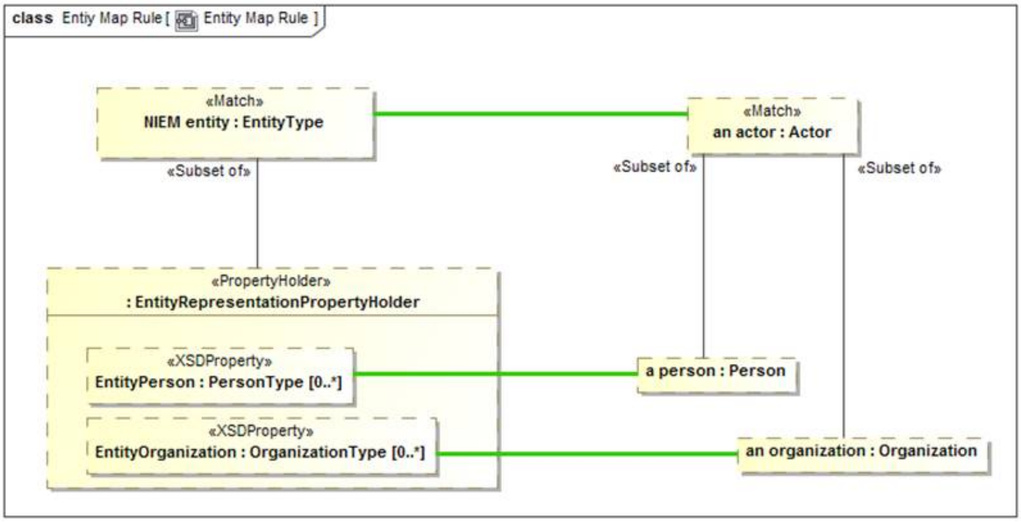


Figure 35 Subset part example

In the above example we see the actor - EntityType mapping. Notice “a person” of type “Person”. “a person” is defined to be a <<Subset of>> actor – that is every actor that is of type “Person” will populate the “a person” part. If an actor is not a Person, “a person” will be null. “a person” is then mapped to “EntityPerson”, a property of “Entity” by way of the substitution group (sorry that this gets into some NIEM substitution group details, but you probably get the basic idea).

Likewise, “an organization” will map to EntityOrganization iff “an actor” is an Organization. Note that if “an actor” is neither of these, it will not map to any NIEM property.

Note also that there could be other constraints on the subset parts, such as required relations or constraint expressions.

7.3.8 <<Pattern Variable>> computations and constraints

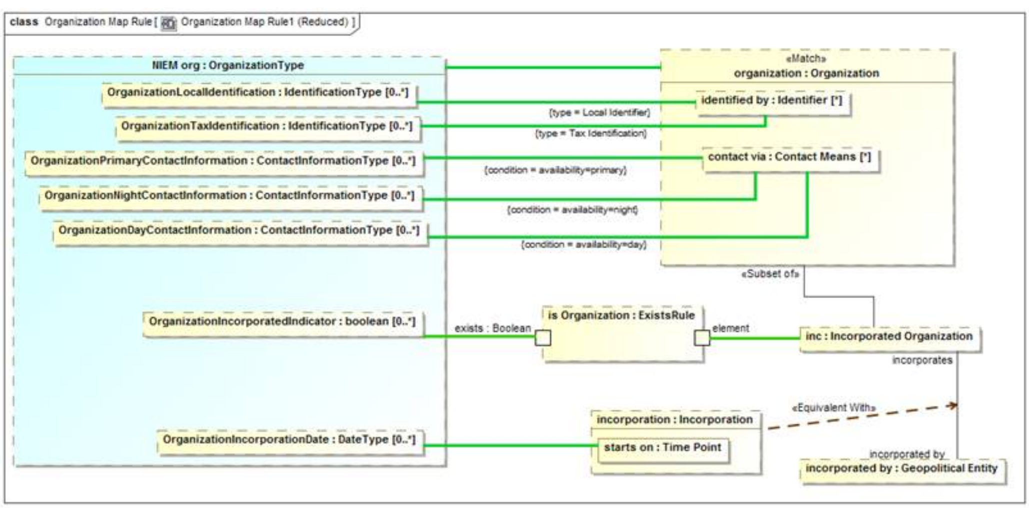


Figure 36 Map constraints example

To continue the tour of the primary mapping capabilities we will look at a subset of the “Organization” mapping.

Note the “type=” on two maps to “identified by”. In the conceptual model there are subtypes of identifiers. In NIEM there are special properties for some of these identifiers. The “type=” constraint on a map says that the map will be constrained to the type (on the specified end) of the actual instance matched the specified type. So

“OrganizationLocalIdentification” will only map to “identified by” if the type if the identifier includes “Local Identifier”. Likewise, “OrganizationTaxIdentification” will only map to “identified by” if the type includes “Tax Identifier” (remembering that a SMIF concept instance can have multiple types). Likewise, the reverse is true; those properties will “assert” the type of the identifiers they reference.

On the maps to “contact via” we see “condition=”. Condition is a tag of <<Pattern Variable>> that references a UML expression. The conditions referenced are properties of the association between an organization and “Contact Means”. The maps will be constrained to the “availability” property is set as indicated. Likewise, if an organization is being created, that property will be set by the same condition.

Note that “inc” is a subset of an organization only if it plays the role of an “Incorporated Organization”. In NIEM there is a Boolean set if the organization is incorporated. The “ExistsRule” is a computation rule (that is its implementation is outside the specification). But in this case ExistsRule’s behavior is defined – the exists Boolean will be true when the mapped “element” has some value. This results in the NIEM “OrgainizationIncorporatedIndicator” corresponding to the organization being incorporated.

If the organization is incorporated it will have an incorporation relationship to its incorporating body (incorporated by). That incorporation relationship will contain its date of incorporation, which is mapped to the NIEM property. In UML association classes have to be put into a structure like this in two pieces, the “line” and the “box”. Since both the line and the box represent the same “fact”, they are asserted to be equivalent – this is only required when association class properties need to be accessed and is required because UML has no way to show connectors as association classes.

The end result is that the more “flat” representation of an Organization in NIEM is mapped to the concept model.

7.3.9 <<Pattern Variable>> explicit

Most elements are mapped regardless of their source – explicitly asserted in a model or derived based on rules. There are times where only explicitly asserted elements need be mapped. In this case the element is marked with the <explicit> tag as TRUE.

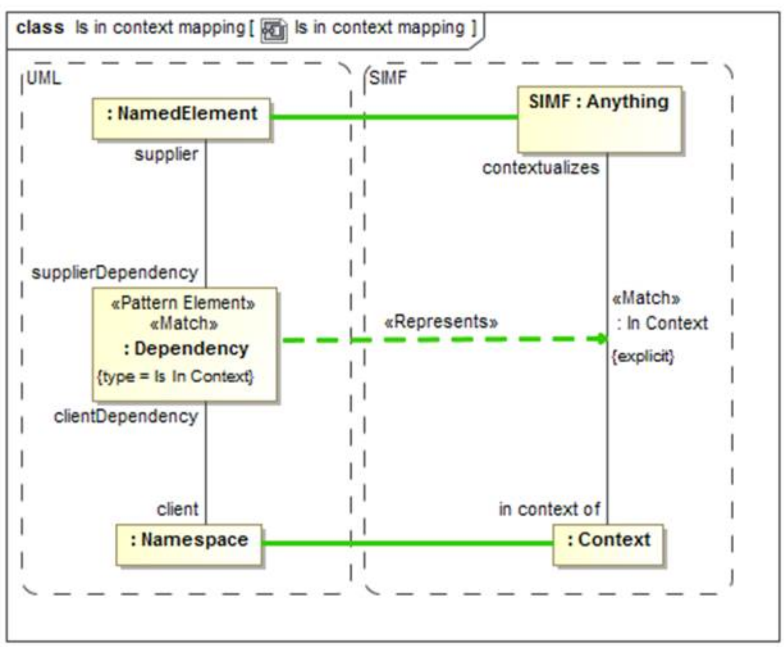


Figure 41 Example of "explicit" Pattern Variables

The above example shows that the “in context of” relationship in SMIF should only be mapped to UML if it is explicitly asserted.

7.3.10 Pattern Precedence

It is possible for more than one pattern to match for the same set of values. The general rule is that all patterns that match will execute. Where this may produce redundant elements a pattern may either subtype or subsume another. Where a pattern subtypes another and the more specific pattern matches, the more specific pattern will include the rules of the more general pattern.

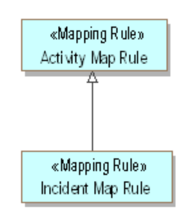


Figure 46. Example of Pattern Generalization

An incident is a kind of activity. The incident rules subtypes and subsumes that activity map. An activity that is an incident will use the incident Match Rules as well as the sub-rules defined within activity.

Where a pattern uses a <<Subsumes>> dependency, if the <supplier> pattern matches it will prevent the <target> pattern from executing for the same set of values.

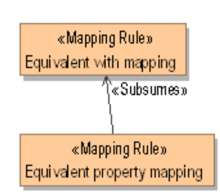


Figure 47. Example of Rule Subsumption

Using "Equivalent With" is more general but >Equivalent Property" more compact. If equivalence can be expressed with "Equivalent Property" it subsumes "Equivalent With".

7.3.11 Generic Rules

Most of our examples have used mapping rules. Rules are also generic patterns that can be asserted to hold within some context. Generic rules generally use quantifiers rather than <<Match>> but can be stated either way. A quantifier defines a pattern property that contains a set of instances defined by the property type. The quantifier specifies how many instances will be in the set from none to all.

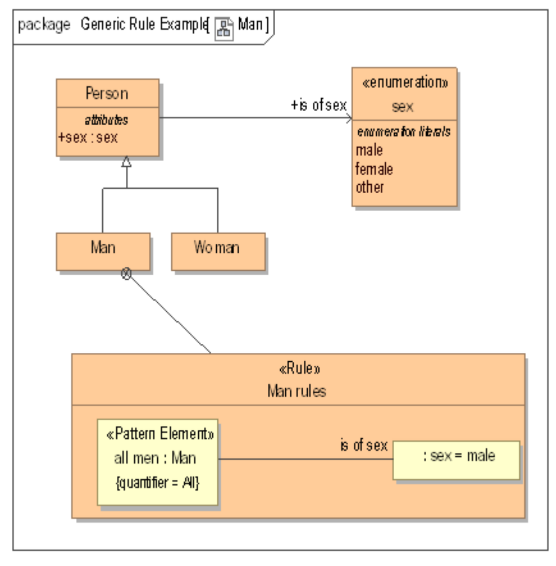


Figure 48. Generic Rule Example

Figure 47 provides an example of a generic <<Pattern Rule>>. The rule states that as part of the definition of the class Man, the "Man rule" applies which says that all men <is of sex> male. The Pattern Variable "all men" has "quantifier = All" which is really what makes it represent all men, not the name. "all men" then has a relationship to a constant "sex = male" (the default value of a property is considered its value). The result is the "assertion" that all men will have the same sex.

Note that more than one property may be quantified, for example we could say "All men like at least one supermodel" by quantifying "a supermodel" with "quantifier = There exists" and creating a connector "likes" between them. Options for quantifiers are: None, There Exists, Exactly One, Some, Most, All. Note that for an interpretation in first order logic, There Exists, Some and Most are the same, even if they may have an intuitive distinction. In other logics concepts like "Most" may offer a default.

7.3.12 Facades and Representation Computations

In some cases, it is desirable to have mapping rules as "reusable pieces" that can provide a "Face" to a model that fits better for one or more mapping rules. There is also the case where these rules fall outside of the expressive power of mapping rules and are best done in calculations (program code or fUML models).

Facades provide for making a new "face" of either a conceptual model or data model element. A Façade is a class with additional properties and/or relations that can be derived from the element it represents. Either mapping rules or computations are then used to "populate" the façade or map the façade back to what it represents. The façade implementation keeps the façade properties consistent as any connector implies change in a property value.

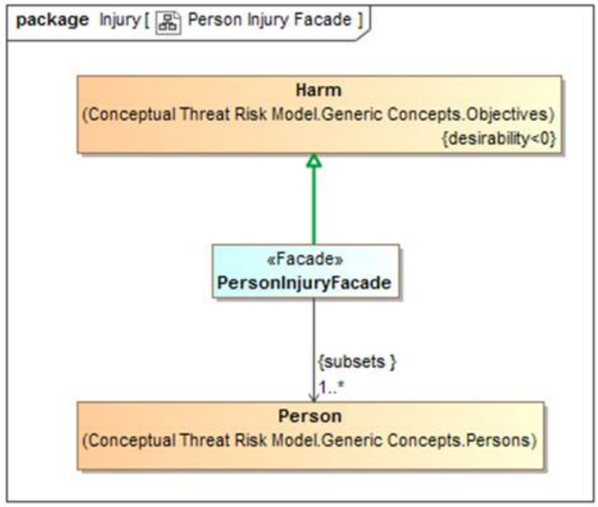
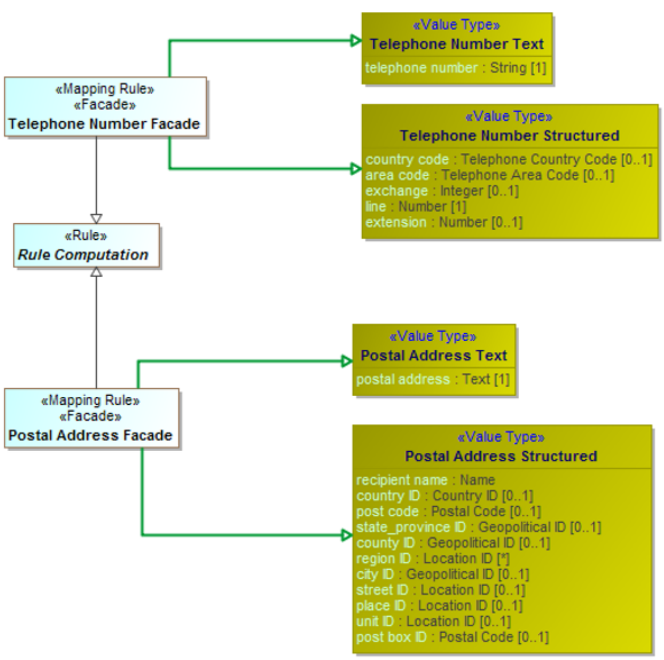


Figure 42 Facade Example

The “PersonalInjuryFacade” above represents the concept of “Harm” but only where the harm impacts a Person. In NIEM, injury is only considered relative to a person – so this façade provides such a “View” of the conceptual model, harm restricted to personal injury. In this case no additional representation rule is required, but such a façade could also define new properties or associations that would be populated in the same way as a data model.

