ArchMind

Software Product Model

An ontology and metamodel for documenting non-trivial software

Jayson Go, ArchMind

The benefit of software documentation is compromised, and in its depreciated form introduces problems and risks to software delivery. And with time, these issues are exacerbated into organizational problems or crisis. I provide a solution for non-trivial software to improve two determinants to this benefit: model precision and structural integrity. To improve the precision of what is being delivered, my strategy is to prevent model deficit by describing an ontology that identifies orthogonal stakeholder groups, their assigned models and the maps between models. To enhance the integrity of software structures, my strategy is to prohibit structural deficiencies by defining a metamodel that derives software structures from composition. The metamodel uses behavior as a foundational concept and capabilities as fundamental elements. My approach is inspired by abstract mathematics where Category Theory is applied to the ontology and function composition is utilized in the metamodel. I extend the basic metamodel with additional and updated elements, including novel techniques with algorithmic elements, to support pragmatic applications of software architecture and engineering.

jayson.go@archmind.io

Contents

1. Introduction

Software documentation is an essential documentation practice that enables various functions in a software delivery cycle. It has come to provide its audience with highly technical documents. But while its adamant focus on the logical and physical aspects of the software system has worked sufficiently for many organizations, its benefits are compromised because of deficiencies in and of its models. In software documentation, the absence of a usable model that informs or enforces the technical models causes imprecision, and such technical models are yielding weak software structures due to impaired dependencies.

Software delivery cycles can vary from one organization to another but it generally includes a delivery phase consisting of various functions such as implementation, governance or strategic planning. These functions rely heavily on complete and coherent documentation artifacts to fulfill their goals. When models are deficient, these functions can lead to incorrect implementation, flawed governance or poorly planned strategies. The deficiencies are caused by a deficit in models and structurally-weak software structures.

Model Deficit

Software documentation has a model deficit when it lacks a complete set of models. Every software has stakeholders and they can be grouped based on their orthogonal needs, concerns or interests. To appropriately represent their stake, to describe and share their ideas, models are assigned to stakeholder groups. When stakeholder-group-assigned models are not included or but exists in low utility, stakeholder representation is diminished which can lead to missed requirements. This is especially true when the missing model precedes the technical models because its absence would prevent it from informing or enforcing the logical or physical models.

Structural Weakness

Software documentation produces structurally-weak software structures when it is based on impaired dependencies. The structure of software is a graph of its software components and its *shape* is determined by its relationships. Relationships are based on dependencies but they can be impaired when their validity is neglected or left unconfirmed. It is logically insufficient to simply imply or assume a dependency is valid

by its mere declaration; a common practice when ubiquitous boxes and line diagrams are drawn. Consider the following factors of validity when declaring a dependency:

a. The dependent component may not have all the necessary *ingredients* to use the target component's capability

b. The target component may not at all contain the desired capability that the dependent component intends to use

c. The dependent component may not have the necessary means to communicate with the target component's interface

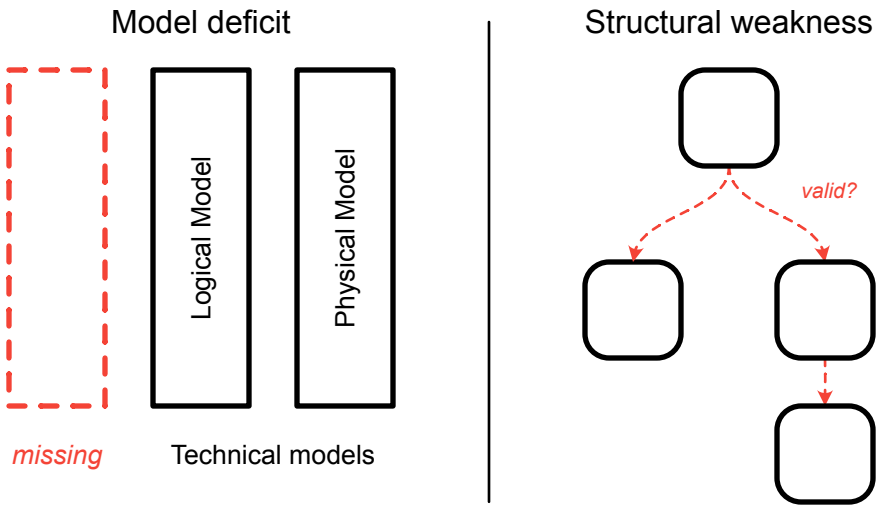

Figure 1. **Overview of deficiencies**. Software documentation's precision and integrity are negatively affected by model deficit and weak software structures.

Model deficit and structural weakness are deficiencies in software documentation and they result in imprecise and weak designs. It devalues the benefits of the practice especially during delivery. And, time is an exacerbating factor that amplifies the design imprecision or lack of structural integrity. Over periods of repeated updates, caused by for example changing requirements, these deficiencies turn into bigger problems that can lead to organizational crisis. Consider that assumptions (of dependency validity) in a earlier designs may become regarded as facts in later designs.

1.1. Solution

Given the deficiencies in software documentation, I propose a solution to the model deficit and weak-structures problems for a specific class of software. My strategy isolates to a type of non-trivial software to induce the primary stakeholder groups to be deterministic; as being finite and final. And, my approach is inspired by mathematical concepts that help to support my conclusions. The solution is named the Software Product Model which is a top-level model for software products which consists of an ontology and metamodel.

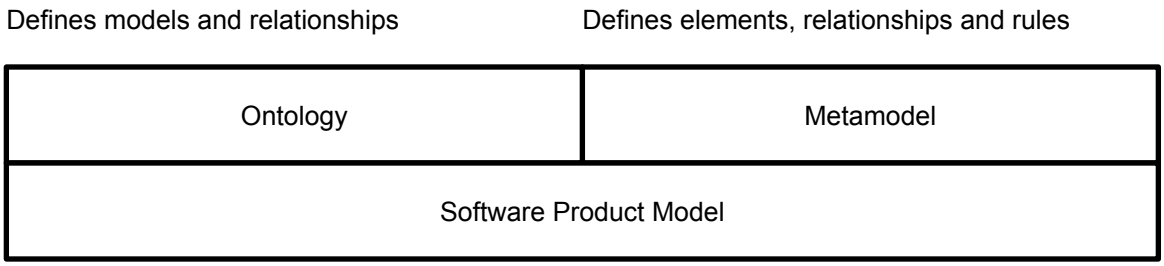

Figure 2. **Overview of the Software Product Model**. The Software Product Model consists of an ontology and metamodel that aims to provide a complete set of models and high-integrity structures.

Software Product Ontology. The ontology provides meaning to the software product by defining the required models and their relationships. It requires the conceptual, logical and physical models and arranges them in a chain according to their dependencies. Consequently, two maps are also defined by the ontology. The conceptual-to-logical map is named Architecture and the logical-to-physical map is named Engineering. With a mathematical approach, I loosely apply Category Theory by treating the software product as a category (Stanford University, 1996), the models as objects within the category and the maps, Architecture and Engineering, as the morphisms. Math can be regarded as the logical study of how logical things work (Cheng, 2018). By defining a complete set of models, the model deficit is removed and technical models become dependent to the conceptual model.

Software Product Metamodel. The metamodel defines elements and attributes, and their elemental relationships across all models. Behavior is the foundational concept of

the metamodel and it is represented logically by a fundamental element called the Capability. Capabilities are functional, composable and stateful by emulating mathematical and programming concepts. It is like a mathematical function that can accept inputs, produce outputs or compose with other capabilities. A capability is also like a software program that executes in a software environment where state is able to be durable and persist beyond execution. Finally, the composable trait of capabilities allows it to derive dependencies, and from dependencies derive a structure. The derivation process provides dependency validity because composition is input- and output-aware. The metamodel also introduces a novel element that can compute or expand structures.

A complete set of related and enforced models, and a metamodel that can derive structurally strong software structures will solve the deficiencies in software documentation. However, in its basic form, the metamodel is not practical enough for any organizational need. To complete the solution, I allow the metamodel to be extensible with new elements, relationships or attributes. I constrain extensibility to only within the maps to avoid unnecessary extensions. As a result, in this paper, I also provide metamodel extensions to enable pragmatic applications of ideation, software architecture and engineering. Section 5 covers ideation extensions that enable better definitions of business processes. Section 6 covers the architectural extensions that enable quality and distributive designs, and Section 7 covers the engineering extensions that enable teaming, versioning and standards.

1.2. Limitations

The solution is limited by the lack of bespoke tooling. The Software Product Model as a comprehensive documentation solution partly requires computations, whereas other documentation practices do not. Today's tools used for software documentation are optimized for drawing and diagramming. The model solution requires computation to derive dependencies from composition, quality benchmarks and behavioral distribution with tiers and capability distribution. At the general period of publication of this paper, there are no known tools that provide any of the necessary computational features. Therefore, the implementation of the Software Product Model will be inefficient until sufficient tooling is available.

2. Software Product

Software can be bi-classified as being trivial or non-trivial based on factors such as complexity and utility. Alternatively, the number of stakeholder groups aggregates those factors and can be used as a single basis for the classification. The number of stakeholder groups is subject to increase if the software becomes more complex because it may require more people or teams to support its construction or maintenance. Similarly, the number can also grow if the software becomes more useful because more people or teams may want to use, monetize or procure it. Therefore, as the number of stakeholder groups increase, software becomes less trivial.

A Software Product is a type of non-trivial software with exactly three stakeholder groups: owners, architects and engineers. The product moniker refers to software that provides ample value to warrant the expenditure of time, money or other resources. It implies that the software must have sufficient utility and complexity. The utility of the software indicates that it is useful to people solving a real-world problem, and its complexity asserts that there is sufficient interest in its design and specification. The software product is conceptually similar to Brooks' (1995) definition of a programming product as something that is run, tested, repaired and extended.

2.1. Stakeholder Groups

Stakeholders are people who have interests or concerns of or about software. An inherent stakeholder might be the developer who creates the software. A *Hello World* program, for example, is a type of trivial software that only has the developer as the sole stakeholder. Alternatively, non-trivial software will have additional stakeholders. Stakeholders can be grouped according to their orthogonal needs, interests or concerns to construct an abstract set of stakeholders. Stakeholders are important because they drive the whole shape and direction of architecture (Rozanski and Woods, 2016), described as the map between the conceptual and logical models (Section 3.2.2).

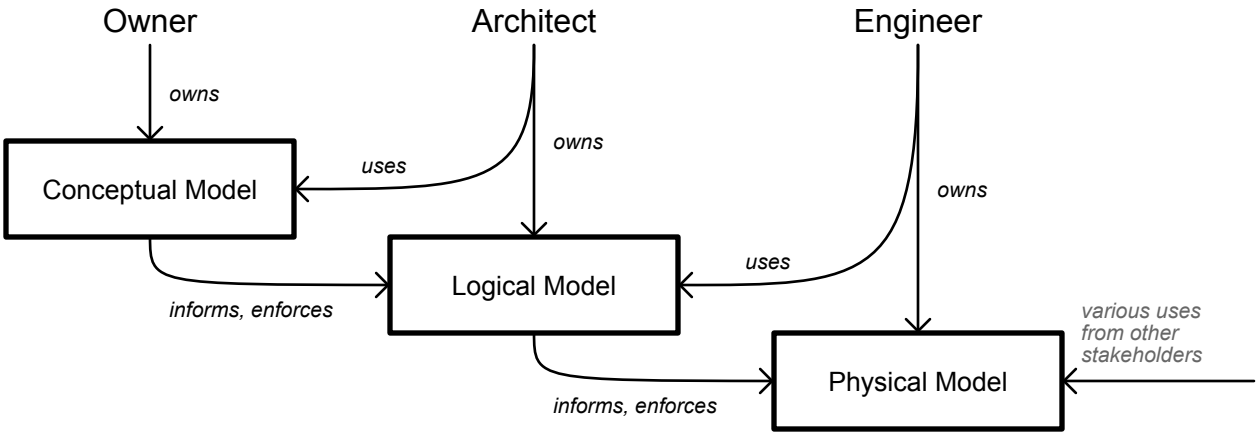

Figure 3. **Overview of stakeholder groups**. The models of the Software Product Model consists of three stakeholder-assigned models with dependencies between them.

2.1.1. Primary

A software product has three primary, orthogonal stakeholder groups that directly resemble the stakeholders discussed in Zachman's article (Zachman, 2000) about software models. However, I use the term architect instead of designer, and engineer instead of builder to better align with the concept of maps (Section 3.2).

Owner. Owners provide the idea or concept of the software product according to some real-world problem that needs to be solved. They produce design-ready information.

Software Architect. Architects design the behavior and structure of the software product according to the owner's concept. They produce engineering-ready information

Software Engineer. Engineers specify the *materials* needed to satisfy the architect's design. They produce highly-engineered, delivery-ready information

There is an implied dependency between stakeholder groups. The architect depends on the owner for the concept, and the engineer depends on the architect for the design. Consequently, the stakeholder group dependencies also imply informational dependencies.

2.1.2. Other

The software product can also have other stakeholder groups where their interests are not directed at the various functions of documentation but rather at its completed form. Secondary stakeholder groups are consumers of the final software product model. In implementation, these may be the development team consisting of developers, testers, planners and managers who uses the documentation to build, test, schedule and staff for the software product. In governance, stakeholder groups may be the governance board consisting of enterprise architects, board members, or facilitators who uses the final document to assess value, risk and alignment to standards. Finally, in planning, stakeholder groups may be an enterprise architecture practice consisting of organization leaders, enterprise architects and program managers who uses the documentation to rationalize capabilities and create multi-year plans. Implementation, governance and planning are the focal delivery functions covered in the software engineering extensions in Section 7.

3. Ontology

The Software Product Model consists of an ontology for software products. It defines a software product according to its models and model relationships. Given that there are three orthogonal stakeholder groups in a software product, it is logical to conclude that there must also be three models where each model is assigned to each stakeholder group. And because of the dependency between stakeholder groups, then there must also be two maps that relate one model to the other. Figure 4 provides an overview of the models and maps.

3.1. Models

The ontology contains three models called the Conceptual Model, Logical Model and Physical Model. The models are named according to Zachman (2000) and conveys the intent behind each model. The conceptual, logical and physical models aggregate and organize information to represent the owner's idea, the architect's design and the engineer's specifications, respectively.

3.1.1. Conceptual Model

The Conceptual Model is the first model in the ontology and it holds the idea or concept of the software product. It has two primary audiences: the owner that owns it and the architect that uses it. From the owner perspective, the conceptual model rationalizes the problem being solved. It describes a technology-agnostic solution to the problem that includes the activities and processes that need to produce the desired outcomes. Owners are responsible for the conceptual model.

Architects are users of the conceptual model. From their perspective, the conceptual model provides activities that describe some unit of behavior relative to the software product. It describes an activity as an action being performed by an actor. The model presents a collaborative opportunity between owner and architect that provides clarity to the activities, where necessary. It identifies which activities need digital enablement or automation. Digitally-enabled activities are described as features in the logical model.

Key Models. The conceptual model consists of sub-models that represent types of information that are important in describing a conceptual solution or idea.

Activity. An Activity models an action that is performed by an Actor in the context of the software product.

Action. An Action is a label given to the execution of an activity. It is performed by an Actor.

Actor. An Actor models a self-initiated entity that performs an Action on an Activity. It can represent the primary or target consumer of the software product. Actors, while historically representative of people, describe various self-activating entities such as people or roles, software, sensors or time.

3.1.2. Logical Model

The Logical Model is the intermediate model in the ontology and it provides the behavioral and structural design of the software product. It has two audiences: the architect who owns it and the engineer that uses it. From the architect perspective, the logical model defines the software product by its behavior and structure. It describes behavior at a product level called features and at a component level called capabilities. It also describes component structure from the composition of capabilities (Section 4.3). Architects are responsible for the logical model.

Engineers are the users of the logical model. From their perspective, the logical model provides capabilities and components. Capabilities are units of software behavior that require methods of interaction. Interfaces are interaction methods assigned to capabilities in the physical model. Components are abstract representations of software that require specification. Specialized components, software, are also described in the physical model.

Key Models. The logical model consists of sub-models that represent types of information that are important in describing behavior and structure.

Feature. A Feature models a single behavior of the software product in the context of an activity. It is essentially the digitally-enabled version of an activity.

Capability. A Capability models a unit of behavior of a software component. It represents the potential work that software and the underlying machine can perform when invoked.

Component. A Component models a container and server of a capability. It abstractly represents a parcel of software.

The logical model includes basic concepts of features, capabilities and components which are captured in the metamodel (Section 4). It also includes concepts that describe quality, benchmarks, tiers and distribution as metamodel extensions for architecture (Section 6).

3.1.3. Physical Model

The Physical Model is the final model in the ontology and it details software specifications of the software product. It has at least one audience: the engineer that owns it. Other audiences for the physical model can vary but may include delivery-oriented stakeholder groups such as development teams for implementation, board members for governance and enterprise architects for planning. From the engineer perspective, the physical model provides specialized components and interfaces assigned to capabilities. The combination of structure (from the logical model), specialized components and the communication protocols from interfaces is an engineering plan called a blueprint. A blueprint is the physical model in its final form; a delivery-ready model. The engineer is responsible for the physical model.

From a delivery-stakeholder perspective, the physical model provides a variety of information about software. It describes software in terms of its specifications, providers and system. Software specifications describe build and runtime information and providers describe platforms and infrastructure for software. And, software systems describe dependencies between software including their methods of interaction. Delivery-oriented stakeholder groups are users of the physical model.

Key Models. The physical model consists of sub-models that represent types of information that are important in delivering the software product.

Software Component. A Software Component models software of various types. It represents software that is distributable and invocable. Examples of software components are scripts, libraries, executables or container images.

Provider. A Provider models a type of software that enables a software component by invocation or similar methods. It is the *platform* that the software component runs on. Providers can be hierarchically be enabled by other providers.

Interface. An Interface models the method of interaction with a capability. It represents the rules for interaction including the transfer of information or invocation. Examples of interfaces are user interfaces, APIs and network protocols such as HTTP.

The physical model includes basic concepts of software components, providers and interfaces which are captured in the metamodel (Section 4). It also includes practical concepts of teams, builds captured as metamodel extensions (Section 7).

3.2. Maps

The dependency between stakeholder groups further defines a similar dependency between their assigned models. Stakeholder groups capture their information into a model and other stakeholder groups use such model as basis for theirs. Therefore, the logical model is dependent on the conceptual model, and the physical model is dependent on the logical model. A mapping is used to inform the next model and enforce coherence.

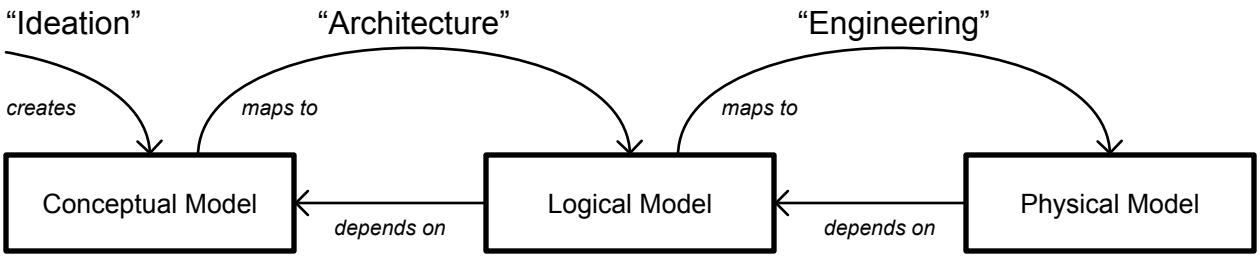

Figure 4. **Overview of maps**. The models are arranged in a chain according to their dependencies with named maps between them. Ideation is a pseudo-map that initially creates the conceptual model.

The ontology contains two, model-to-model maps called Architecture and Engineering. The Architecture-named map relates the conceptual model to the logical model, and the Engineering-named map relates the logical model to the physical model.

They are named according to the stakeholder group responsible for its tasks. An initial, pseudo-map called Ideation is a creational step that does not have a preceding model.

3.2.1. Ideation

Ideation is not a model-to-model map but is nonetheless a creational process similar to the architecture and engineering maps. It is the initial step that creates the conceptual model. Although it is not explicitly preceded by another model from which it maps from, it is a human-to-model map. Concepts or ideas originating from the human mind are more or less mapped to the conceptual model during the ideation process. It is a pseudo-map that translates those ideas from the owner into the conceptual model. Practically, many ideas are pre-recorded externally from the software product model. Requirements can exist in artifacts such as documents and spreadsheets or within issue-tracking tools. So, ideation may also refer to the oft translation of other sources into the conceptual model. The business process extension related to this pseudo-map is covered in Section 5.

3.2.2. Architecture

The first map is called Architecture and it relates the conceptual model to the logical model. Because the logical model is owned by the architect, so too are its tasks associated with this map; thus named Architecture. Architecture is an essential map that provides the logical model with information necessary to output software structure according to conceptual requirements.