Note that in this case the <<Represents>> relation is applied to a generalization to assert that “PersonInjuryFacade” includes all of the features of “Harm” and is also a representation of it.



Facades can also use “Computations” or Representation Rules to define their properties.

Figure 43 Computation Facade Examples

In the above example both a telephone number façade and address façade are "computed" based on combining both a structured and unstructured representation of telephone numbers and addresses. The specific computation is external to the specification and defined by implementations. These implementations could be implemented in any language, including "ALF", the executable language of UML.

The mapping engine is responsible for implementation of computation behavior and should update a computed Façade whenever any of its elements changes (some implementations may group such changes in a transaction).

In summary, facades and computations provide for reusability and extensibility of mappings.

7.4 SMIF Profile::SMIF Patterns Profile Reference

The SMIF rules profile defines the way to model rules and mapping within and between data sources via a conceptual model.

7.4.1 Diagram SMIF Patterns Profile

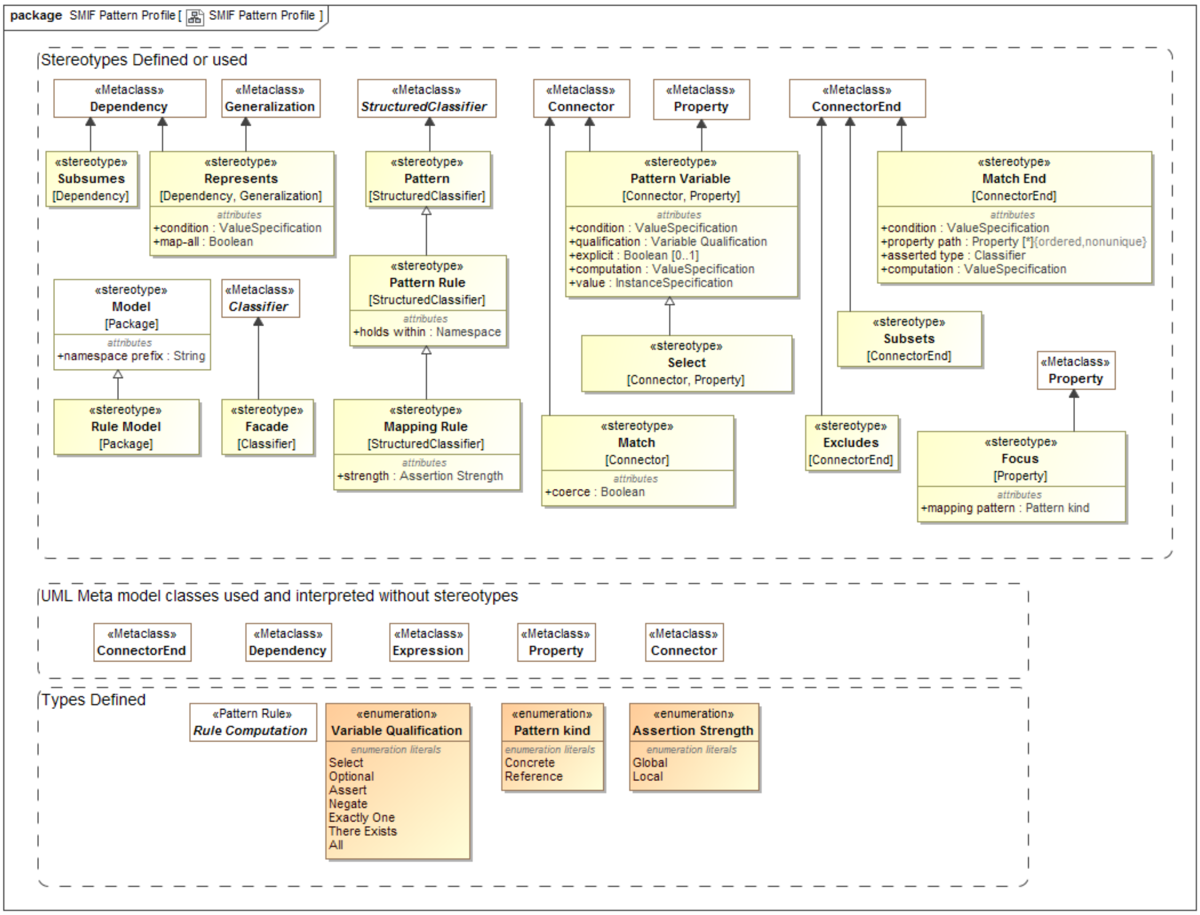


Figure 1 SMIF Pattern Profile

Computation computes a value for the mapping end based on the expression applied to the mapped property or relationship.

Where computation is used inverse mapping is not specified - any inverse mapping is implementation specific.

7.4.2 Stereotype Excludes

In a pattern or mapping rule, <Excludes> defines a pattern variable that represents a set of elements to be subtracted from the set of elements on the opposite side of the exclude connector (note that this is set subtraction, not numeric subtraction). Where more than one variable is is exclulded, the union of those variables will be used as the basis for the substraction.

<<Excludes>> stereotypes the end of a connector that is to be excluded from the opposite end.

Base Classes

- Connector

7.4.3 Stereotype Facade

<<Facade>> defines a classifier as being a view of (facade of) one or more other classifiers. Facades usually define additional properties that match some external view of a conceptual model element.

A facade will represent the classifier for which it is a facade. A Facade will use one of two methods to relate the facade properties to the conceptual Model:

* <<Pattern Rule>> using the facade.

* Applying the <<computation>> stereotype and Subclassing "Representation Computation"

Base Classes

- Classifier

7.4.4 Stereotype Match

<<Match>> defines an equality rule between two properties in a <<Mapping Rule>> - they must represent the same information, perhaps using different representations.

<<Match>> may be used between sub-patterns, as is common for a <<Mapping Rule>> or within one model to equate different representations for the same thing (e.g., property paths).

Base Classes

- Connector

Tag Definitions

coerce : Boolean

> Where <coerce> has a value of TRUE a Match Rule will be evaluated even if the <from> is not type compatible with the <to> type.
>
> Where <coerce> is FALSE or unstated a Match Rule will be evaluated only if the <from> is type compatible with the <to> type.
>
> Type compatible shall be defined as one of: Being the same type, <from> being a subtype of <to> (as defined by a type generalization rule), <from> being a representation of <to> (as defined by a representation rule).
>
> Representation rules applied to a supertype apply to a subtype.

7.4.5 Stereotype Mapping Rule

<<Mapping Rule>> defines a pattern structure described by a structured classifier that shows how both "sides" of a representation (conceptual and reference) are related. Each "side" must match, including any traversals through structures defined with properties and connectors. Such traversals are links which may also have conditions on the ends to more precisely define the pattern.

The pattern is described using structured classifier properties and connectors.

The mapping engine ensures that the patterns match, bidirectionally.

Base Classes

- StructuredClassifier

Direct Supertypes

- Pattern Rule

7.4.6 Stereotype Select

<<Select>> specifies an element in a structure that must match a data source for the pattern to qualify to hold. <<Select>> is similar to an SQL where clause.

<<Select> is a shortcut for <<Pattern Variable>> qualification=Select

Base Classes

- Connector
- Property

Direct Supertypes

- Pattern Variable

7.4.7 Stereotype Pattern Variable

<<Pattern Variable>> further defines a connector or property within a pattern as a pattern variable.

Note that the UML default value may be used to set the initial value of a Pattern Variable for primitive values.

Base Classes

- Connector
- Property

Tag Definitions

computation : ValueSpecification

<computation> computes a value for the Pattern Variable based on the expression.

Where computation is used inverse mapping is not specified - any inverse mapping is implementation specific.

condition : ValueSpecification

<condition> states a condition that must be true within the scope of the Pattern. This can be used for pattern matching, setting values or restriction of paths.

explicit : Boolean [0..1]

If <explicit> is true, the Pattern Variable must be explicitly asserted as the indicted type, not derived or inferred from a supertype or super property.

qualification : Variable Qualification

<quantification> defines the behavior of an element with respect to a pattern - how it impacts the selection, evaluation or assertion of the pattern. Specifics are defined in each variable qualification enumeration literal.

value : InstanceSpecification

Instance specification of a fixed value for a pattern variable.

7.4.8 Enumeration Variable Qualification

Variable qualification values define the behavior of an element with respect to a pattern - how it impacts the selection, evaluation or assertion of the pattern.

7.4.8.1 Literals

Select

Select is used in query and mapping patterns, all elements of the classified type that match the pattern are selected as instances of the pattern.
Select may be considered a qualified "All". Select does not assert the existence of something, it determines the existence of a pattern match such that other assertions may be made.
Where a pattern is asserted, "Select" variables shall be asserted.
Relationships between properties with <quantifier>=Select must hold between the selected properties for the pattern to be asserted.

Optional

Optional is used in query and mapping patterns, the property shall be populated as a consequence of the pattern matching.
Where a pattern is asserted, "Optional" variables shall not be asserted.
Optional is the default if no qualification is stated.

Default

The element will be asserted only if no other values are asserted within the pattern or as pre-existing assertions.

Assert

The property does not impact the selection of the pattern, it is an asserted consequence of the pattern.

Negate

The property does not impact the selection of the pattern, it is negated consequence of the pattern - it may not exist.

Exactly One

The existential quantifier limited to exactly one of a potentially larger set of the properties type.

There Exists

The existential quantifier - at least one of the properties type.

All

The universal quantifier - the quantified property is a stand-in for all elements of the existent of the quantified type

7.4.9 Stereotype Represents

<<Represents>> is an assertion that the source *concrete* type or feature provides a more concrete way to represent the target reference type. Represents may be used within conceptual models or from a physical model to a conceptual model.

- A representation that is a dependency or realization makes no assumption that the types are substitutable.
- A representation that is a generalization is substitutable for what it represents.

Base Classes

- Dependency
- Generalization

Tag Definitions

condition : ValueSpecification

<condition> is an expression that must be true for the source to represent the target.

map-all : Boolean

<map-all> implies a direct mapping between instances of the types in both directions.

<map all> is equivalent to a mapping with a rule mapping properties of each type but is lower precedence than other mappings - if types have a more specific map it will apply first.

7.4.10 Stereotype Pattern Rule

<<Pattern Rule>> defines a pattern that must hold true for the context of the rule.

A pattern rule is a pattern structure described by a structured classifier that shows how elements are related. Each mapped element must match, including any traversals through structures defined with properties and connectors. Such traversals are links which may also have filters to more precisely define the pattern. An implementation of SMIF may ensures that the patterns are asserted for the scope the rule holds for.

Base Classes

- StructuredClassifier

Direct Supertypes

- Pattern

Tag Definitions

holds within : Namespace

<holds within> is the context in which a rule is asserted (required to be true). Anything contextualized by the context is subject to the rule.

If not stated the rule is asserted by its owning scope.

7.4.11 Stereotype Rule Model

A <<Rule Model>> defines a package as containing rule specifications and asserts those rules to be true.

Base Classes

- Package

Direct Supertypes

- Model

7.4.12 Stereotype Subsets

In a pattern or mapping rule, <Subsets> defines a pattern variable that represents a subset of another variable. The subset may be constrained by a more specific type, expressions, values or required cardinalities. Where more than one variable is is subset, the union of those variables will be used as the basis for the subset.

Subset stereotypes the end of a connector that is the superset.

Base Classes

- ConnectorEnd

7.4.13 Stereotype Subsumes

<<Subsumes>> is a dependency between rules. When a rule subsumes another the subsumed rule will not apply (fire). if the <subsumed by> rules applies (fires).

Where rules are also patterns, a rule may specialize another which will subsume the specialized rule as well as include the generalized rule parts as parts of the specialized rule.

Base Classes

- Dependency

7.5 SMIF Profile::SMIF Computation Rules

Computation rules define mappings that are implemented via external methods. As such the implementation is defined by implementations, not the specification.

7.5.1 Diagram SMIF Computation Rules

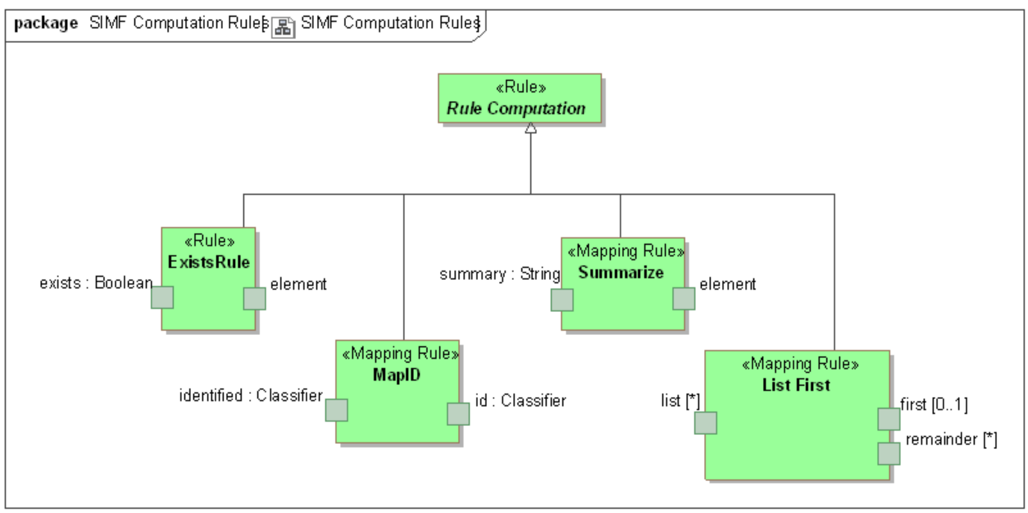


Figure 1 SMIF Computation Rules

7.5.2 Class ExistsRule

<<Exists Rule> is a rule to map the existence of an <element> to a boolean.

<exists> is true iff <element> is not null.

Direct Supertypes

- Rule Computation

Attributes

element

exists : Boolean

7.5.3 Class List First

The <List First> rules will take the <list> property and place the first element into<first>. If <list> is empty, <first> will be empty.

If there are more <list> elements than 1, all remaining elements are placed as a set in <remainder>.