Architecture maps the activities from the conceptual model to software features in the logical model. Not every activity requires a corresponding feature because they may be temporarily or permanently manual. If an activity is indeed mapped to a feature, the activity is deemed software-enabled. A feature is a unit of behavior of the software product by referring to an isolated structure of components. After engineering, the structure becomes a software systems, therefore, at its completion, an activity that is mapped to a feature is essentially represented by a bespoke software system that behaves according to the activity.

While mapping an activity to a feature leads to a software structure, the intended behavior of the feature is not the only factor for the structure. Quality is also a

determinant for structure that defines a minimum performance metric for the activity and feature. Unlike a feature where the structure itself is direct proof of its existence, the existence of quality is proven by quantitative or qualitative aspects about the structure. That, to achieve a given quality, the structure must be modified while maintaining the intended desired behavior. The metamodel is extended to include quality and benchmarks (Section 6.1).

3.2.3. Engineering

The second and final map is called Engineering and it relates the logical model to the physical model. Similarly to Architecture, because the physical model is owned by the engineer, so too are its tasks. Engineering is also an important map because it provides capabilities, components and structure necessary for the engineering tasks.

Engineering maps each component in the logical model to a fully-specified software component in the physical model. A software component is a type of distributable software that can be programmed or configured to satisfy the intended behavior of the capability. Thus, a software component is also a container and servicer of capabilities.

Each software component is coupled to a provider that enables or activates it during engineering. A software component and provider share a common integration method that allow for activation. Providers are hierarchical where one provider can be the provider to another.

Engineering also maps capabilities to one or more interfaces. An interface describes the method of interaction with the capability. Some interfaces describe a user interface that further describes some user-to-machine interaction, while other interfaces describe a protocol that define connectivity and data structure rules.

At a basic level, engineering is the specialization of components (a.k.a. software), providers and interfaces. The component specifications required are determined by the intended usage of the documentation which can vary from one organization to another. In delivery, implementation may require build information such as language and runtime, or teaming information such people and their roles. For strategic planning, which employ various reporting methods, may require lifecycle status (e.g., invest, maintain, retire) or application portfolio information.

4. Metamodel

The Software Product Model consists of a metamodel for software products. It is an expression of the ontology and of its models in a schematic form. It contains elemental forms of the concepts from the conceptual, logical and physical models.

4.1. Elements

The metamodel consists of related elements that represent concepts from the conceptual, logical and physical models described by the ontology (Section 3).

Activity. The Activity element describes an action performed in the context of the software product. It is performed by an Actor, optionally produces a Result and can be sequenced with other activities as a Process. The activity represents a conceptual model.

Actor. The Actor element describes a real-world concept or object. It is a self-initiated element that is a catalyst of behavior. Historically, actors represented people evidenced by the human stick figure shape chosen by UML (x, x). Beyond people, actors can also be software, sensors, time or Events. An actor can be represented by a Capability. The actor represents a conceptual model.

Capability. The Capability is a fundamental element in the metamodel that describes a unit of behavior in software. It accepts input and produces output of a set of named Types and optionally persist its output in its serving Component. The output of a capability, or the capability itself, can be used as input to another capability; a process called composition (Section 4.3). A capability is made available by a set of Interfaces. A set of capabilities, usually in composition, are captured as a Feature. A capability can represent an Actor. The capability represents a logical model.

Component. The Component element describes a type of distributable software such as scripts, libraries, executables or container images. It serves a set of Capabilities. The component represents a logical model.

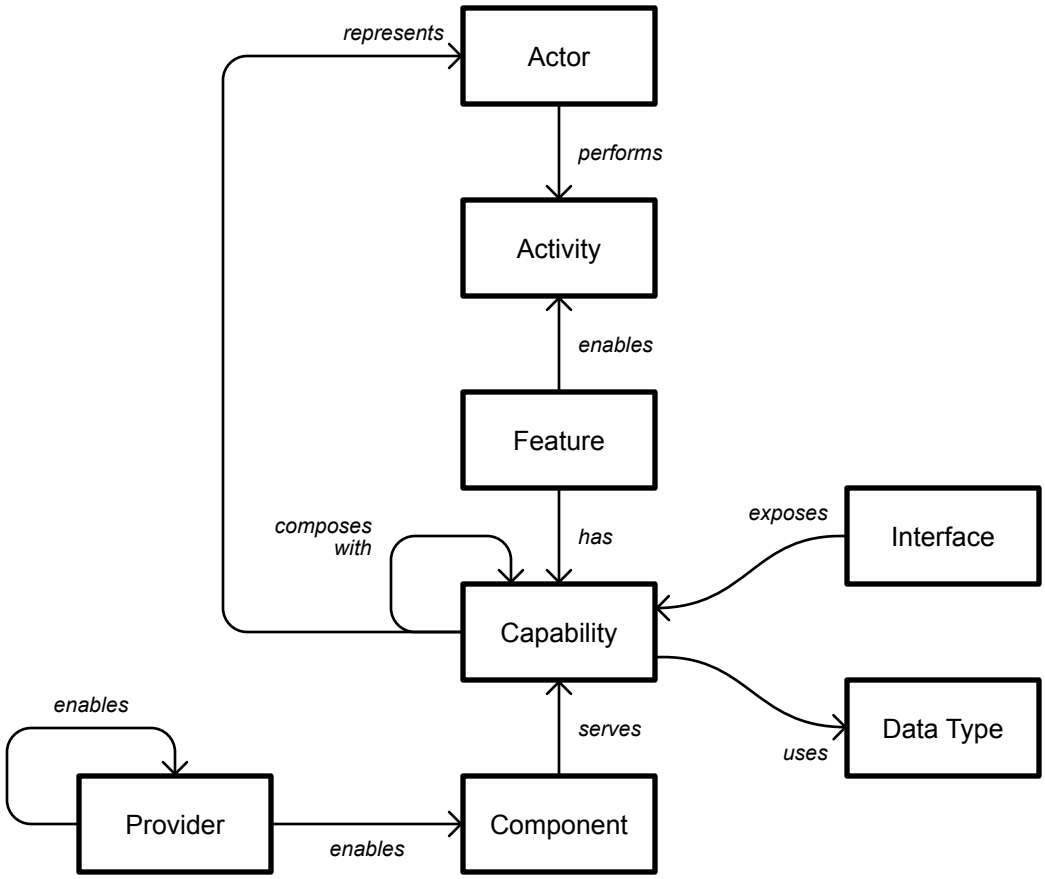

Figure 5. **Overview of elements**. The metamodel consists of related elements that represent concepts from the span of models.

Feature. The Feature element is the digital enablement of an Activity and it consists of a set of capabilities. It describes a unit of software product behavior which it aggregates from its capabilities. The feature represents a logical model.

Interface. The Interface element describes a method of interaction with a Capability. An interface can expose a capability in to a variety of mediums including peripherals, screens or software. Software interfaces are governed by a protocol that provides connectivity and data structure rules. The interface represents a physical model.

Provider. The Provider element describes a type of (software) Component that enables another a component. Because it itself is also a component, then it is also

enabled by another provider. Thus, providers are naturally hierarchical. The provider represents a physical model.

Data Type. The Data Type element refers to a class of information and describes data that is used as input or output of a Capability. A type can be defined as scalar or complex where a complex type consists of typed properties. The data type represents a logical model.

4.2. Mapping

The ontology defines two maps, conceptual-to-logical (Architecture) and logical-to-physical (Engineering), which have to be accounted for in the metamodel. The transition from conceptual to logical elements is creational, and from logical to physical is both creational and attributional. Specifically, to transition from conceptual to logical requires the creation of a Feature, and from logical to physical requires the creation of Interfaces and attribution to the Component.

4.2.1. Activity and Feature

An activity is digitally-enabled when it is related to a feature. An activity in the conceptual model is the relationship between an actor and the activity. And, a feature in the logical model is a set of capabilities. Mapping is accomplished by representing each actor and activity as a capability in the feature.

In this mapping, both actor and activity become related capabilities. The actor-based-capability has a dependency to the activity-based-capability and forms an initial but loose composition (Section 4.3). The root capability is one that was mapped from the actor, and it is valid because actors are self-initiated. Inversely, a composition of capabilities where the root capability is not from an actor is a composition that is forever inert, and thus, provides little or no value. Instead, when an actor activates, it signals the activation of the composition.

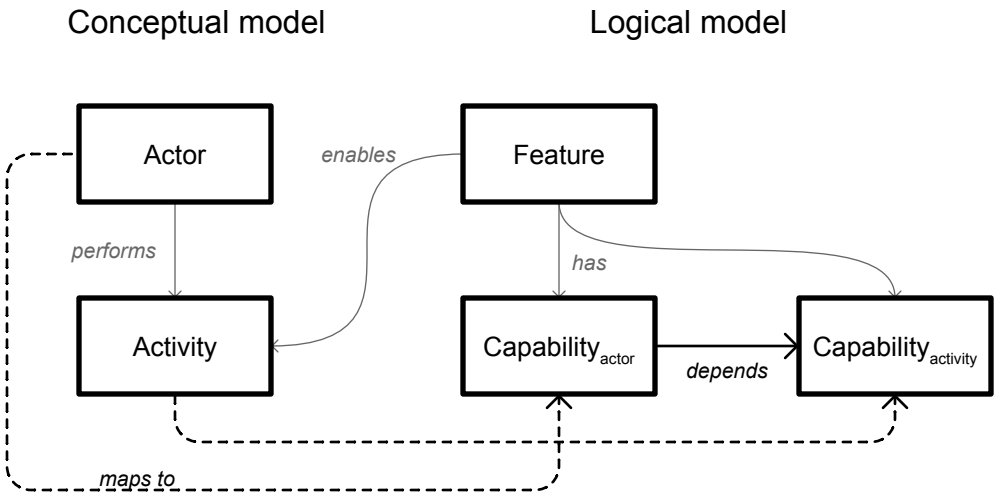

Figure 7. **Creation of a feature**. A feature is created from an activity by creating two capabilities that represent the actor and activity.

4.2.2. Component Specification

A component in the logical model is an abstraction of software where details are not yet revealed or known, whereas, the same component in the physical model is a specialized component with software specifications. Specification is achieved by component attribution, the addition of software-related properties to the component. The revelation of software properties (attributes) to the component enables further specification of its provider.

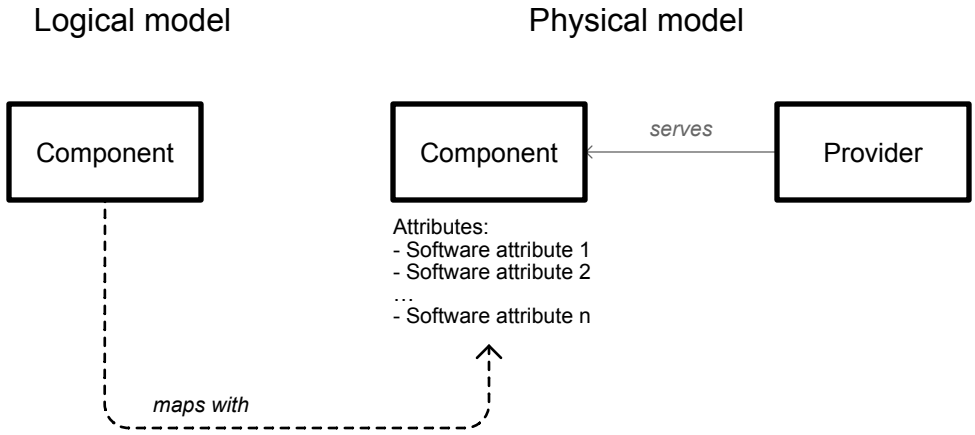

Figure 6. **Specialization of a component**. Component attribution or specification accepts attributes to specify software-related properties. A specialized component is also called software.

4.2.3. Interface Assignment

Capabilities in the logical model are related to a set of interfaces in the physical model. An interface describes a method of interaction with the capability. Two common types of interfaces are the user interface and the communication protocol. A user interface is a type of interface that enables for a person-to-software interaction. A protocol is an abstraction of type of interface that provides software-to-software communication. Familiar examples of protocol subtypes are TCP, UDP, HTTP and Kafka.

The assignment of an interface to a capability has implications to the component assigned to that capability. That by assigning an interface to a capability, the component bears the responsibility of exposing the interface type. Additionally, a component may explicitly prescribe a set of available interfaces which leads to the ability for interface verification. That, if the difference between the set of available of interfaces and the set of capability interfaces is the empty set, then the capability is valid (and supportable).

4.3. Composition

Software behavior is divisible into smaller behaviors and performed for various practical reasons. It is not sufficient for behavior to *just* be decomposable because as when its broken into smaller parts they must be put back together. Therefore, behavior must also be composable. That, decomposition and composition is an essential pairing. Several reasons may drive the practice of decomposition and composition.

Complexity. The complexity of software behavior may warrant its decomposition into smaller, simpler behaviors. That, when a [behavior] grows it becomes unwieldy (Evans, 2003).

Utility. A sub-behavior may be identified as being useful to other behaviors that warrants its isolation to become reusable.

Organization. The teams of people collectively chartered to deliver the software product may warrant its inherent decomposition to reflect their communication structures, as a reinterpretation of Conway's Law (Conway, 1968).

The Capability is the element in the metamodel that describes a unit of (software) behavior and it is composable. The composition of capabilities uses a combination of

two methods, functional composition and statefulness, that together achieves the representation of behavior composition in software. Because capabilities represent a unit of behavior, then their composition essentially represent an aggregate of behaviors. The aggregation of behavior is a unit of software product's behavior which is represented by a Feature.

4.3.1. Functional

The functional aspect of a capability draws from its functional composition ability and its emulation of a function. In mathematics, function composition is the process of applying one function to another to create a new function (Judson, 2012). This is popularly expressed with the following example:

> Given functions f and g
> Then $g \circ f \rightarrow h$
> where h is a new function that applies f to g

Figure 8. **Composition with abstraction**. The composition of two functions is when one function is applied to the other to create a new function. The constituent functions are abstracted (hidden) away.

The definition of creating new a function from composed functions is important to note but is, however, less useful in the context of capability composition because of its destructive nature. By creating a new function, it abstracts (hides) the constituent functions and if the concept is applied to capabilities, then it would also hide the capabilities. The preservation of capabilities is important for deriving structures (Section 4.4).

Functional composition can be regarded in a non-destructive way. Alternatively, function composition can be defined as the usage of the range of the first function as the domain to the second function. That is, the output of one function can be used as the input to another (Khan Academy). This definition is useful in two ways. First, it preserves the functions and secondly, it formalizes the use of variables.

Given functions $f(x) \rightarrow y$ and $g(y) \rightarrow z$
Then $g \circ f = g(f(x)) \rightarrow z$

Figure 10. **Overview of function-preserved composition**. The composition of two functions can be defined as the output (variables) of one function is utilized as the input to another..

The concept of function composition is applied to capabilities where a capability is defined like a function where it can receive input and produce an output. A capability is attributed with input and output properties. However, mathematical functions differ from software capabilities in terms of (data) types and state.

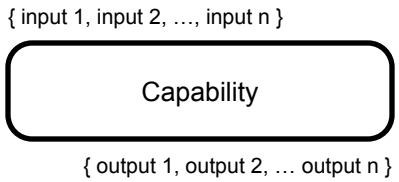

Figure 9. **Notation of a capability**. A capability can be visually depicted with a rounded rectangle, input oriented top-left and output oriented bottom-right.

4.3.2. Data Type

Unlike mathematical functions where function range and domain are homogenous, software capabilities often accept and produce data of varying types. Therefore, type validation, or at least type awareness, is necessary with capability composition where it may be otherwise implicit with functional composition in mathematics.

In the metamodel, there is less concern for instances of data because those values are irrelevant to design. Instead, there is greater interest for types of data, referred to as Data Type in the metamodel. Type validation is the foundational concept that guarantees structural strength.

A capability is attributed with input and output properties where each property is a set of named types. A named type is similar to a variable but its ability to carry an instance of that type is ignored.

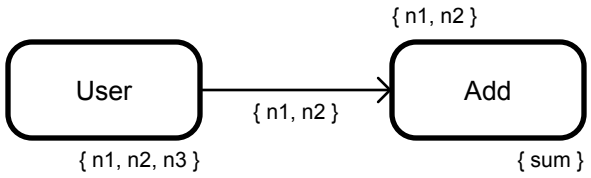

Figure 12. **Composition of a capability**. A capability can be composed with another capability, its composition is type- and state-aware.

4.3.3. State

The stateful aspect of a capability draws from its emulation of a software program. Programs have the ability to persist state to its environment and beyond its execution. Unlike mathematical functions where state is implicit in the way it is input or output, a capability differs in how it retrieves input or produces output.

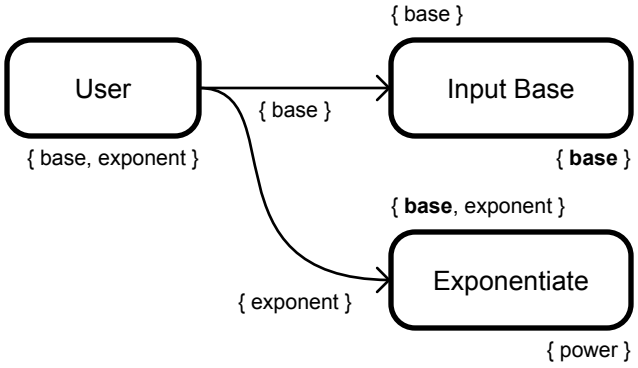

Figure 11. **Example of compositional state**. An example of an exponentiation feature that initially persists the initial input called “base” and is later retrieved, from state, in the Exponentiate capability.

Unlike functions, a capability has an environment provided by the Component that serves it. This environment augments how input is retrieved or output is produced because of the environment’s ability to persist state. When a capability executes, it retrieves its input from two locations, the parent capability or its component.

Capability State. A capability can retrieve its input from the implicit state created by the capability it may have been composed with. This is similar to the implicit state of function composition.

Component State. A capability can retrieve its input from any state provided by its component.

Similarly, a capability is attributed with a property that indicates whether its output is persisted to its component.

4.4. Structure

The goal of the logical model in the ontology is to produce a software structure, therefore, the logical elements of the metamodel must also satisfy that goal. However, structures are not directly defined as they are in traditional or current documentation practices. Instead structures are derived from capabilities. A structure is a graph of component and their dependencies.

4.4.1. Capability Graph

The derivation of structure utilizes the composition of capabilities. In composition, the output of one capability is used as the input to another. The capability whose output is used as input is called the source (capability) while the related capability utilizing the output is called the receiver (capability). A dependency type of relationship is implied between source and receiver. A directed graph can be constructed using a pair of a set of capabilities and a set of dependencies, where each dependency is a pair of distinct capabilities. The definition is the same used in graph theory (Bang-Jensen et al., 2007).

$G = (C, D)$	where: G is the graph of capabilities, C is the set of capabilities, D is the ordered set of dependencies, where each element is a distinct pair of capabilities

Figure 13. **Definition of a capability graph**. The graph of capabilities expressed in a mathematical notation.

4.4.2. Component Graph

The graph of capabilities is used to derive a graph of components which is the required graph to produce a software structure. Component graphs are different than

the capability graph they are derived from because multiple capability dependency edges will reduce to a single component dependency edge. And unlike capability graphs where the edges are comprised of distinct capability pairs, component graph edges do not have to be distinct. Therefore, the component graph is a directed graph that allows for loops.

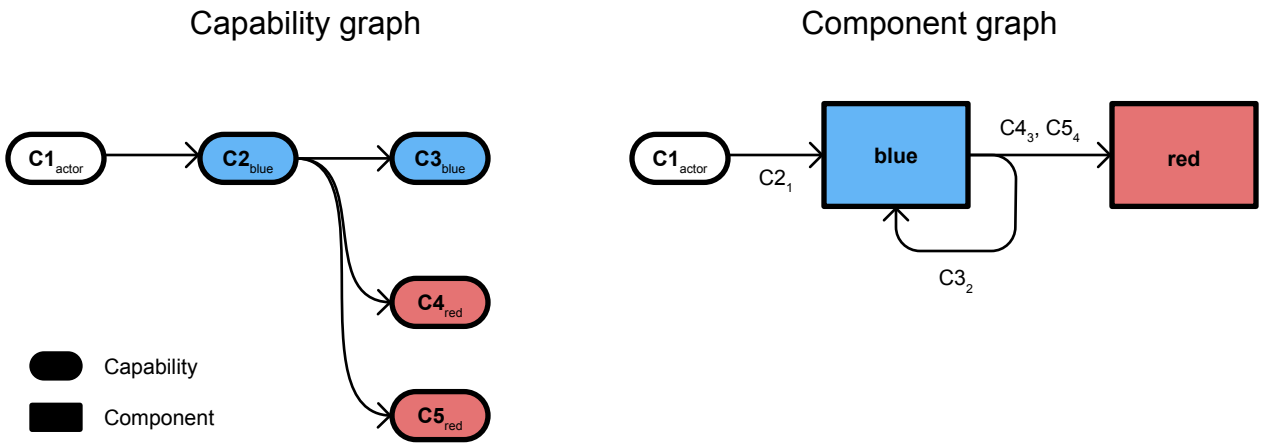

Figure 14. **Derivation of a component graph**. A component graph is a digraph with loops allowed. It is derived from a capability graph.