If <list> is an un-ordered set the order will be indeterminate but repeatable.

<<List First>> is bidirectional and will compute <list> by appending <first> and <remainder>.

Note that this will act like a LISP CDR/CAR pair

Direct Supertypes

- Rule Computation

Attributes

first [0..1]

list [*]

remainder [*]

7.5.4 Class MapID

<<MapID>> is a rule where the source is an ID and the target is a class, maps an instance of the ID to an instance of the class.

Direct Supertypes

- Rule Computation

Attributes

id : Classifier

identified : Classifier

7.5.5 Class Rule Computation

<<Rule Computation>> is an abstract supertype for a facade that includes external implementation. The implementation is outside of this specification.

7.5.6 Class Summarize

<<Summarize>> is a rule that produces a natural language description of an element. Summarize may not be bi-directional and is expected to have information loss.

<summary> is a summary of <element>.

Content of summary is implementation specific.

Direct Supertypes

- Rule Computation

Attributes

element

summary : String

7.6 Profile mapping to SMIF Model (Normative)

The following diagrams summarize the mapping which is further defined in the UML mapping model.

Note – this section is out of date and due to be revised.

7.6.1 SMIFProfileToModelMapping::High level representation

The following diagrams show the <<Represents>> rules defined between the profile and the SMIF model.