4.4.3. Scoping

Structures are inherently scoped at a feature level because the basis of structure are the compositions described by the feature. Feature-scoped structures are helpful in understanding participating capabilities or components for the given feature because it removes the *noise* (non-participating components). This is generally helpful when collaborating with stakeholders at this level.

A structure scoped at higher levels is also helpful because it provides the audience with a broader view. It is beneficial to view the structure of a business process or other related features because it provides a precise focus on relevant behavioral or structural elements. It is also useful to look at the structure at the software product level because of its complete view of all elements. Governance, for example, is a beneficiary of a software-product-scoped structure.

To scope a structure, a binary, non-disjoint union operation is performed. It is an operation described as part of graph theory (Nguyen et al., 2023) that adds two graphs together. By taking the set of given features, determined by the desired scope and their associated capability graphs, then performing a augmented binary union of the graphs, a new graph is created that consists of all vertices and edges.

Augmented Binary Union

Edge fidelity cannot be compromised during a union and must preserve their lineage. In the context of scoping structures, the feature is the source for each capability graph, and therefore must be preserved. The binary union of two graphs must be augmented to produce a new graph that provides lineage of each edge. Each edge in the union must provide a set of sources (features) that they originated from.

The importance of edge fidelity is a result from the importance of descriptive edges, that is, the ability to rationalize the existence of an edge has value. In features, the edges represent the compositional dependency between two capabilities. The descriptive form of the edge is by its inputs. With the union of features, the descriptive form of the edge adds the source of the edge; the feature in this context.

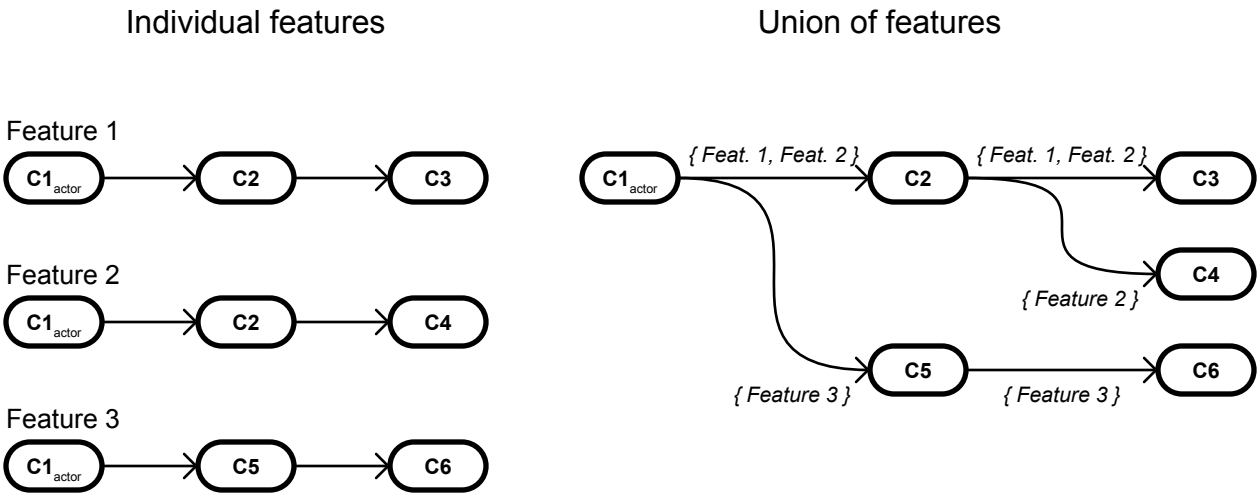

Figure 15. **Overview of scoping**. A the graph of capabilities, in this example, from three features, form a scope and union to form a larger composition. The edges in the union is are sourced edges which describe their source features.

5. Ideation Extension

The metamodel is extended to support definitions during ideation that generally center around describing business processes. In the base metamodel, an activity defines a single action taken by an actor. In the updated metamodel, activities can be connected to form a process. It also introduces the (business) event as a type of actor.

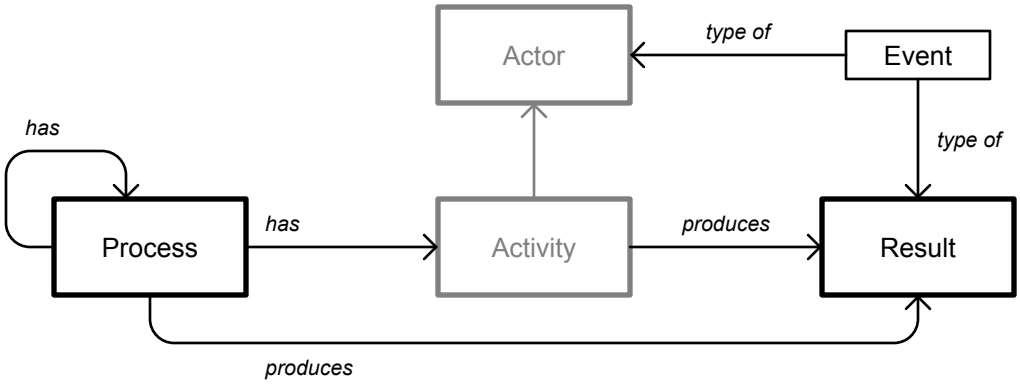

Figure 16. **Summary of ideation extension**. The metamodel is extended to enable the definition of business processes and events.

5.1. Process

The Process element describes a complex business- or product-oriented *activity*. At a lower order, it consists of related activities and generally in a sequence. Higher-order processes can consists of a heterogenous set of activities and other processes, thus, a process can be hierarchical.

5.2. Result

The Result element describes information (state) produced by a completed activity or process. An activity or process can yield zero or more results. A result represents either the state itself or an event that occurred in conjunction with the state. As a stateful element, a result can hold named and typed data as attributes.

5.2.1. Event

The Event element is a type of Result and it describes a named occurrence relative to

some information. An event is also a type of actor which enables it to be a performer of an action.

5.3. Sequence

An activity or process can be sequenced when the source element produces a result. However, there are constraints based on the result. When the result is an event, it can only be sequenced with an activity with the same event. When the result is state, it can only be sequenced with an activity with a non-event actor.

A sequence represents a transfer of state. When the result is a state type, all of its data (attributes) are transferred to the actor of the target activity. When the result is an event, the state is implicitly transferred because the event is the actor of the target activity. State transfer in the conceptual model is not a technical occurrence and cannot be logically validated. Instead, it must be rationalized by the owner.

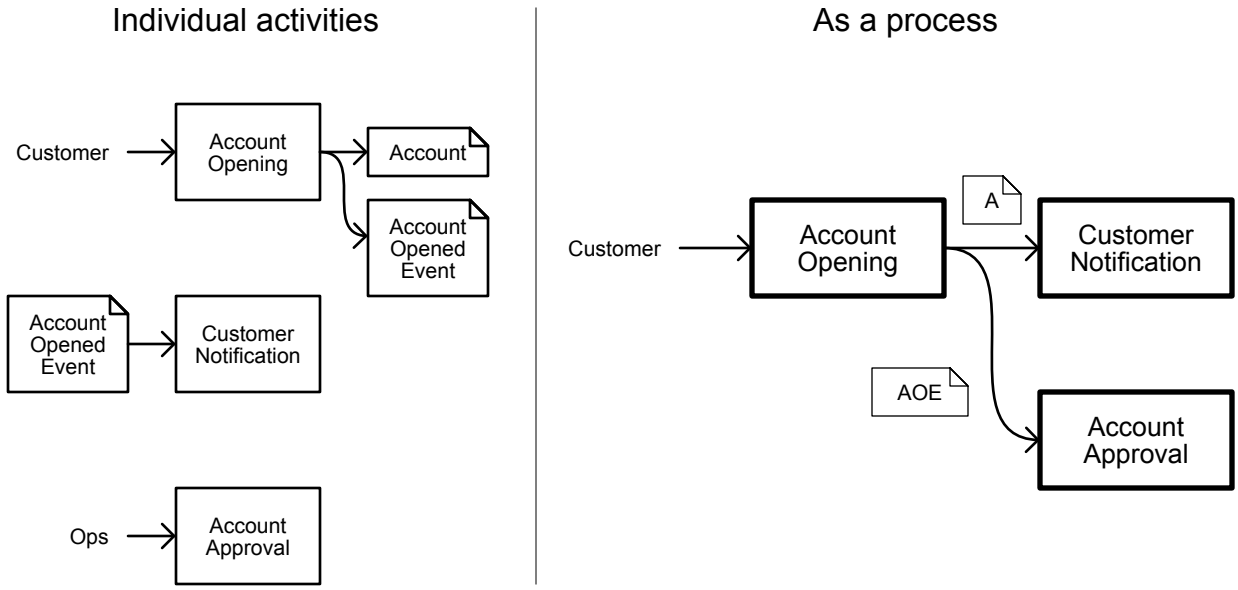

Figure 17. **Example of state transfer**. Activities can be sequenced into a process where state can be transferred as state or an event to downstream activities.

5.4. BPMN

The extension for ideation is intended to represent currently accepted techniques of modeling a business at a conceptual level. The BPMN (Business Process Model and

Notation) standard is a graphical approach to modeling a business using various shapes and connectors (Object Management Group, 2013). While the ideation metamodel is a structural definition of business, it can be further extended for alignment to BPMN to enable similar visualization.

BPMN enables conditional sequencing of activities using the Gateway shape. Conceptual elements that seem to represent logic, such as a gateway, does not materially affect the technical models. Such elements aid in conveying a business process but does not translate to the logical model. Only activities and actors (including events) are mapped to the logical model. While the ideation extension omits additional elements that may align to BPMN, it does not prevent such additions to alignment.

6. Architecture Extension

The metamodel is extended to support the definition of quality and additional behavioral distributions. In the base metamodel, activities and features are solely focused on behavior sans quality. Additionally, structure is based on only one type of behavioral distribution with capability composition. The metamodel is updated to support the specification of quality and their benchmarks. It is also updated to support the definition of tiering and scaling, two behavioral distributions that optimize structural outcomes. The architecture extension adds a novel software modeling technique with dynamic elements and environment. An algorithmic element, such the benchmark, tier and distribution elements, is a type of dynamic element that can compute new elements or attributes. An environment provides context to the software product especially during execution.