7.6.1.1 Diagram: Anything

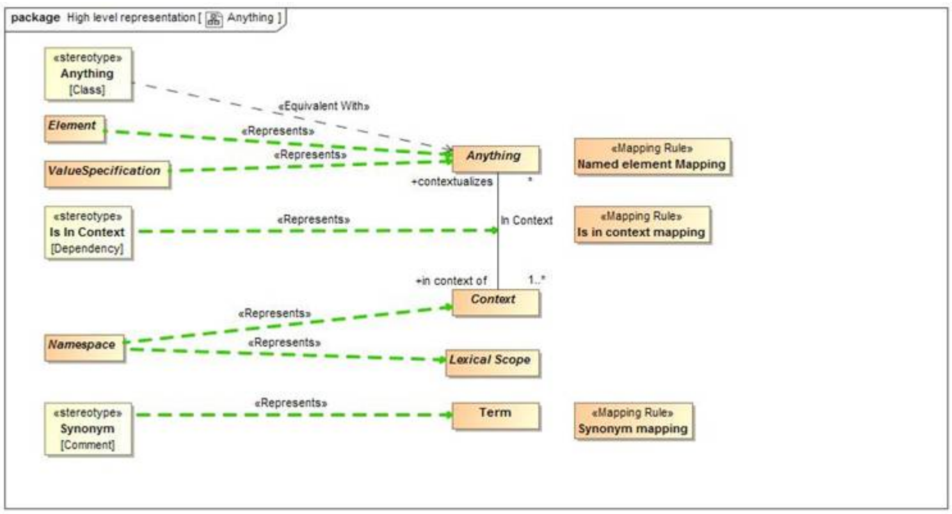


Figure 1. Anything

7.6.1.2 Diagram: Classes

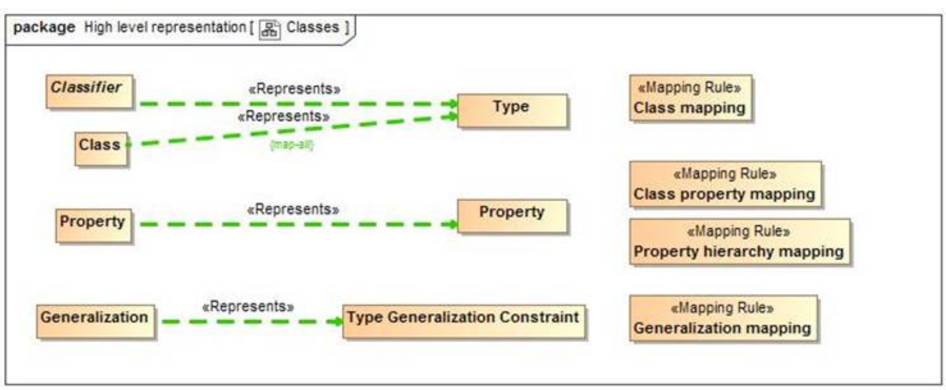


Figure 2. Classes

7.6.1.3 Diagram: Lexical Structure

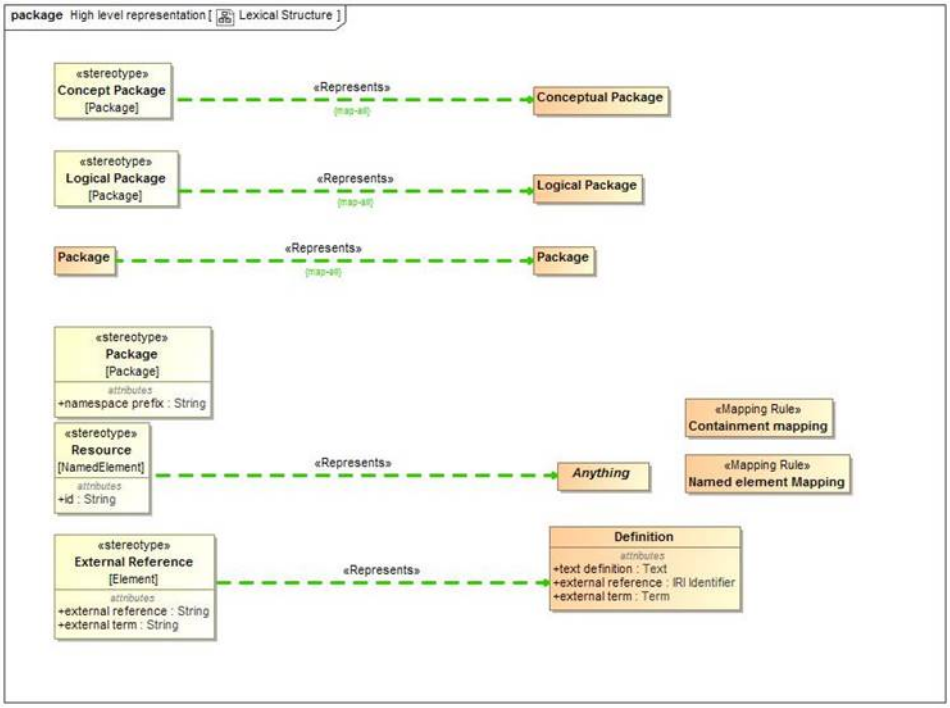


Figure 3. Lexical Structure

7.6.1.4 Diagram: Patterns

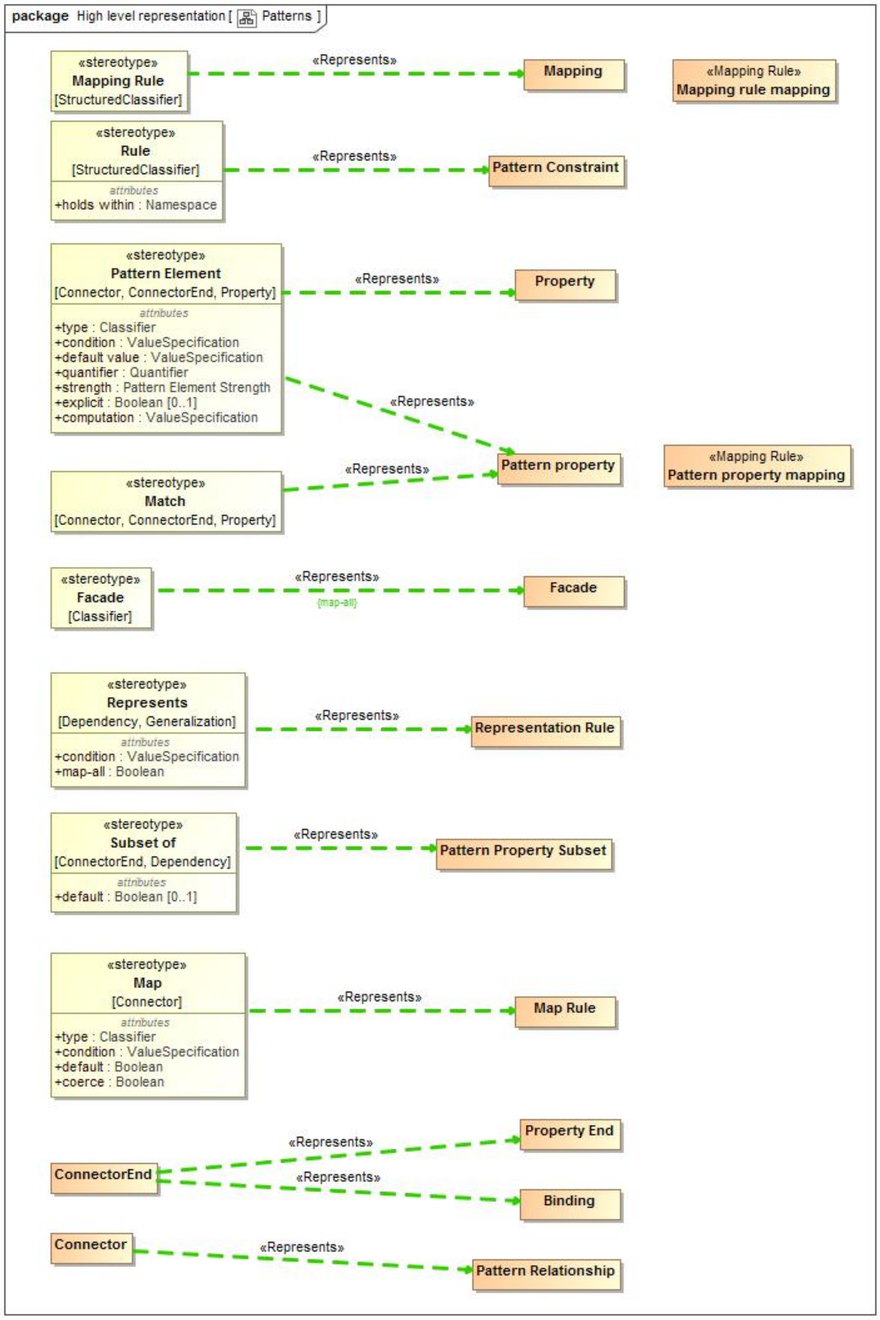


Figure 4. Patterns

7.6.1.5 Diagram: Relationships

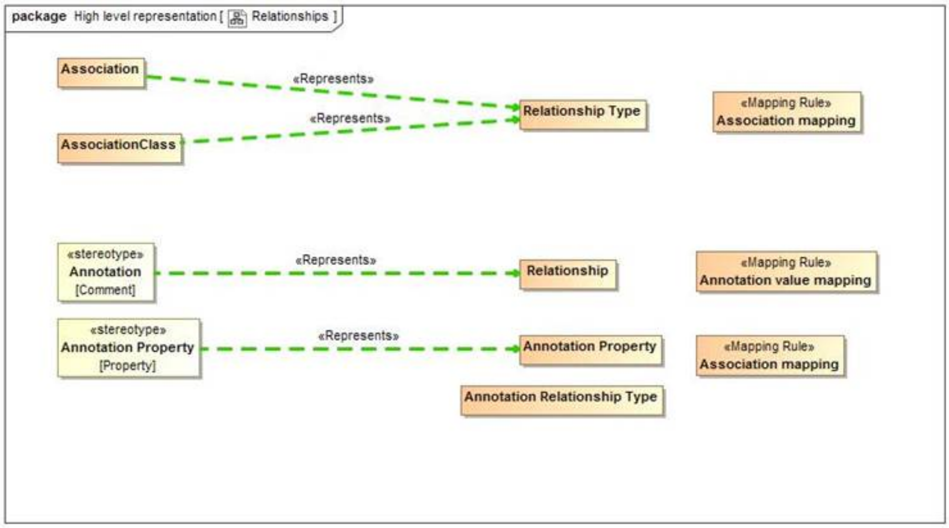


Figure 5. Relationships

7.6.1.6 Diagram: Rules

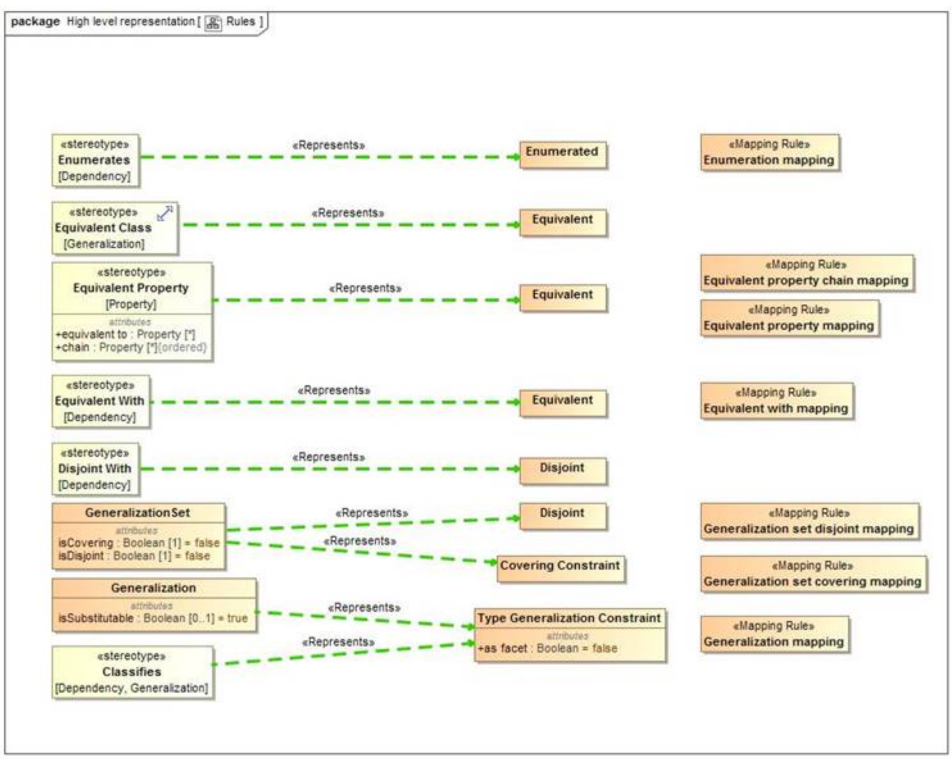


Figure 6. Rules

7.6.1.7 Diagram: Types

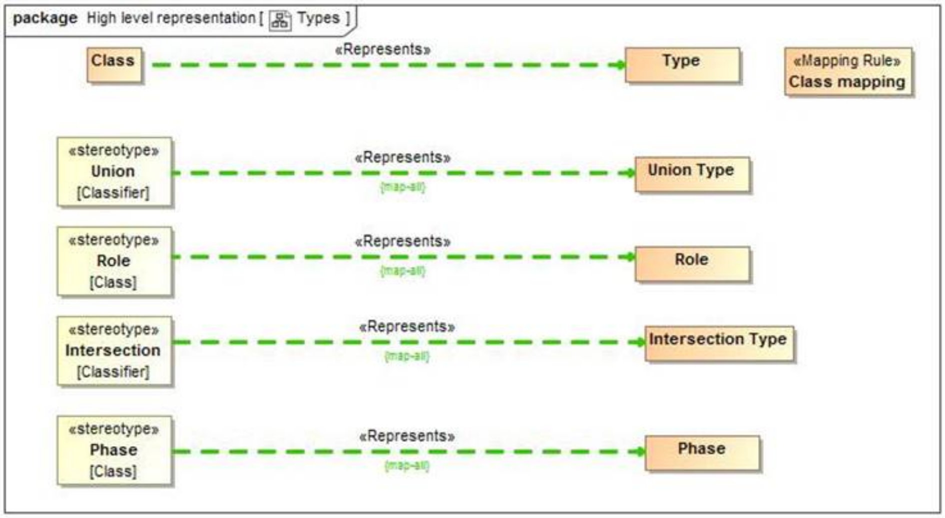


Figure 7. Types

7.6.1.8 Diagram: Values

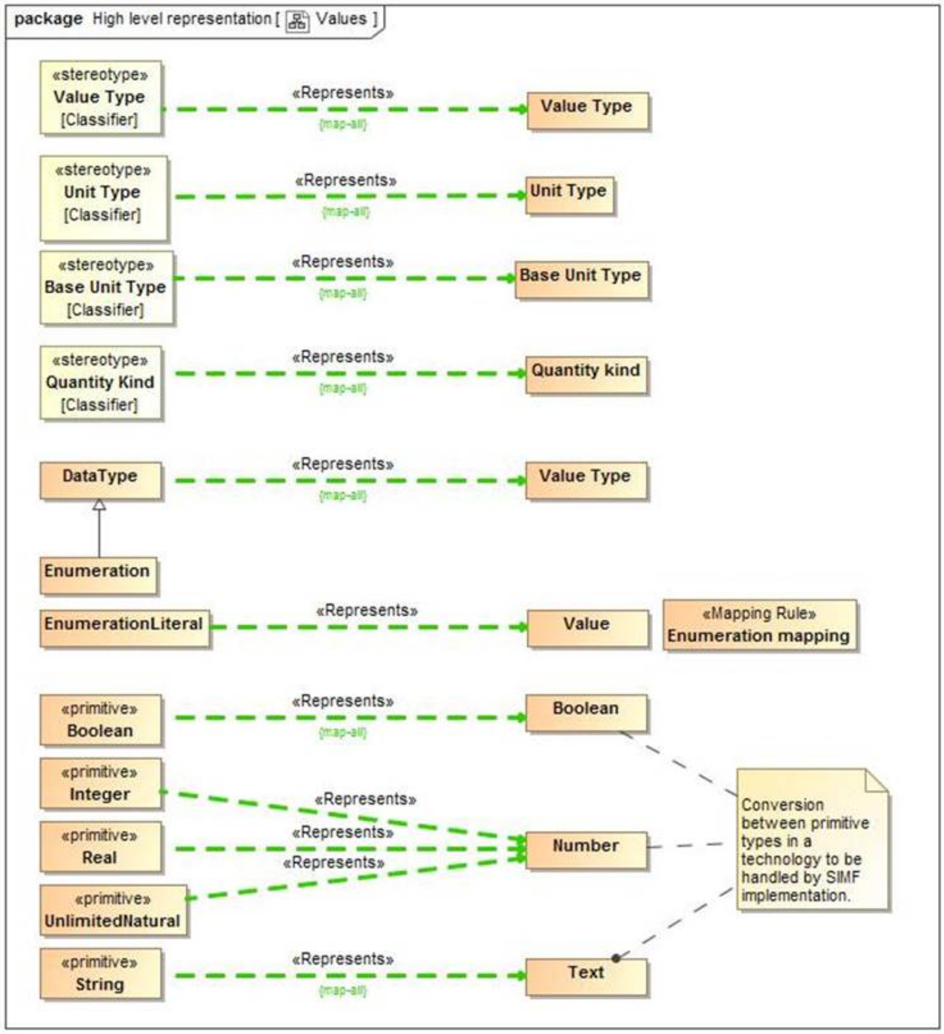


Figure 8. Values

7.6.2 SMIFProfileToModelMapping::Mapping rules

The following are the mapping rules that hold for mapping the SMIF profile to the SMIF model.

7.6.3 Class Annotation value mapping

Annotation value mapping defines a direct correspondence between a UML <<Annotation>> stereotype of a comment and a SMIF Relationship typed by a Annotation Relationship Type. It then creates the properties for the Annotation Relationship Type and binds the subject of the annotation to the annotated element and the annotation property to the string "body" value of the annotation.

Note that an annotation defines a property instance, it does not define a new property.

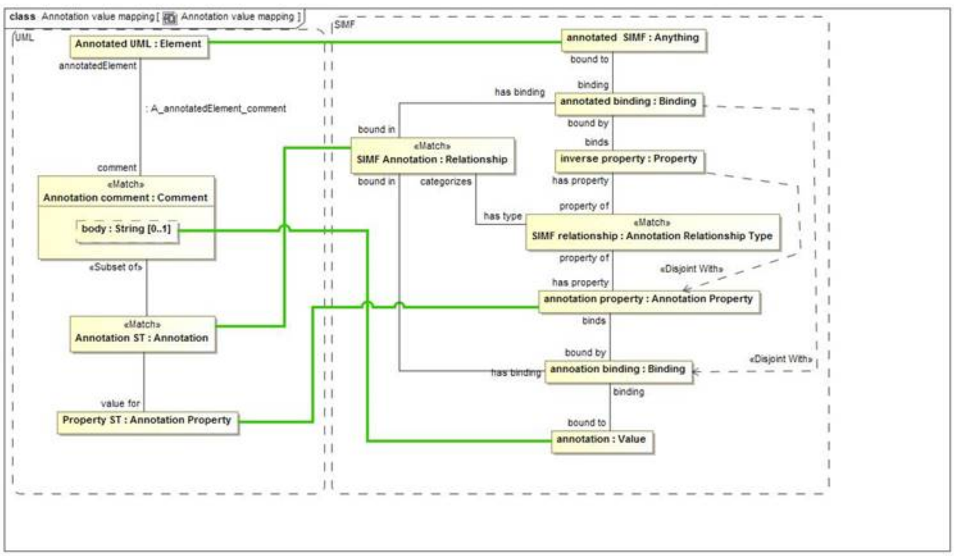


Figure 1. Annotation value mapping

package SMIFProfileToModelMapping::Mapping rules

7.6.4 Class Association mapping

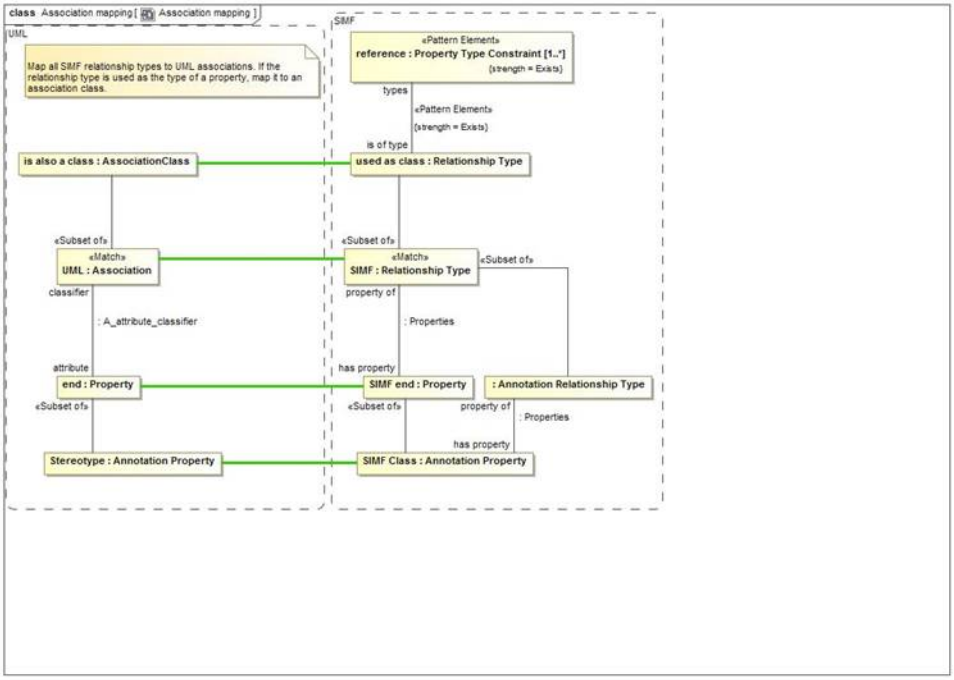


Figure 2. Association mapping

The Association mapping draws a direct correspondence between a UML association and a SMIF Relationship Type. It maps each property of the association to a SMIF property.

SMIF does not distinguish between associations and association classes (all associations are essentially classes). As a convention, a UML association class will be created only when the association is used as an type of some other property.

For annotations, UML <<Annotation Property>> corresponds with a SMIF Annotation Property as a property of a Annotation Relationship Type.

package SMIFProfileToModelMapping::Mapping rules

7.6.5 Class Class mapping

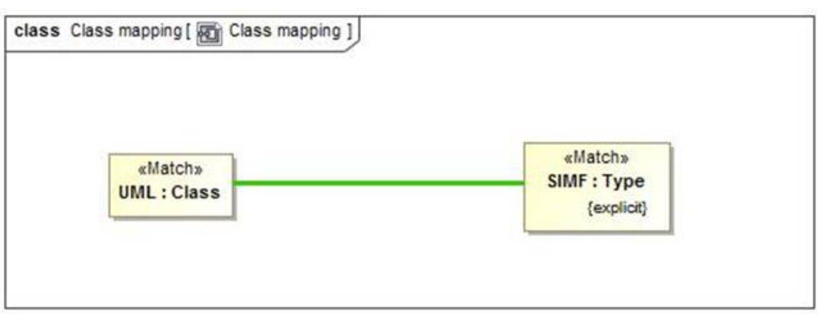


Figure 3. Class mapping

UML Classes correspond directly to explicitly asserted SMIF types.

package SMIFProfileToModelMapping::Mapping rules

7.6.6 Class Class property mapping

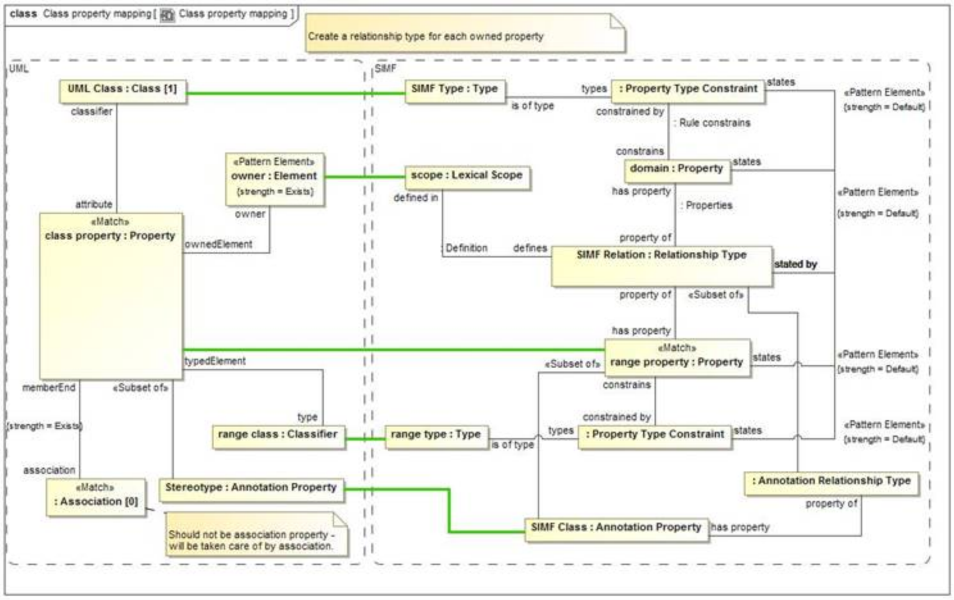


Figure 4. Class property mapping

Properties of UML classes correspond with a pattern involving a relationship. SMIF conceptual models to not define properties directly on types but each is an independent relationship type.

Each UML Property corresponds with a SMIF property with a type that matches the UML property type. The SMIF property becomes a property of a Relationship Type with a "domain" property corresponding to the UML class owning the property.

UML properties marked as an <Annotation Property>> Correspond to a SMIF Annotation Property and further classify the relationship as a Annotation Relationship Type.

package SMIFProfileToModelMapping::Mapping rules

7.6.7 Class Containment mapping

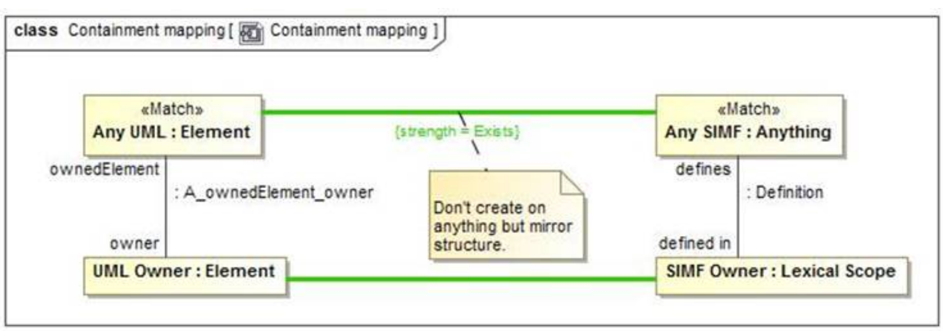


Figure 5. Containment mapping

Containment mapping forces the UML ownership structure and the SMIF lexical structure to match.

package SMIFProfileToModelMapping::Mapping rules

7.6.8 Class Enumeration mapping

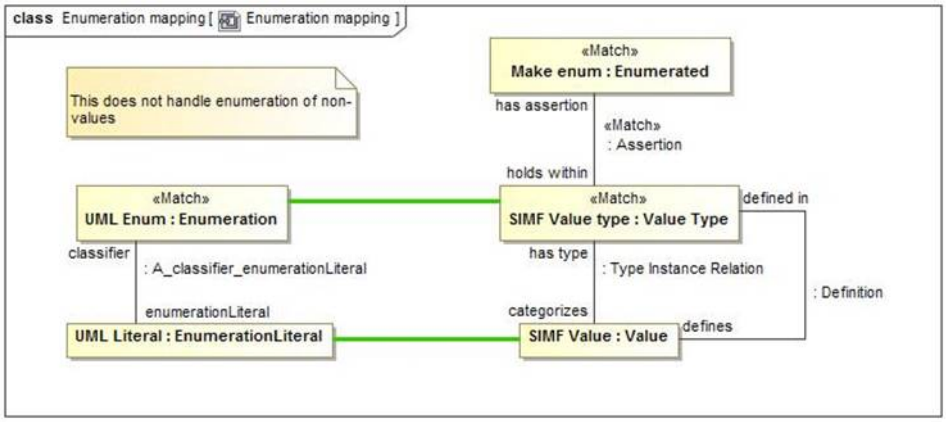


Figure 6. Enumeration mapping

Enumeration mapping draws a correspondence between UML Enumerations and a SMIF Value with an "Enumerated" constraint.

While SMIF may enumerate non-Values UML does not support this semantic.

package SMIFProfileToModelMapping::Mapping rules

7.6.9 Class Equivalent property chain mapping

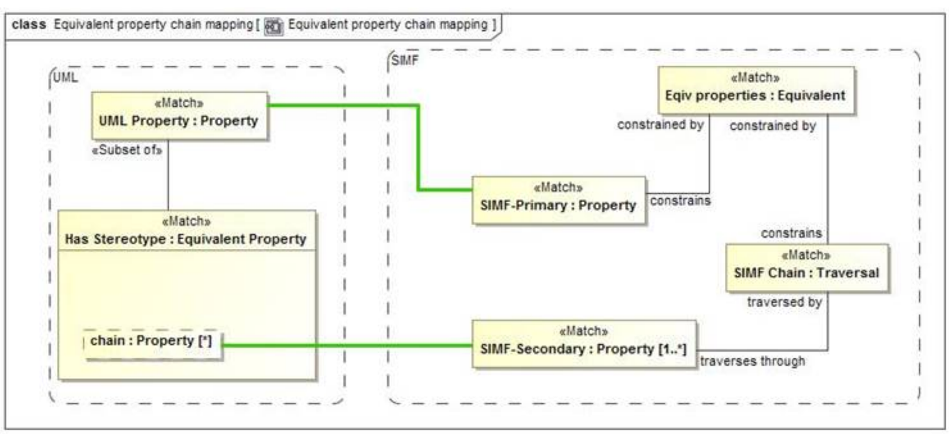


Figure 7. Equivalent property chain mapping

Equivalent property chain mapping maps the UML stereotype <<Equivalent Property>> with a "chain" tag to a SMIF "Equivalent" constraint constraining a SMIF Traversal of that chain.

The stereotyped property corresponds with the <constrains> property of the Equivalent constraint.

package SMIFProfileToModelMapping::Mapping rules

7.6.10 Class Equivalent property mapping

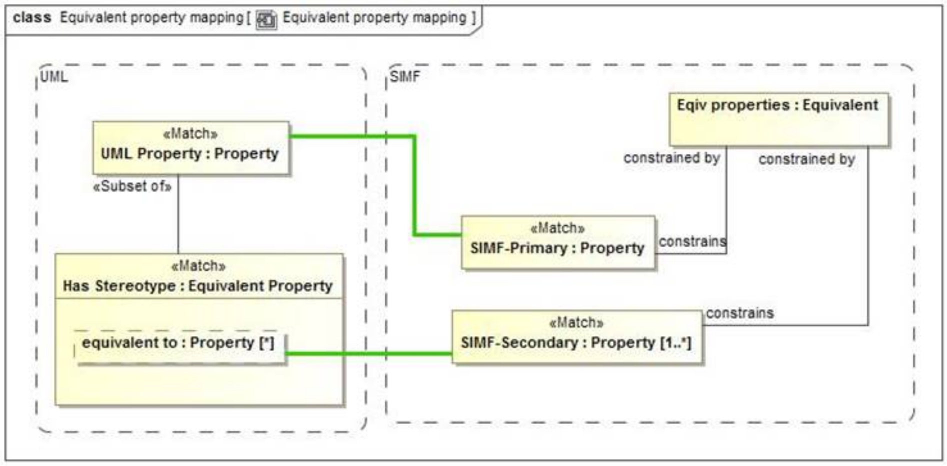


Figure 8. Equivalent property mapping

Equivalent property chain mapping maps the UML stereotype <<Equivalent Property>> with a <equivalent to> tag to a SMIF "Equivalent" constraint constraining a SMIF Secondary property.

The stereotyped property corresponds with the <constrains> property of the Equivalent constraint.

package SMIFProfileToModelMapping::Mapping rules

7.6.11 Class Equivalent with mapping

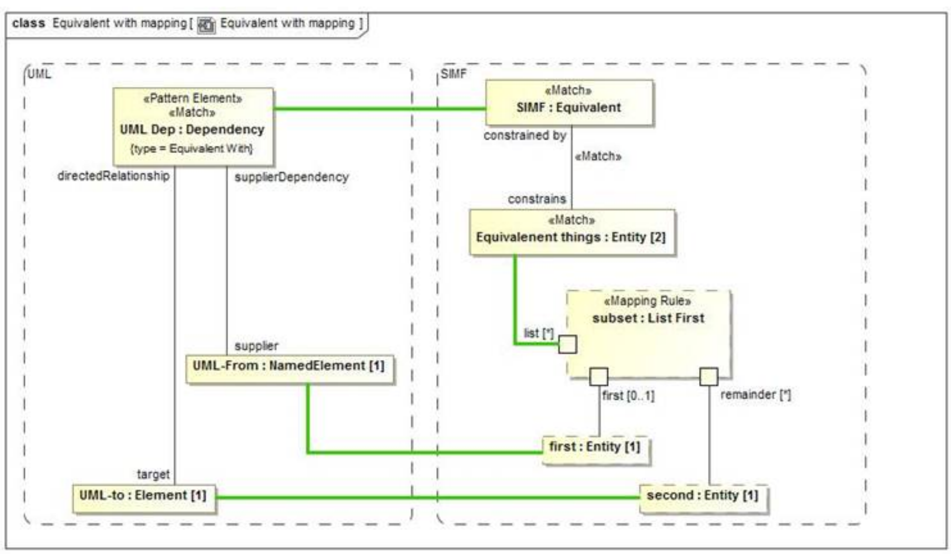


Figure 9. Equivalent with mapping

Equivalent with draws a correspondence between a UML dependency stereotyped as <<Equivalent With>> and a SMIF equivalent constraint with exactly 2 constrained elements. The first element is mapped to the dependency supplier and the second element to the dependency target.

The "List First" rule is used to divide the list elements.

Note that SMIF Equivalent constraints with more than 2 constrained elements will map to a generalization set.

package SMIFProfileToModelMapping::Mapping rules

7.6.12 Class Generalization mapping

The generalization mapping rule draws a direct correspondence between a UML Generalization and a SMIF Type Generalization Constraint.

The UML <<Facet Of>> constraint corresponds with the "as facet" boolean of the SMIF constraint.

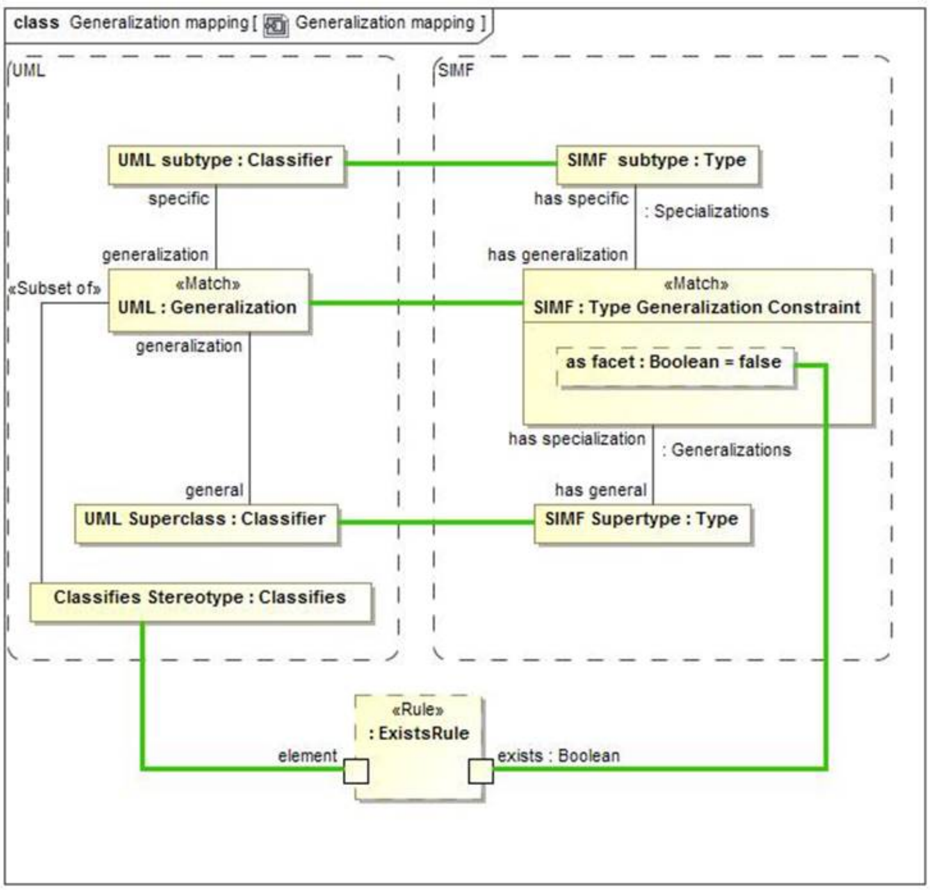


Figure 10. Generalization mapping

package SMIFProfileToModelMapping::Mapping rules

7.6.13 Class Generalization set covering mapping

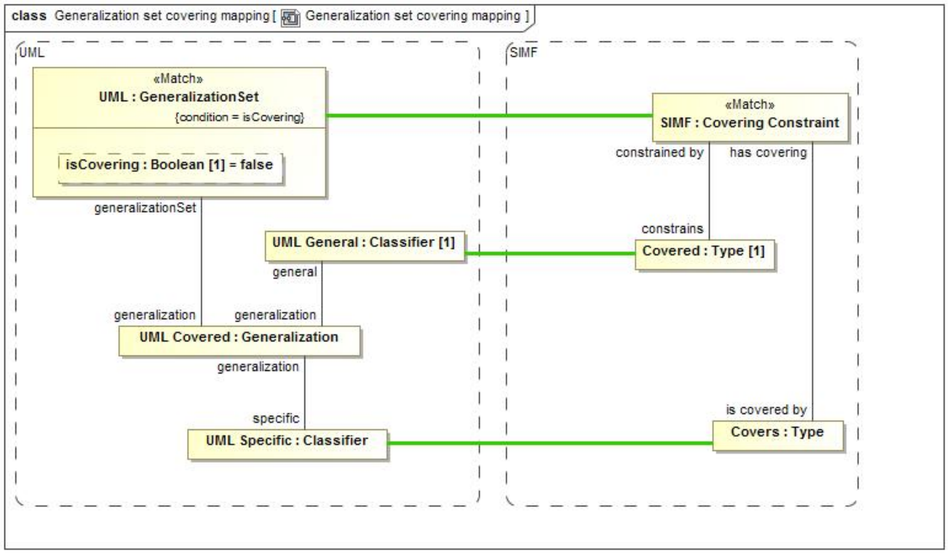


Figure 11. Generalization set covering mapping

Generalization set covering constraint draws a correspondence between a generalization set where isCovering is true and a SMIF Covering Constraint.

The generalization set has a set of UML covered <<Generalization>>s, all of which must have the same <general> Classifier, which is mapped to the <holds within> type.

The <specific> classifiers correspond to a set of <is covered by> types.

package SMIFProfileToModelMapping::Mapping rules

7.6.14 Class Generalization set disjoint mapping

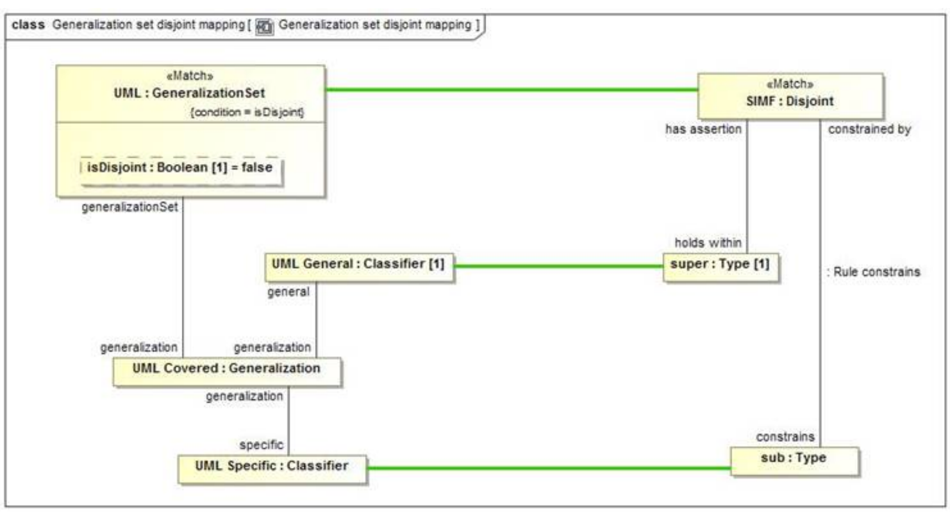


Figure 12. Generalization set disjoint mapping

Generalization set disjoint constraint draws a correspondence between a generalization set where isDisjoint is true and a SMIF Disjoint Constraint.