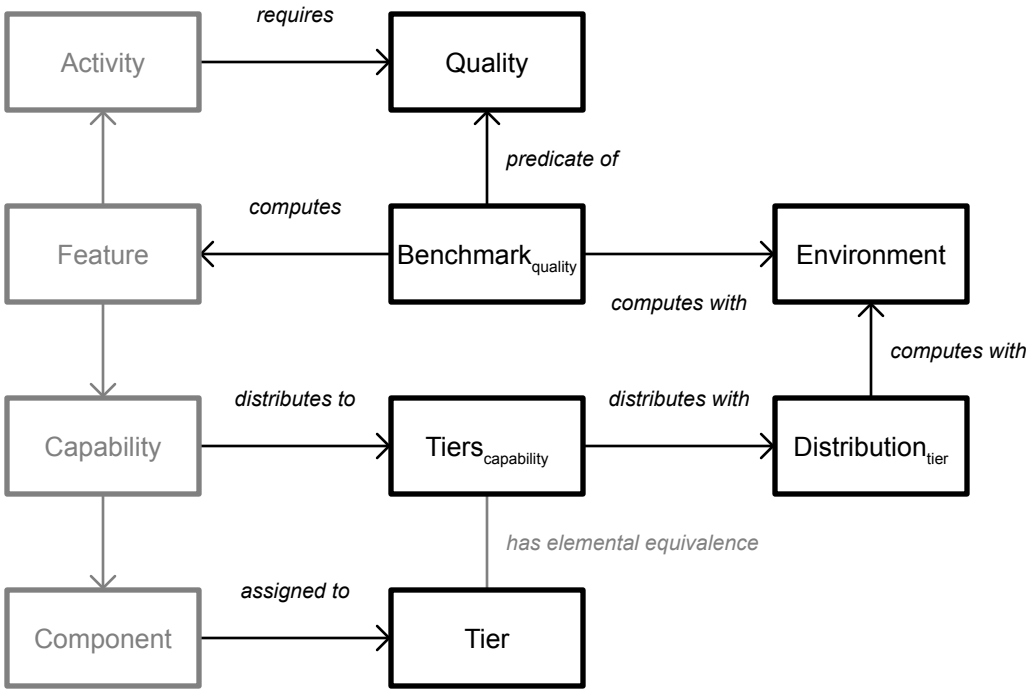

Figure 18. **Summary of architecture extensions**. The metamodel is extended to enable the definition of quality and behavioral distributions tiering and scaling. It also includes algorithmic elements for quality benchmarks, tiered and scaled distribution.

6.1. Quality

The Quality element describes a performance requirement that its associated activity is expected to minimally perform. An activity can have zero or more qualities. It is a conceptual or high-level description of the quality. With the updated metamodel, activities gain quality requirements along with behavioral requirements. Structure plays an important role in satisfying both behavioral and quality requirements.

A quality element consists of a conceptual description of the quality and a set of Benchmarks. Each benchmark is computed to indicate whether their quality metric is satisfied.

6.2. Benchmark

The Benchmark element is an algorithmic element that computes a quality metric. It has a dual relationship. It is associated with a quality as one of its metrics, and it computes the feature that it is associated with the activity that is associated with the quality that it is associated with.

A benchmark is a predicate. It is an algorithm that produces a binary, boolean value that indicates whether the quality metric it represents is satisfied. During computation, the benchmark algorithm uses two inputs, an environment and a feature.

The quality of a feature can be assessed quantitatively or qualitatively. Consequently, benchmark algorithms can be classified similarly. A class of benchmark algorithms can inspect, transform or aggregate attributes from any element in the feature. For example, if the quality metric is for latency, the algorithm may return a sum of all capability's interface latency, or if the quality metric is for availability, it may return the product of all component's availability. Another class of benchmark algorithms can inspect the *quality of the structure* of the feature by asserting facts about elements. For example, if the quality metric is security, it may assert the presence of a token and its origin.

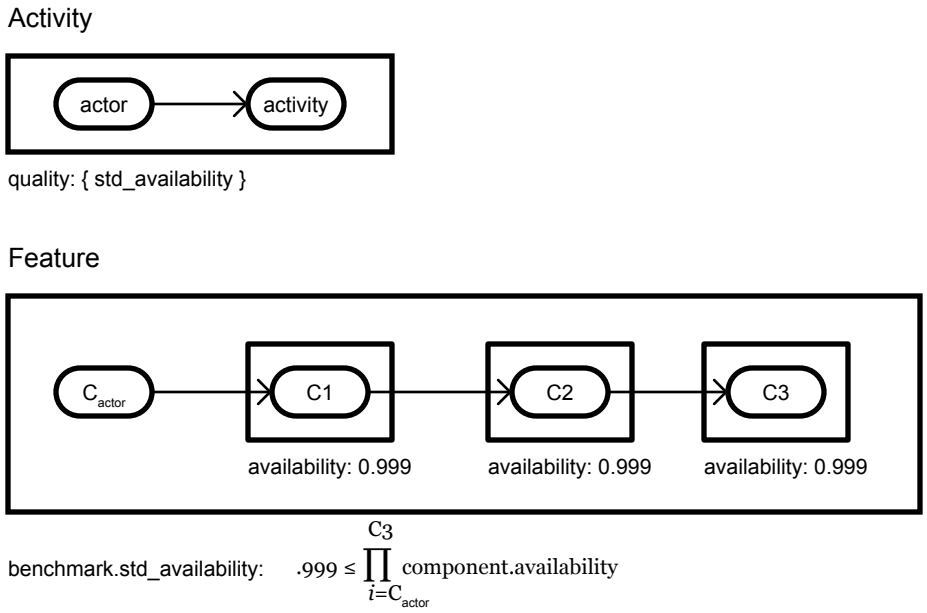

Figure 19. **Example of a benchmark**. A given activity and feature with a standard-availability requirement. The quality requirement is assigned a benchmark that computes the product of each capability's component's availability. In this example, the benchmark would compute false because a 99.7% availability is less than 99.%.

6.3. Environment

The Environment element describes a software product context. It consists of attributes that generalize the execution or operational trait of the software product. An environment is a novel element that supports the algorithmic benchmark, tier set and distribution elements. It provides information that is necessary for quality-related computations. Generally, an empty environment indicates normal operation whereas its attributes may indicate abnormalities. For example, it may indicate hight network latencies or a disaster status.

6.4. Tier

The Tier element describes a computing space contained within a process boundary. It is the computing space for a software component and its capabilities. Tiers are used for describing a component's computing space and the distribution of a capability. Computing spaces or process boundaries are necessary for information locality which is one approach to information sharing. In the updated metamodel, there are two types of

tiers: a single tier assigned to a component, or a tier set that is assigned to a component.

A single tier is assigned to a component to describe its computing space. In this context, a tier is a scalar-value attribute to a component. It is a logical fact about the component and a declaration of its logical location.

An ordered set of tiers is assigned to a capability to describe its distribution. The ordered set of tiers is a union of two other sets: an ordered set of proxy tiers and a single-element set of a target tier, in order. In this context, the tier set is an algorithmic element that replicates and composes the capability. A tier set is regarded as a feature-preserving, structure-altering algorithm and is used as an optimization technique in a documentation practice that avoids definition duplicity. The algorithm accepts a capability as a sole input.

Tier validation is necessary because it is possible for a mismatch to occur. Component are assigned to capabilities including expanded capabilities. Therefore, it is possible for the tier of a component to not match the tier of a capability-tier. A valid component assignment is where the capability and component tier match.

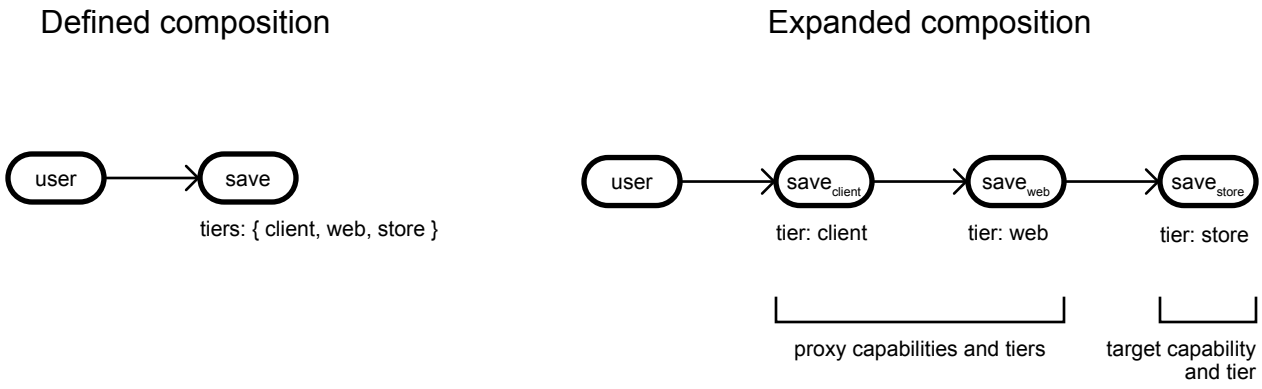

Figure 20. **Example of tiering**. A defined composition with a specified ordered set of tiers is expanded into their base capabilities and tiers.

6.5. Distribution

The Distribution element is an algorithmic element that describes a homogenous distribution of a capability-tier. It is used to provide the capability with extra computing resources. The additional resources enable the capability to meet scalability or reliability

quality requirements. The distribution element accepts a tiered capability and an environment to produce a set of ordered partitions where each partition represents a percentage of distribution. It is an algorithmic element that replicates the tiered capability per size of the distribution set; specifically, the original capability-tier is replaced with an expanded set. Each new capability gains a partition attribute.