The generalization set has a set of UML covered <<Generalization>>s, all of which must have the same <general> Classifier, which is mapped to the <holds within> type.

The <specific> classifiers correspond to a set of <is covered by> types.

package SMIFProfileToModelMapping::Mapping rules

7.6.15 Class Is in context mapping

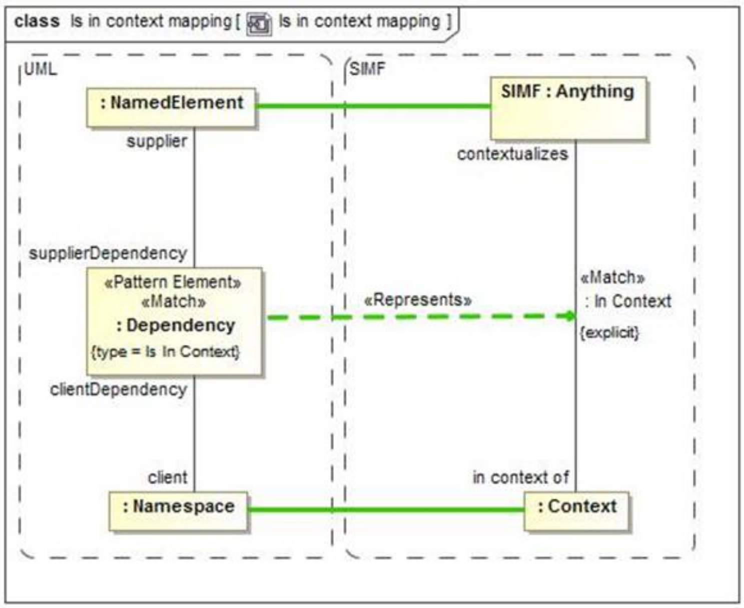


Figure 13. Is in context mapping

Is in contest mapping draws a correspondence between a UML dependency with a <<Is In CContext>> stereotype and an explicit "In Context" Relationship. The relationship is Explicit if it has been explicitly asserted, not inferred.

package SMIFProfileToModelMapping::Mapping rules

7.6.16 Class Mapping rule mapping

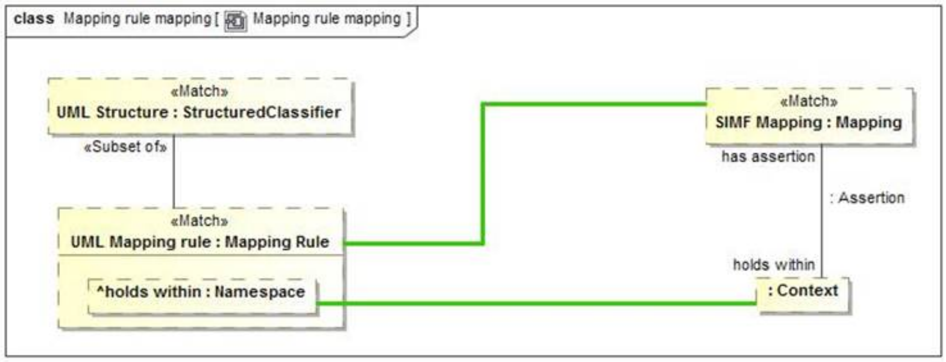


Figure 14. Mapping rule mapping

The Mapping rule mapping draws a correspondence between any UML StructuredClassifier marked as a <<Mapping Rule>> and a SMIF Mapping. The <holds within> tag then maps to the context the mapping holds within.

Any structured classifier may be created from SMIF.

package SMIFProfileToModelMapping::Mapping rules

7.6.17 Class Named element Mapping

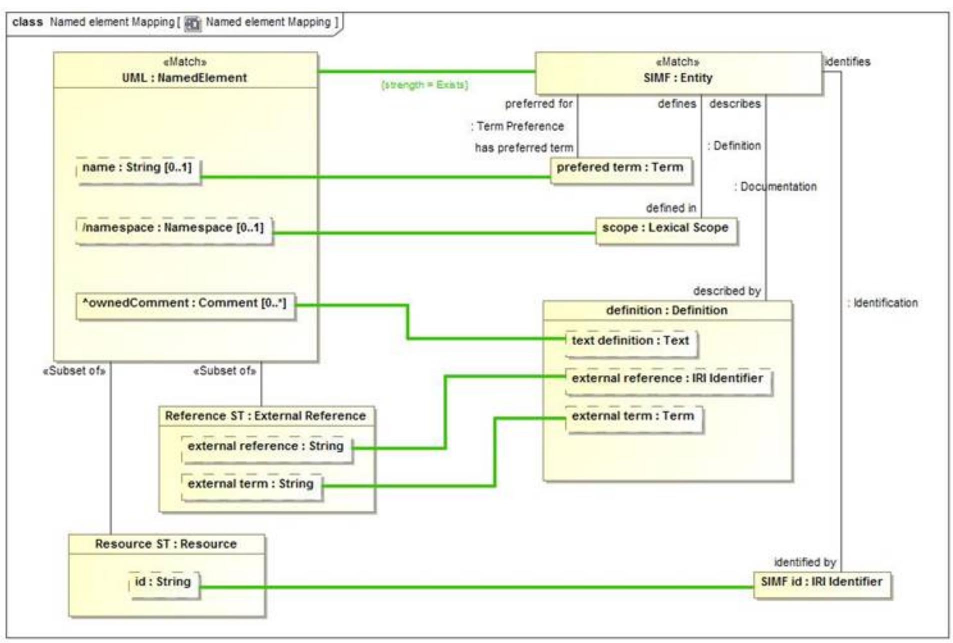


Figure 15. Named element Mapping

package SMIFProfileToModelMapping::Mapping rules

7.6.18 Class Pattern property mapping

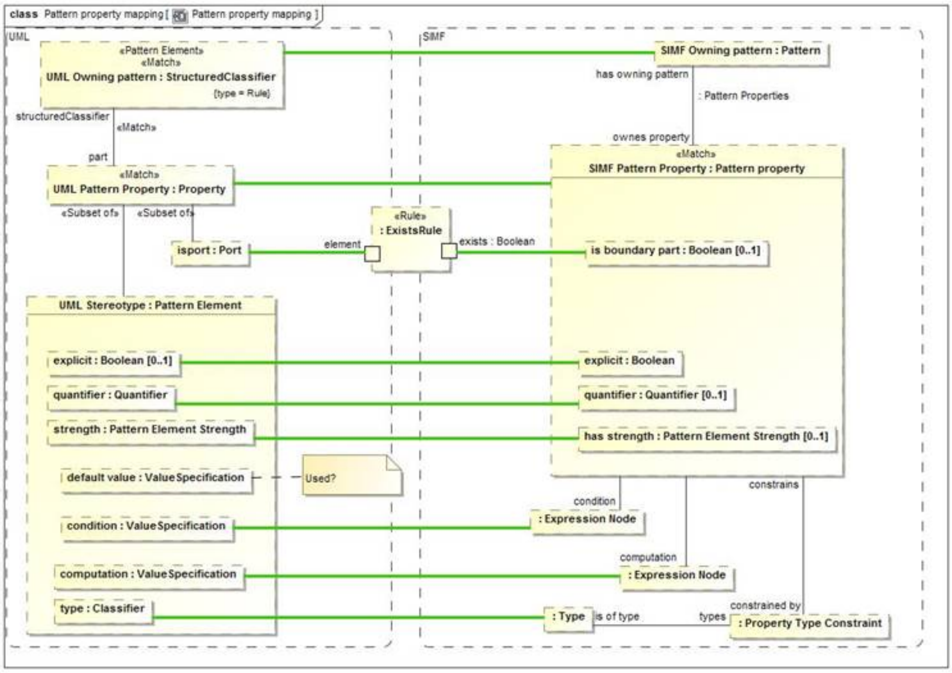


Figure 16. Pattern property mapping

Pattern property mapping makes a correspondence between any UML property that is <part> of a <Rule> and a SMIF Pattern Property.

The owning StructuredClassifier corresponds with the pattern that owns the Pattern Property.

If the UML Property is stereotyped as a <<Pattern Variable>> the tags correspond with the explicit, quantifier, condition, computation and has strength properties of the SMIF pattern property.

The <type> of the UML Pattern Variable is constrained to be a required <is of type> of the pattern property.

package SMIFProfileToModelMapping::Mapping rules

7.6.19 Class Property hierarchy mapping

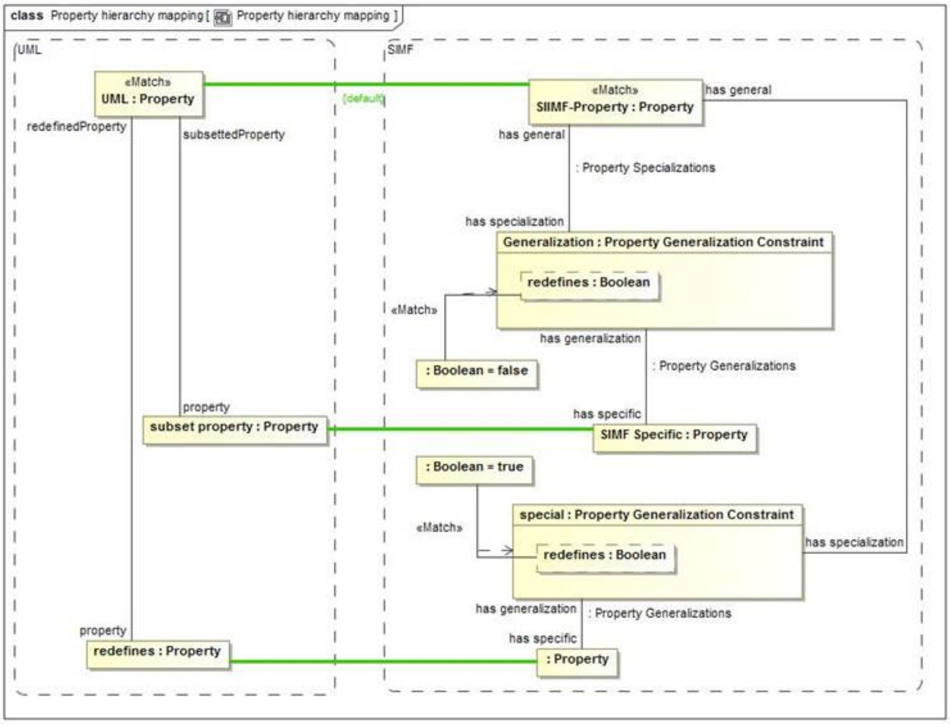


Figure 17. Property hierarchy mapping

Property hierarchy mapping, for properties that have been mapped in other ways, maps UML subsetted properties to a Property Generalization Constraint with redefines=false.

It also maps UML redefines properties to a Property Generalization Constraint with redefines=true.

package SMIFProfileToModelMapping::Mapping rules

7.6.20 Class Synonym mapping

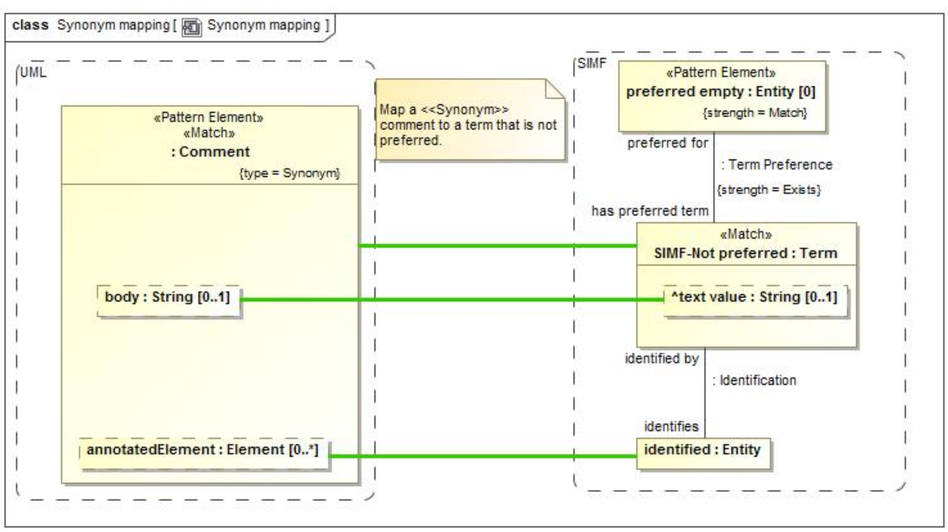


Figure 18. Synonym mapping

Synonym mapping maps a UML Comment with a <<Synonym>> stereotype to a SMIF Term that identifies the element that is annotated by the comment in UML.

A Synonym term is not a preferred term.

package SMIFProfileToModelMapping::Mapping rules

[1] https://en.wikipedia.org/wiki/Mars_Climate_Orbiter

[CC1]Need to nail down our terminology and use consistently.

[CC2]Pete asserts the semantics are different – I don't think so. Need to resolve.

[CC3]Redundant and not well worded

[CC4]This section is somewhat redundant with the specification introduction but you may want to include it for those only reading this section. However, it should be consistent.

[CC5]We have been using "real or a possible world"

[CC6]Perhaps a better example would be ones that did not share a direct common supertype – perhaps animal and mineral.

[CC7]By the way, when you paste in the diagrams – do it in SVG to make Andrew happy.

[CC8]Doesn't seem right. Both are manufacturers.

[CC9]Not a great example in that the set is not complete.

Perhaps Silverware: Knife, fork, spoon? (perhaps not exactly true but more true). Remember the "Spork"?

[CC10]Also not an example that fits the assertion. Perhaps the animal kingdom?? Or, Computer storage: Rotating or solid state?

[CC11]I would remove – not true for n-ary and would need to define term.

[CC12]Don't know that this is true, the representation should not change the semantics

[CC13]Need definitions

[CC14]May want to mention and make an example of the built-in annotation <<anything>> described by Definition. Note that the mapping supports <<Annotation>> of these and also maps the UML documentation element to a Definition which can be augmented by <<External Reference>>.

[CC15]From pete:
"is a comment an annotation property"? Make it clear it is an annotation value.

[CC16]☺ Seems odd.

[CC17]Not sure this constraint is needed.

[CC18]Why would we jump to that conclusion? Seems like a mxing of semantics and would be non-obvious.

[CC19]Redundant.

[CC20]Could use your generalization set here.

[CC21]Changed picture

[CC22] don't assume math people.

[CC23]Somewhere these need to be defines, perhaps this is the place.