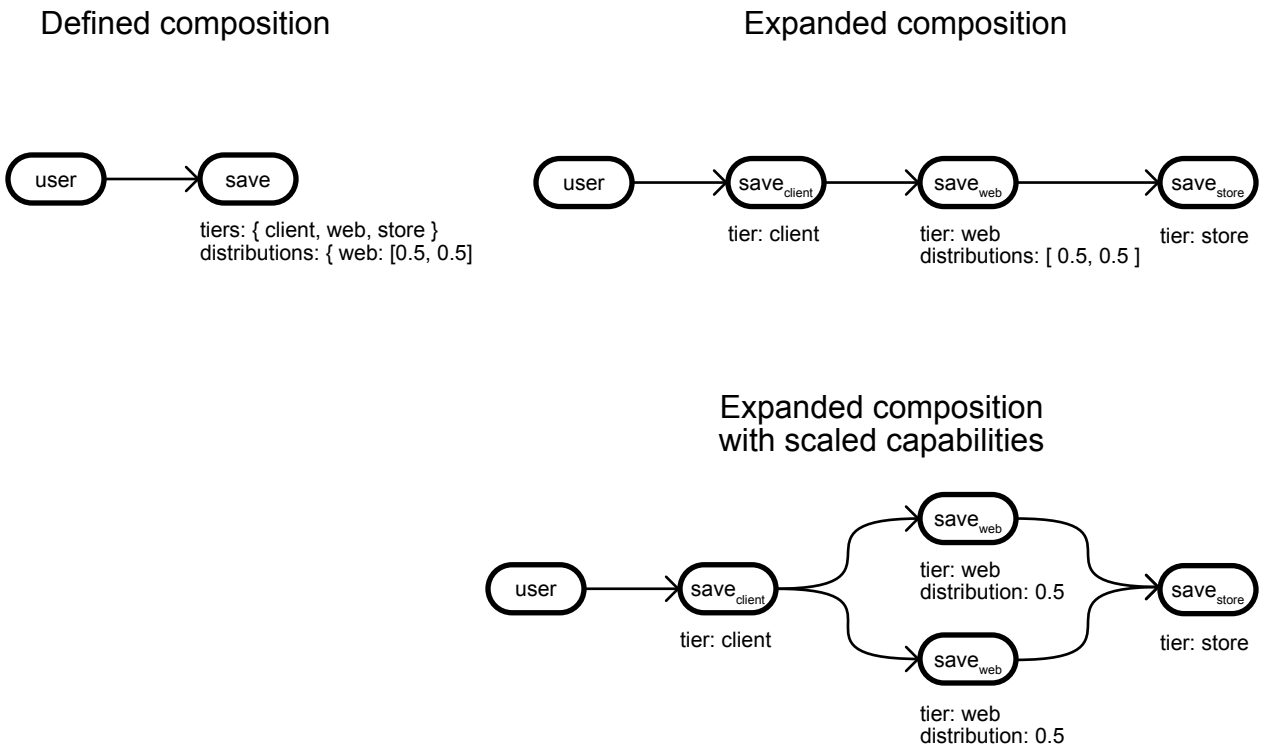

Figure 21. **Example of scaling**. A defined composition with a specified ordered set of tiers and a distribution for the web tier is expanded into their base capabilities and tiers. The final expansion includes replicas of the capability at the web tier.

7. Engineering Extension

The metamodel is extended to support the definitions of teams, versions and standards. The extension also enables post-engineering functions such as governance. In the base metamodel, elements are unaware of resources such as time or people. They do not have time-dimension definitions nor assigned stewards that oversee or make decisions about them. In the updated metamodel, software products become version-aware to support major, minor and optional changes. Also, the elements in the updated metamodel, such as the software product, feature and components, gain the ability to be assigned a person of a specific role from a specific team. The updated metamodel also gains the element, standard, that provides a standard model to technical models.

7.1. Teaming

Teaming includes Person, Role and Team elements that combine together to form a Steward element. A Team is a part of an Organization, and organizations can be hierarchically structured.

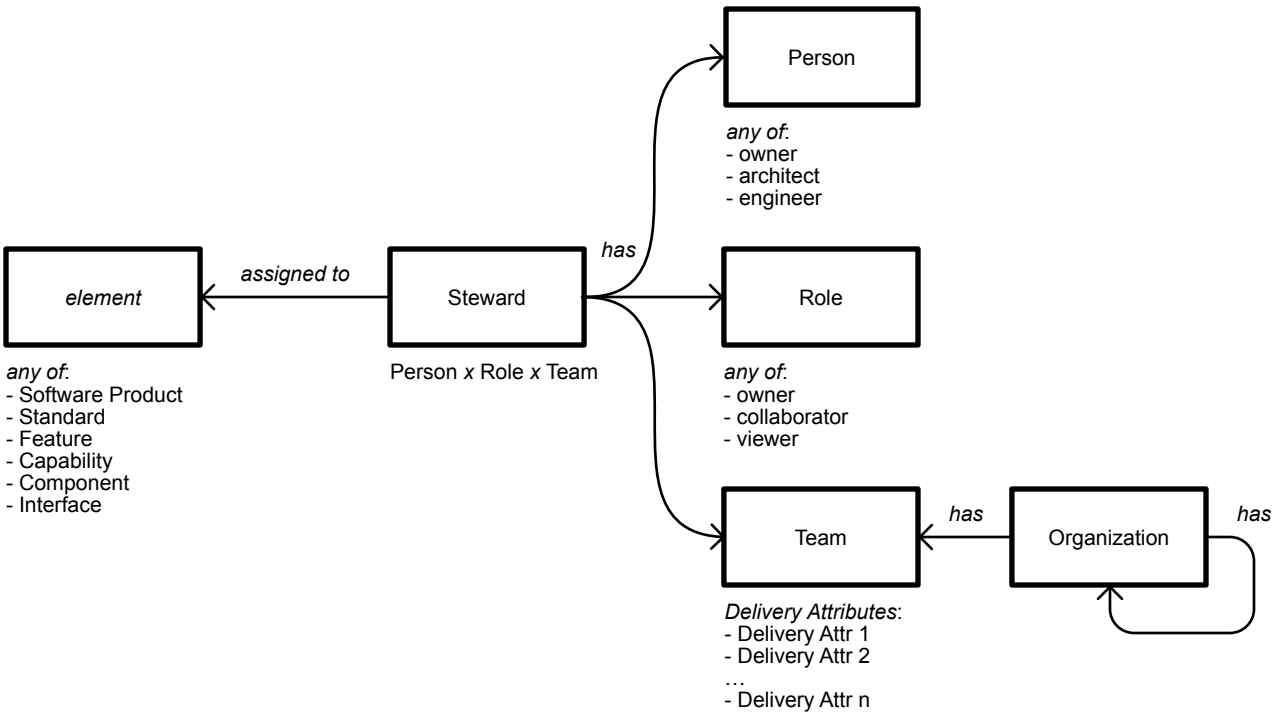

Figure 22. **Generalization of teaming**. The metamodel with steward assignment. A steward is the cross of a Person, Role and Team. Organizations are hierarchical.

7.1.1. Person

The Person element represents a real-world person that belongs to one or more stakeholder groups; usually an owner, architect or engineer. They have the ability to view, create, modify or remove elements according to their Role and the type of element they are assigned to. A person is an attribute on a Steward.

7.1.2. Role

The Role element represents an abstraction of allowable permissions where such permissions usually center around the ability to view, create, modify or remove elements. Generally, an owner can perform all permissions including deletion. Generally, a collaborator has the same permissions as the owner excluding destructive permissions such as deletion. And generally, a viewer only has the permission to view. A role is an attribute on a Steward.

7.1.3. Team

The Team element represents a group of people (Persons) with the same charter. The team element is an attribute on a Steward.

7.1.4. Steward

The Steward element represents a person that is assigned to an element with a specific role and from a specific team. The intersection of person, role and team is necessary because a person can have multiple roles and belong to multiple teams.

7.1.5. Organization

The Organization element describes a group of teams or organizations with the same charter. Organizations are hierarchical.

7.1.6. Delivery Attributes

A Team can be specified with delivery attributes that indicate resource utilization metrics for delivering an element. This paper omits specificity for these attributes due to team or organizational variance in how it measures resources. Teams or organizations can consider attributes such as story-points per element type or time-based durations. They can also consider dynamic attributes such as algorithmic ones that can perform

calculations given an element or complexity.

7.2. Versioning

From a metamodel perspective, versioning is achieved by inserting a Software Product Version element where the Software Product is. This is a drastic change because it moves any relationships between the Software Product and its elements to the Software Product Version. For brevity, the Software Product Version is aliased and referred to as Version.

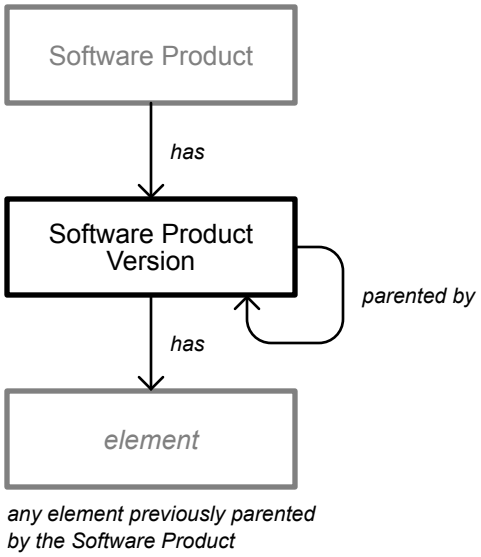

Figure 23. **Summary of versioning updates**. The metamodel is updated with a new Software Product Version element that captures a snapshot of the Software Product.

A Version, in effect, is a snapshot of the software product. It includes every element in the metamodel that spans from the conceptual model to the physical model. Versioning is a key function of a software engineering discipline because it provides an account and a plan for a software product. Versioning enables backwards-accounting by providing a historical set of versions including inception. Historical views on a software product may provide additional context to its current state. It also enables planning of future versions of the software product.

Versioning for future forms of the software product includes three types of versions.

The metamodel is updated with a version attribute at the software product level.

Strategic. A Strategic version is a desired future definition of the software product on an extended timeline. While a formal definition is not possible because of the differences in organizations, a strategic version can be described as a version where multiple tactical versions can precede it. Generally, a strategic version is a multi-year outlook of the software product.

Tactical. A Tactical version is a definition of the software product that immediately follows the current version. A tactical version is a definition that intends to conform to a strategic version. Generally, a tactical version is the result of an iterative delivery cycle such as a sprint in Scrum or predetermined, iterative release.

Optional. An Optional version is a definition of the software product that branches from a current or tactical version and precedes a tactical version. An optional version provides a choice for the next tactical version. In practice, an optional version is used when there are two or more choices.

The minimal schema for a software product version partially follows semantic versioning (Preston-Werner) by using the concept of major and minor monikers for strategic and tactical versioning. However, semantic versioning uses a linear versioning approach and does naturally support the branching aspect of optional versioning. For branching, a letter suffix is used.

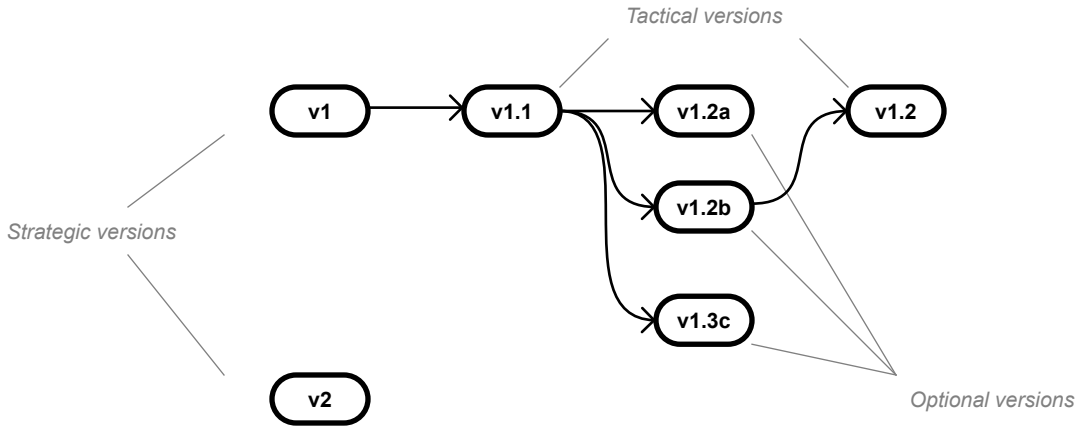

Figure 24. **Example of versioning**. Versioning describes strategic (v1 and v2) and tactical versions (v1.1 and v1.2), as well as options (v1.2a, v1.2b and v1.3b) that precede a tactical version. Option v1.2b was chosen and promoted to v1.2.

7.3. Standards

The metamodel is updated to enable standards-enabled engineering practices. There are two types of standards: non-technical and technical standards. They are an abstract model established by a team, usually one with sufficient, direct or delegated, authority. Standards model parts of a software product. For non-technical standards, the model provides universal concepts to be followed and often exists as principles to be adhered to. Non-technical standards are not in scope in this paper. For technical standards, the model provides logical or physical elements or attributes that other technical models follow, and utilizes pattern matching to localize the standard within the compared model.

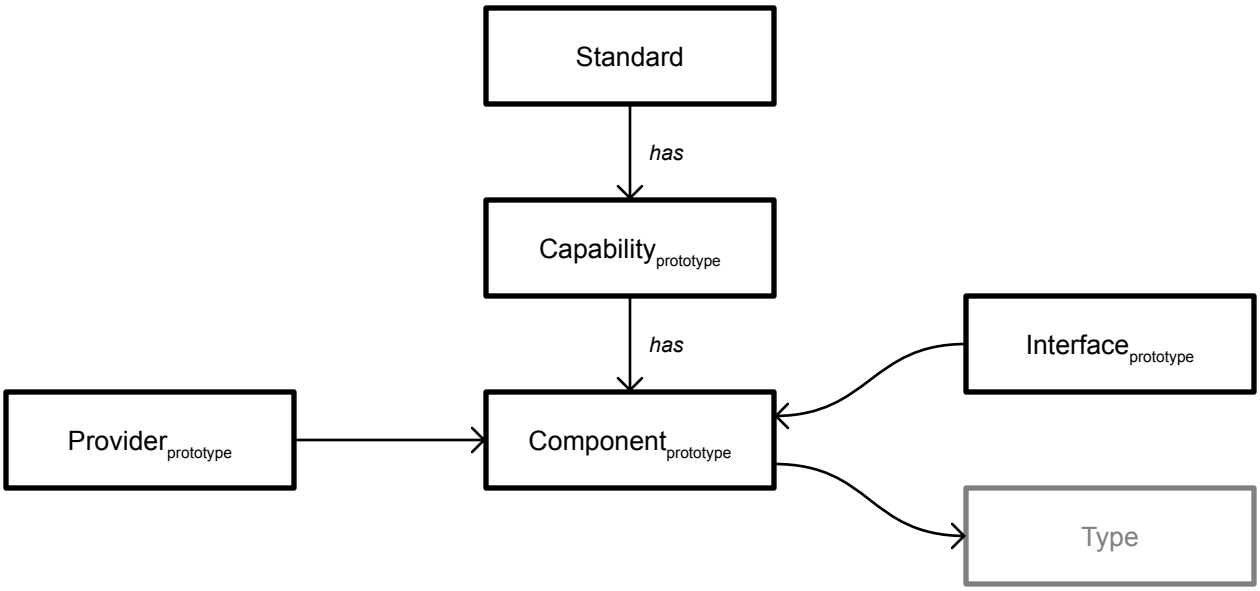

Figure 25. **Overview of standard elements**. The metamodel is extended with a Standard element that provides abstraction to other elements using prototypes.

7.3.1. Standard

A Standard is an element that models a technical pattern to scaffold, modify or measure other technical models. Ontologically, a standard consists of a logical model and/or physical model. A standard is similar to the feature element because the feature is the root element in the logical model. Therefore, like a feature, a standard consists of capabilities, components, etc. The updated metamodel for standards also adds a

pseudo-element called a Prototype that abstracts any element such as a capability.

7.3.2. Prototype

The Prototype element is a pseudo-element that serves as a surrogate for another element of the same type. In the metamodel, capabilities, components, providers and interfaces can be substituted with a prototype. A prototype is used to abstract a standard model such that the standard model can be matched with a target model. Matched standards can be applied to scaffold (construct), modify or measure the target model. A prototype element can only be associated within a standard context.

7.3.3. Matching

Standards contain patterns, and pattern matching yields zero or more locations within a target model that a standard model matches. Localization is a general concept that describes where a pattern matches. Its final implementation can vary across different implementations or optimizations of the software product model. It is omitted in the interest of keeping to the core concept rather than its mechanism.

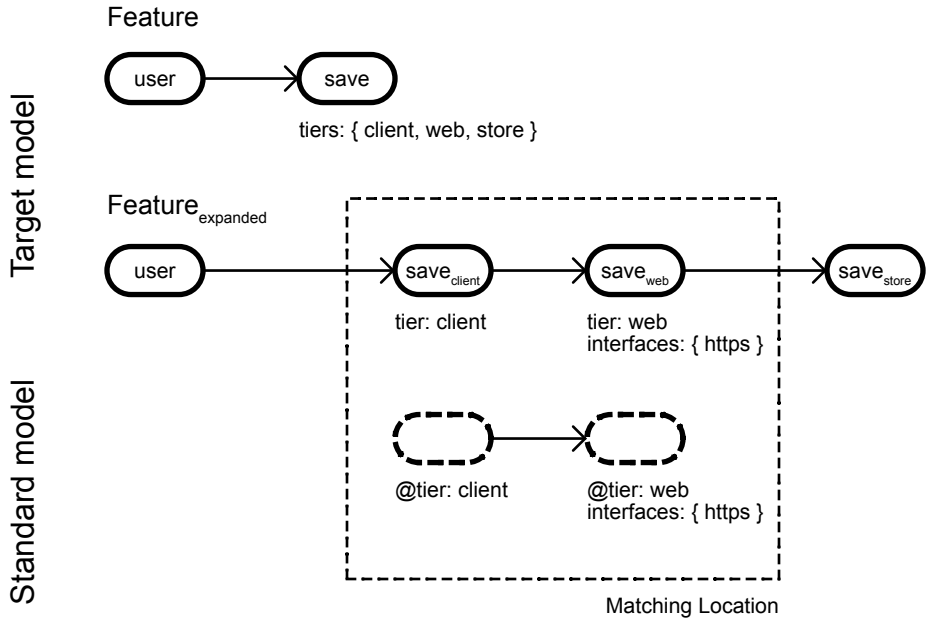

Figure 26. **Example of pattern matching**. An example of a standard describing an HTTP interface between client and web tiers, and its matching location on a target model.

7.3.4. Application

Standard application is an algorithmic utilization of a standard and its matching pattern locations in a target model. A standard can be used to scaffold, modify or score a target model.

a. Scaffolding. A standard can be applied to scaffold a new target model where prototypes must be replaced with actual elements. Standards scaffolding is useful during architecture or engineering phases of a software product because it increase efficiencies in design or specification.

b. Modification. A standard can be applied to modify an existing, target model where non-pattern-matching standard attributes are added or updated to their corresponding elements in the target model. Standards modification is useful when standards undergo change where target models may require recent versions of a standard.

c. Scoring. A standard can be applied to score an existing, target model where various scoring algorithms produce a value that indicates the distance between the standard and target model. In most cases, scoring algorithms will measure the alignment or deviation the target model has relative to the standard, therefore, a score of zero indicates 100% alignment. Standards scoring is useful during delivery, especially in governance, where an alignment score may optimize an approval process.

8. Conclusion

The Software Product Model provides both fundamental and novel strategies to solving the deficiencies in software documentation. Its complete, end-to-end modeling approach that integrates a highly utilized conceptual model with the technical models will yield precise models. Its use of behavioral composition with capabilities to derive software structures will produce strong software structures. I combine these approaches into an ontology for software products and a software metamodel that expresses the ontology into a tangible, schematic form.

In this paper, I introduced novel ideas. I intersected mathematics and software delivery with category theory and function composition. And, I introduced dynamic elements that can derive structure from composition, compute quality benchmarks or replicate capabilities across tiers. Due to is novelty, tooling is still nascent. My hope is for this paper to shed light on a comprehensive and precise approach to documentation and grow a community around it.

References

E. Cheng. *The way to think about math. The Art of the Logic in an Illogical World*, page 23. 2018.

E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. 2003.

F. P. Brooks, Jr.. *The Mythical Man-Month, Anniversary Edition*. The Tar Pit, page 6. 1995.

J. Bang-Jensen, G. Gutin. Springer-Verlag. *Digraphs Theory, Algorithms and Applications*. Chapter 1. Basic Terminology, Notation and Results. Page 2. 2007. (Link).

J. Zachman. Zachman International Enterprise Architecture. *Conceptual, Logical, Physical: It is Simple*. 2000. (Link).

J. Nguyen, E. Mateo. MIT. *Navigating the Network: Real-Life Applications of Graph Traversal Algorithms*. Page 6, 2023. (Link).

Khan Academy. *Composing functions*. n.d., Accessed: 2025-04-25. (Link).

M. Conway. Datamation. *How do Committees Invent*. 1968. (Link).

Object Management Group. *Business Process Model and Notation (BPMN) Version 2.0.2.*, December 2013. (Link).

N. Rozanski, E. Woods. *Software Systems Architecture, Working with Stakeholders Using Viewpoints and Perspectives*, page 23. 2016.

Stanford University, Metaphysics Research Lab. *Category Theory*. 1996. (Link).

T. Judson. Stephen F. Austin State University. *Abstract Algebra, Theory and Applications*. 2012. (Link).

T. Preston-Werner. The Semantic Versioning. *Semantic Versioning 2.0*. n.d., Accessed: 2025-04-30. (Link).