[CC24]Out of date – also, mixing <<Anything>> with this example may be more challenging.

[CC25]Not true (was a change)

[CC26]May be to complex an example since you have others.

[CC27]Only if 1+

[CC28]Pete: Bad term

[CC29]Hmmm. Could it be sufficient and not necessary?

[CC30]Should show "1"

[CC31]Need to think about how much we -require-inference. I'm not sure we should here. It could be just a model validity check, other implementations could do inference. I think we should keep this very open. For one thing, this could make it fall into the "ontology" category. This is true in other places as well.

[CC32]Said again.

[CC33]Again

[CC34]Pete: still not clear whether the sufficient condition is to have a contract with a Steering Wheel AND a Windshield manufacturer

[CC35]Out of date

[CC36]Can we not use the term in its definition?? Perhaps "be defining a path through other properties that have the same starting point.

[CC37]Pete: not in example; in any case surely it will be applied to a property not a class

[CC38]More accurately – it has one tag value with 2 or more elements.

[CC39]Don't know where this restriction came from, seems counter-productive and may prevent some of the OntoUML restrictions. REMOVE!

[CC40]Should show for example. Also show property chain ST

[CC41]Pete: how does this relate to previous section using the same stereotype?

[CC42]Perhaps you could be less circular?

[CC43]Again, don't know where this restriction comes from or why.

[CC44]Pete: not a good example - these are synonyms

[CC45]Remove or make optional to stay within normal UML

[CC46]More – what is a global property?

[CC47]facet (including roles and phases)

[CC48]<<Facet Of>>

[CC49]facet

[CC50]facet

[CC51]facets

[CC52]tbd

[CC53]tbd

[CC54]below it says [RFC3987] We should be consistent.

[CC55]Tbd – really need to nail down.

[CC56]A bit hard to read

[CC57]Pete: not so simple - semiotic triangle involved

consider putting in semiotic triangle picture and explanation.

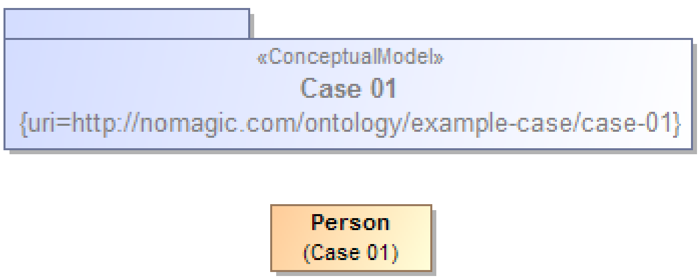
[CC58]Consider name change for one or the other

8 SMIF Mapping to OWL 2 (normative)

Examples are given below that show the transformation of UML modeled in SMIF to an exported OWL 2 ontology. The OWL ontologies are presented in OWL Functional Syntax.

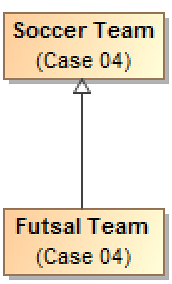
The first diagram below, for a simple UML class, shows the ontology is transformed as the package containing the UML class. Subsequent diagrams do not show the package in the diagram for the sake of brevity.

8.1 Class



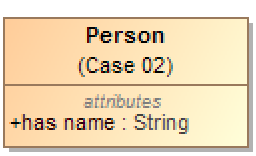

```
Ontology(<http://nomagic.com/ontology/example-case/case-01>
	Declaration(
		Class(:Person)
	)
	AnnotationAssertion(rdfs:label :Person "Person"@en)
)
```

8.2 Class Generalization



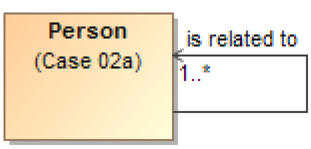

```
Ontology(<http://nomagic.com/ontology/example-case/case-04>
	Declaration(
		Class(:FutsalTeam)
	)
	Declaration(
		Class(:SoccerTeam)
	)
	AnnotationAssertion(rdfs:label :FutsalTeam "Futsal Team"@en)
	SubClassOf(:FutsalTeam :SoccerTeam)
	AnnotationAssertion(rdfs:label :SoccerTeam "Soccer Team"@en)
)
```

8.3 Class with Datatype Property



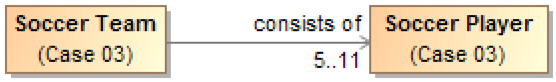
```
Ontology(<http://nomagic.com/ontology/example-case/case-02>
       Import(<http://www.omg.org/spec/PrimitiveTypes/20100901>)
       Declaration(
              Class(:Person)
       )
       Declaration(
              DataProperty(:hasName)
       )
       Declaration(
              AnnotationProperty(<http://purl.org/dc/terms/description>)
       )
       Declaration(
              Datatype(xsd:string)
              )
       AnnotationAssertion(rdfs:label :Person "Person"@en)
       SubClassOf(
              :Person
              ObjectIntersectionOf(
                     DataMaxCardinality(1 :hasName xsd:string)
                     DataMinCardinality(1 :hasName xsd:string)
              )
       )
       AnnotationAssertion(rdfs:label :hasName "has name"@en)
       DataPropertyDomain(:hasName :Person)
       DataPropertyRange(:hasName xsd:string)
       AnnotationAssertion(http://purl.org/dc/terms/description
       <http://www.omg.org/spec/PrimitiveTypes/20100901#String> "An instance of String
       defines a piece of text. The semantics of the string itself depends on its
       purpose, it can be a comment, computational language expression, OCL expression,
       etc. It is used for String attributes and String expressions in the
       metamodel."@en)
)
```

8.4 Class with Self-Referential Object Property



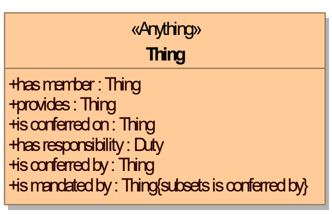

```
Ontology(<http://nomagic.com/ontology/example-case/case-02a>
       Declaration(
              Class(:Person)
       )
       Declaration(
              ObjectProperty(:isRelatedTo)
       )
       AnnotationAssertion(rdfs:label :Person "Person"@en)
       SubClassOf(
              :Person
              ObjectIntersectionOf(
                     ObjectMinCardinality(1 :isRelatedTo :Person)
              )
       )
       AnnotationAssertion(rdfs:label :isRelatedTo "is related to"@en)
       ObjectPropertyDomain(:isRelatedTo :Person)
       ObjectPropertyRange(:isRelatedTo :Person)
)
```

8.5 Class with Object Property




```
Ontology(<http://nomagic.com/ontology/example-case/case-03>
       Declaration(
              Class(:SoccerPlayer)
       )
       Declaration(
              Class(:SoccerTeam)
       )
       Declaration(
              ObjectProperty(:consistsOf)
       )
       AnnotationAssertion(rdfs:label :SoccerPlayer "Soccer Player"@en)
       AnnotationAssertion(rdfs:label :SoccerTeam "Soccer Team"@en)
       SubClassOf(
              :SoccerTeam
              ObjectIntersectionOf(
                     ObjectMaxCardinality(11 :consistsOf :SoccerPlayer)
                     ObjectMinCardinality(5 :consistsOf :SoccerPlayer)
              )
       )
       AnnotationAssertion(rdfs:label :consistsOf "consists of"@en)
       ObjectPropertyDomain(:consistsOf :SoccerTeam)
       ObjectPropertyRange(:consistsOf :SoccerPlayer)
)
```

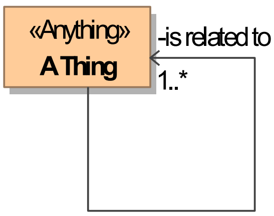
8.6 <<Anything>> with Datatype Property




```
Ontology(<http://nomagic.com/ontology/example-case/case-03a>
       Import(<http://www.omg.org/spec/PrimitiveTypes/20100901>)
       Declaration(
              DataProperty(:hasName)
       )
       Declaration(
              AnnotationProperty(<http://purl.org/dc/terms/description>)
       )
       Declaration(
              Datatype(xsd:string)
       )
       SubClassOf(
              owl:Thing
              ObjectIntersectionOf(
                     DataMaxCardinality(3 :hasName xsd:string)
                     DataMinCardinality(2 :hasName xsd:string)
              )
       )
       AnnotationAssertion(rdfs:label :hasName "has name"@en)
       DataPropertyRange(:hasName xsd:string)
       AnnotationAssertion(http://purl.org/dc/terms/description
       <http://www.omg.org/spec/PrimitiveTypes/20100901#String> "An instance of String
       defines a piece of text. The semantics of the string itself depends on its
```

```
        purpose, it can be a comment, computational language expression, OCL expression,
        etc. It is used for String attributes and String expressions in the
        metamodel."@en)
)
```

8.7 <<Anything>>with Self-Referential Object Property

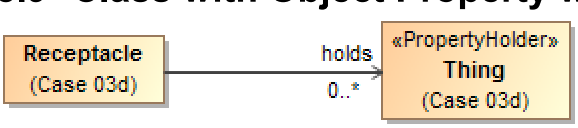



```
Ontology(<http://nomagic.com/ontology/example-case/case-03b>
        Declaration(
                ObjectProperty(:isRelatedTo)
        )
        SubClassOf(
                owl:Thing
                ObjectIntersectionOf(
                        ObjectMinCardinality(1 :isRelatedTo)
                )
        )
        AnnotationAssertion(rdfs:label :isRelatedTo "is related to"@en)
)
```

8.8 <<Anything>> with Object Property


```
        Ontology(<http://nomagic.com/ontology/example-case/case-03c>
                Declaration(
                        Class(:Liquid)
                )
                Declaration(
                        ObjectProperty(:isDissolvedBy)
                )
                AnnotationAssertion(rdfs:label :Liquid "Liquid"@en)
                SubClassOf(
                        owl:Thing
                        ObjectIntersectionOf(
                                ObjectMinCardinality(1 :isDissolvedBy :Liquid)
                        )
                )
                AnnotationAssertion(rdfs:label :isDissolvedBy "is dissolved by"@en)
                ObjectPropertyRange(:isDissolvedBy :Liquid)
        )
```

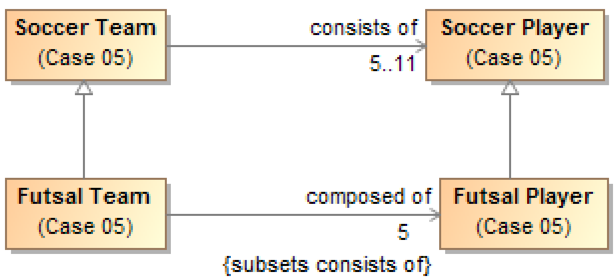
8.9 Class with Object Property without Range




```
Ontology(<http://nomagic.com/ontology/example-case/case-03d>
        Declaration(
```

```
		Class(:Receptacle)
	)
	Declaration(
		ObjectProperty(:holds)
	)
	AnnotationAssertion(rdfs:label :Receptacle "Receptacle"@en)
	AnnotationAssertion(rdfs:label :holds "holds"@en)
	ObjectPropertyDomain(:holds :Receptacle)
)
```

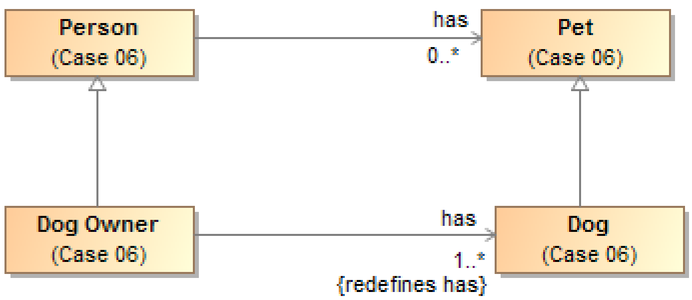
8.10 Class with Subproperty




```
Ontology(<http://nomagic.com/ontology/example-case/case-05>
	Declaration(
		Class(:FutsalPlayer)
	)
	Declaration(
		Class(:FutsalTeam)
	)
	Declaration(
		Class(:SoccerPlayer)
	)
	Declaration(
		Class(:SoccerTeam)
	)
	Declaration(
		ObjectProperty(:composedOf)
	)
	Declaration(
		ObjectProperty(:consistsOf)
	)
	AnnotationAssertion(rdfs:label :FutsalPlayer "Futsal Player"@en)
	SubClassOf(:FutsalPlayer :SoccerPlayer)
	AnnotationAssertion(rdfs:label :FutsalTeam "Futsal Team"@en)
	SubClassOf(:FutsalTeam :SoccerTeam)
	SubClassOf(
		:FutsalTeam
		ObjectIntersectionOf(
			ObjectMaxCardinality(5 :composedOf :FutsalPlayer)
			ObjectMinCardinality(5 :composedOf :FutsalPlayer)
		)
	)
	AnnotationAssertion(rdfs:label :SoccerPlayer "Soccer Player"@en)
	AnnotationAssertion(rdfs:label :SoccerTeam "Soccer Team"@en)
	SubClassOf(
		:SoccerTeam
		ObjectIntersectionOf(
			ObjectMaxCardinality(11 :consistsOf :SoccerPlayer)
			ObjectMinCardinality(5 :consistsOf :SoccerPlayer)
		)
	)
	AnnotationAssertion(rdfs:label :composedOf "composed of"@en)
```

```
        SubObjectPropertyOf(:composedOf :consistsOf)
        ObjectPropertyDomain(:composedOf :FutsalTeam)
        ObjectPropertyRange(:composedOf :FutsalPlayer)
        AnnotationAssertion(rdfs:label :consistsOf "consists of"@en)
        ObjectPropertyDomain(:consistsOf :SoccerTeam)
        ObjectPropertyRange(:consistsOf :SoccerPlayer)
)
```

8.11 Class with Universal Quantification Constraint on Property I

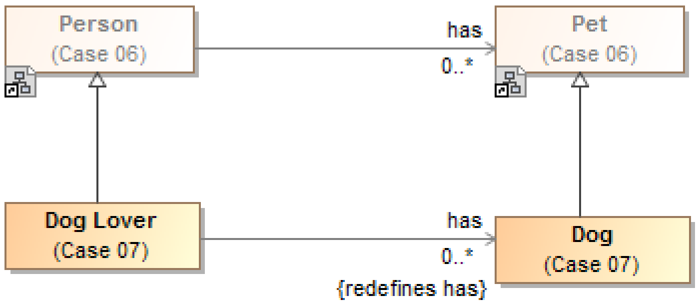



```
Ontology(<http://nomagic.com/ontology/example-case/case-06>
        Declaration(
                Class(:Dog)
        )
        Declaration(
                Class(:DogOwner)
        )
        Declaration(
                Class(:Person)
        )
        Declaration(
                Class(:Pet)
        )
        Declaration(
                ObjectProperty(:has)
        )
        AnnotationAssertion(rdfs:label :Dog "Dog"@en)
        SubClassOf(:Dog :Pet)
        AnnotationAssertion(rdfs:label :DogOwner "Dog Owner"@en)
        SubClassOf(:DogOwner :Person)
        SubClassOf(
                :DogOwner
                ObjectIntersectionOf(
                        ObjectMinCardinality(1 :has :Dog)
                        ObjectAllValuesFrom(:has :Dog)
                )
        )
        AnnotationAssertion(rdfs:label :Person "Person"@en)
        AnnotationAssertion(rdfs:label :Pet "Pet"@en)
        AnnotationAssertion(rdfs:label :has "has"@en)
        ObjectPropertyDomain(:has :Person)
        ObjectPropertyRange(:has :Pet)
)
```

8.12 Class with Universal Quantification Constraint on Property II

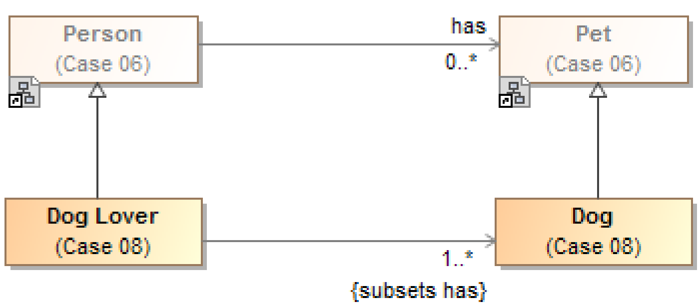
This example differs from the previous example primarily in that the superclasses “Person” and “Pet” are from a different package than their subclasses “Dog Lover” and “Dog,” respectively. This is reflected in the OWL ontology by the import of this namespace.

The superclasses “Person” and “Pet”, defined in the package “Case 06”, are a different color and a lighter shade than the classes defined in the package “Case 07”. This is to distinguish them from the classes defined in this package. MagicDraw’s AutoStyler plugin can automatically set the properties for classes and other UML elements “defined elsewhere,” that is in a package not containing the defining diagram for the UML element (See section 2.2, Automatic Styling of Concept Models.).




```
Ontology(<http://nomagic.com/ontology/example-case/case-07>
	Import(<http://nomagic.com/ontology/example-case/case-06>)
	Declaration(
		Class(<http://nomagic.com/ontology/example-case/case-06#Person>)
	)
	Declaration(
		Class(<http://nomagic.com/ontology/example-case/case-06#Pet>)
	)
	Declaration(
		Class(:Dog)
	)
	Declaration(
		Class(:DogLover)
	)
	Declaration(
		ObjectProperty(<http://nomagic.com/ontology/example-case/case-06#has>)
	)
	AnnotationAssertion(rdfs:label :Dog "Dog"@en)
	SubClassOf(:Dog <http://nomagic.com/ontology/example-case/case-06#Pet>)
	AnnotationAssertion(rdfs:label :DogLover "Dog Lover"@en)
	SubClassOf(:DogLover <http://nomagic.com/ontology/example-case/case-06#Person>)
	SubClassOf(
		:DogLover ObjectIntersectionOf(
			ObjectAllValuesFrom(<http://nomagic.com/ontology/example-case/case-
			06#has> :Dog)
		)
	)
)
```

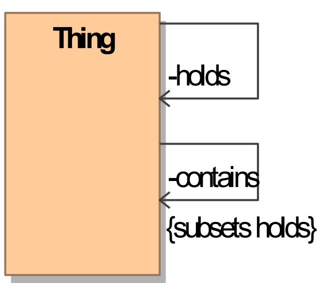
8.13 Class with Existential Quantification Constraint on Property




```
Ontology(<http://nomagic.com/ontology/example-case/case-08>
	Import(<http://nomagic.com/ontology/example-case/case-06>)
```

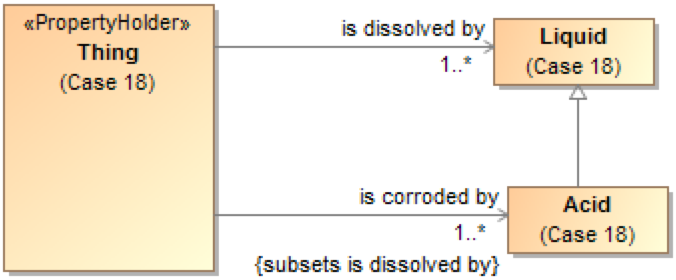
```
	Declaration(
		Class(<http://nomagic.com/ontology/example-case/case-06#Person>)
	)
	Declaration(
		Class(<http://nomagic.com/ontology/example-case/case-06#Pet>)
	)
	Declaration(
		Class(:Dog)
	)
	Declaration(
		Class(:DogLover)
	)
	Declaration(
		ObjectProperty(<http://nomagic.com/ontology/example-case/case-06#has>)
	)
	AnnotationAssertion(rdfs:label :Dog "Dog"@en)
	SubClassOf(:Dog <http://nomagic.com/ontology/example-case/case-06#Pet>)
	AnnotationAssertion(rdfs:label :DogLover "Dog Lover"@en)
	SubClassOf(:DogLover <http://nomagic.com/ontology/example-case/case-06#Person>)
	SubClassOf(
		:DogLover
		ObjectIntersectionOf(
			ObjectMinCardinality(1 <http://nomagic.com/ontology/example-
			case/case-06#has> :Dog)
			ObjectSomeValuesFrom(<http://nomagic.com/ontology/example-case/case-
			06#has> :Dog)
		)
	)
)
```

8.14 <<Anything>> with Self-Referential Subproperty



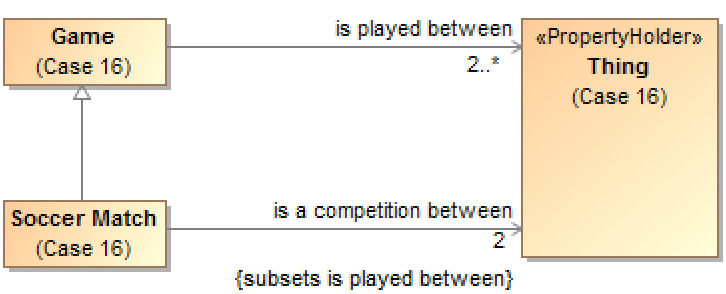

```
Ontology(<http://nomagic.com/ontology/example-case/case-11>
	Declaration(
		ObjectProperty(:contains)
	)
	Declaration(
		ObjectProperty(:holds)
	)
	AnnotationAssertion(rdfs:label :contains "contains"@en)
	SubObjectPropertyOf(:contains :holds)
	AnnotationAssertion(rdfs:label :holds "holds"@en)
)
```

8.15 <<Anything>> Holder with Subproperty




```
Ontology(<http://nomagic.com/ontology/example-case/case-18>
	Declaration(
		Class(:Acid)
	)
	Declaration(
		Class(:Liquid)
	)
	Declaration(
		ObjectProperty(:isCorrodedBy)
	)
	Declaration(
		ObjectProperty(:isDissolvedBy)
	)
	AnnotationAssertion(rdfs:label :Acid "Acid"@en)
	SubClassOf(:Acid :Liquid)
	AnnotationAssertion(rdfs:label :Liquid "Liquid"@en)
	SubClassOf(
		owl:Thing
		ObjectIntersectionOf(
			ObjectMinCardinality(1 :isCorrodedBy :Acid)
		)
	)
	SubClassOf(
		owl:Thing
		ObjectIntersectionOf(
			ObjectMinCardinality(1 :isDissolvedBy :Liquid)
		)
	)
	AnnotationAssertion(rdfs:label :isCorrodedBy "is corroded by"@en)
	SubObjectPropertyOf(:isCorrodedBy :isDissolvedBy)
	ObjectPropertyRange(:isCorrodedBy :Acid)
	AnnotationAssertion(rdfs:label :isDissolvedBy "is dissolved by"@en)
	ObjectPropertyRange(:isDissolvedBy :Liquid)
)
```

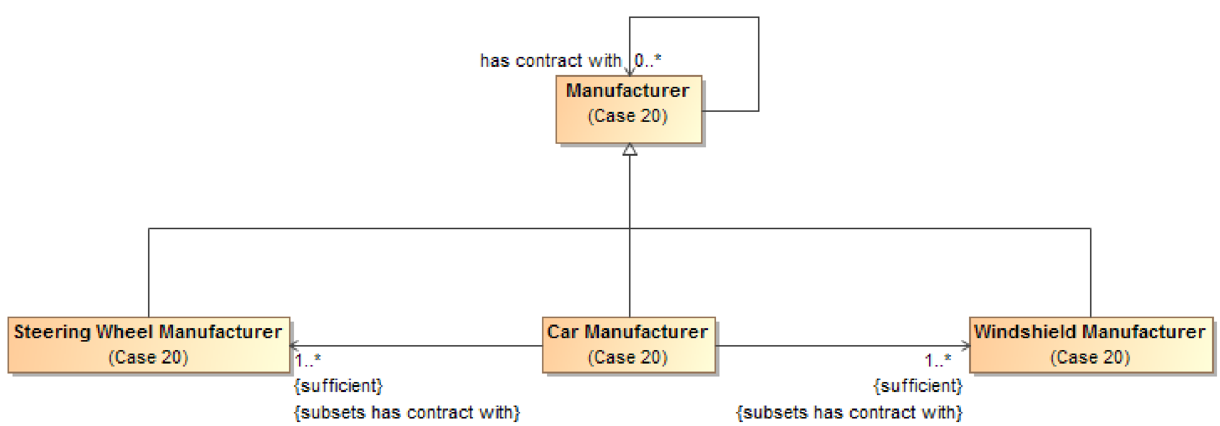
8.16 Class with Subproperty without a Range




```
Ontology(<http://nomagic.com/ontology/example-case/case-16>
	Declaration(
```

```
        Class(:Game)
    )
    Declaration(
        Class(:SoccerMatch)
    )
    Declaration(
        ObjectProperty(:isACompetitionBetween)
    )
    Declaration(
        ObjectProperty(:isPlayedBetween)
    )
    AnnotationAssertion(rdfs:label :Game "Game"@en)
    SubClassOf(
        :Game
        ObjectIntersectionOf(
            ObjectMinCardinality(2 :isPlayedBetween)
        )
    )
    AnnotationAssertion(rdfs:label :SoccerMatch "Soccer Match"@en)
    SubClassOf(:SoccerMatch :Game)
    SubClassOf(
        :SoccerMatch
        ObjectIntersectionOf(
            ObjectMaxCardinality(2 :isACompetitionBetween) ObjectMinCardinality(2
            :isACompetitionBetween)
        )
)
    AnnotationAssertion(rdfs:label :isACompetitionBetween "is a competition
    between"@en)
    SubObjectPropertyOf(:isACompetitionBetween :isPlayedBetween)
    ObjectPropertyDomain(:isACompetitionBetween :SoccerMatch)
    AnnotationAssertion(rdfs:label :isPlayedBetween "is played between"@en)
    ObjectPropertyDomain(:isPlayedBetween :Game)
)
```

8.17 Class with Necessary and Sufficient Property

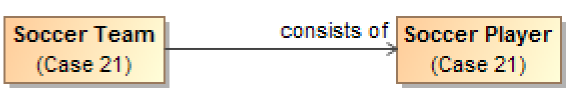



```
Ontology(<http://nomagic.com/ontology/example-case/case-20>
    Declaration(
        Class(:CarManufacturer)
    )
    Declaration(
        Class(:Manufacturer)
    )
    Declaration(
```

```
        Class(:SteeringWheelManufacturer)
    )
    Declaration(
        Class(:WindshieldManufacturer)
    )
    Declaration(
        ObjectProperty(:hasContractWith)
    )
    AnnotationAssertion(rdfs:label :CarManufacturer "Car Manufacturer"@en)
    EquivalentClasses(
        :CarManufacturer
        ObjectIntersectionOf(
            ObjectMinCardinality(1 :hasContractWith :SteeringWheelManufacturer)
            ObjectSomeValuesFrom(:hasContractWith :SteeringWheelManufacturer)
        )
    )
    EquivalentClasses(
        :CarManufacturer
        ObjectIntersectionOf(
            ObjectMinCardinality(1 :hasContractWith :WindshieldManufacturer)
            ObjectSomeValuesFrom(:hasContractWith :WindshieldManufacturer)
        )
    )
    SubClassOf(:CarManufacturer :Manufacturer)
    AnnotationAssertion(rdfs:label :Manufacturer "Manufacturer"@en)
    AnnotationAssertion(rdfs:label :SteeringWheelManufacturer "Steering Wheel
    Manufacturer"@en)
    SubClassOf(:SteeringWheelManufacturer :Manufacturer)
    AnnotationAssertion(rdfs:label :WindshieldManufacturer "Windshield
    Manufacturer"@en)
    SubClassOf(:WindshieldManufacturer :Manufacturer)
    AnnotationAssertion(rdfs:label :hasContractWith "has contract with"@en)
    ObjectPropertyDomain(:hasContractWith :Manufacturer)
    ObjectPropertyRange(:hasContractWith :Manufacturer)
)
```

8.18 Class With Property Having Unspecified Multiplicity

UML allows the cardinality of a property to be left unspecified. The concept modeling profile interprets unspecified cardinalities as being zero to many (“0..*”).




```
Ontology(<http://nomagic.com/ontology/example-case/case-21>
    Declaration(
        Class(:SoccerPlayer)
    )
    Declaration(
        Class(:SoccerTeam)
    )
    Declaration(ObjectProperty(:consistsOf))
    AnnotationAssertion(rdfs:label :SoccerPlayer "Soccer Player"@en)
    AnnotationAssertion(rdfs:label :SoccerTeam "Soccer Team"@en)
    AnnotationAssertion(rdfs:label :consistsOf "consists of"@en)
    ObjectPropertyDomain(:consistsOf :SoccerTeam)
    ObjectPropertyRange(:consistsOf :SoccerPlayer)
)
```

Alphabetical Index