What we do

Epistimis makes Epistimis Modeling Tool (EMT), a tool for process design and evaluation. EMT does for (software) processes what Electronic Design Automation (EDA) tooling does for chip design. It is a platform that allows process designers to specify the data and data flows planned for a process and then evaluates sets of rules against those designs.

It can be used for any process design – no matter how that process will be implemented. EMT can already generate source code in a variety of languages for process designs that will be implemented in software. If desired, that capability can be extended to generate process details as step-by-step instructions (from text to IKEA style instruction diagrams), workflow configuration, or other process specification formats.

The initial set of rules EMT will evaluate is data privacy rules, starting with GDPR. Epistimis will continue to add US state and federal laws as additional rulesets based on demand. These rulesets will be licensed to users, and maintained so that users can be assured they are always working with the relevant sets of rules. It will also provide a rule specification capability for custom rules such as corporate privacy policies.

We plan to add libraries of predefined functionality such as AWS offerings that can be dropped into process designs. Using these libraries will not only speed process design, they will also enable generation of configuration information from the design that matches how an offering is used in the design.

Outputs generated from the tool will consist of 1) Rule evaluation results; 2) Optional code/ configuration generation. Code/ configuration generation will only be done if all rule checks succeed – so users will always know the results are correct by construction. Rule evaluation results can be used for Privacy Impact Assessments (PIAs), eliminating the need for manual input in the PIA process.

Tutorials

1. Basic Conceptual Modeling
2. Conceptual + Components (includes Queries)
3. Wiring Diagrams
4. Privacy / OModel
5. Code Generation

Basic Conceptual Modeling

EMT starts with conceptual data modeling. Conceptual data models focus on the underlying semantics behind data while ignoring implementation details like units or storage. For example,

think of the concept of ‘Temperature’. At a conceptual level, I don’t have to worry if I’m dealing with *F or *C. Nor do I care if a Temperature value will be stored in a float, an int, or a string. All I care about initially is that I’m dealing with Temperature – everything else is a detail that I can put off for now, and possible ignore forever.

Why can I ignore it? Because in most cases the rules don’t care about those details. Privacy rules, such as those that deal with ‘Special Categories of Data’ don’t refer to units or storage details. If my objective is to create a model that can be evaluated against those rules, there is no need to include that additional detail. I can include that detail later if I want to do something with that detail (like code generation).

Data modeling starts with ‘Observables’. An Observable is the fundamental concept behind UDDL modeling. It consists of only a name and a description. Because it consists of only this, the name and description are important. They define the semantics – what it means - of each Observable. This is so important that EMT supplies predefined sets of Observables. The core set – the FACE Shared Data Model – has been defined by the FACE Standards Committee. Because FACE was originally developed for the US Military, this SDM is focused on physical / engineering concepts like time, electric current, and location. The content of this SDM is managed by the FACE Standards Committee. Changes and additions are possible but must go through a rigorous review process.

EMT also supplies a Privacy Shared Data Model that contains extensions specifically addressing privacy concepts that are not already defined in the FACE SDM. These include things like special categories of data and organization/ institution/ business concepts. As with the FACE SDM, this SDM is also managed, with changes and additions going through the same rigorous review process. This SDM is managed by Epistimis in collaboration with regulators, partners, and customers.

Each Observable only tracks a single type. Obviously, we need a way to track groups of data – we need data structures. That’s where Conceptual Entities come in.

Conceptual Entities contain Composition elements. Each of those Composition elements has a rolename, a type and a cardinality and a description – all inherited from Characteristic. We could say that the type of each Composition element is an Observable but that would limit our ability to create structures of structures. Instead, we want the ability to set the type of each Composition element to be either an Observable or an Entity. That means we need a new base class – Composable Element.

Note that Composition elements have ‘rolename’ not ‘name’. The reason is this: The ‘rolename’ specifies the role that Composition element plays in that Entity. It is possible for an Entity to have multiple Composition elements with the same type but different role names. The lowerBound and upperBound values of each Composition element determine the overall cardinality of that element. If the lowerBound = 0 and upperBound = 1, then that element is optional. If lowerBound = 1 and upperBound = 1, then that element is required. If lowerBound = 0 and upperBound = -1, then that element is an unbounded collection. Other lowerBound and

upperBound values indicate bounded collections that have a minimum and maximum number of elements.

Note that Entity can specialize another Entity. This is like but not the same as inheritance in OO languages. Here, specialization only means that the specialization has all the Characteristics of the thing it specializes. It does not imply 'is-a' or polymorphism – why? Because these are just data structures – they have no functionality – therefore we can say nothing about 'is-a' / polymorphism.

In most cases, Observables and Conceptual Entities will be enough to model everything you need. But sometimes, you need to define associations between Conceptual Entities that have attributes. For that, you need Conceptual Associations.

Conceptual Associations are association classes. As you would expect, a Conceptual Association is a Conceptual Entity (it can have Composition Elements) with the addition of relationships with other Conceptual Entities. These relationships are handled by Conceptual Participants. Each participant in a Conceptual Association has a rolename, type, cardinality, and a description. Each participant attribute has the same meaning as the corresponding attribute for a Conceptual Composition. In addition, each participant has sourceLowerBound and sourceUpperBound for defining the cardinality when traversing from the participant's type to the association.

While the FACE SDM does not include any Conceptual Entities or Conceptual Associations, the EMT Privacy SDM does. It includes things like NaturalPerson and other data structures referenced in various rules. These Conceptual Entities are managed along with the rest of the Privacy SDM.

When modeling your data, you should prefer these predefined concepts because the prepackaged rule sets already work with them. If you are concerned that these SDM concepts contain more than you plan to use, don't be. You can select slices or subsets of Conceptual Entities using Conceptual Views.

Conceptual Views use SQL SELECT syntax. The only difference from the standard use of SQL is this: When SQL is used with a DBMS, the returned result is a set of data selected from data available in the DBMS at that time. When SQL is used in a UDDL Conceptual View, it defines a selection / subset / slice of the data that you want to use at a particular point in your model. It is a static selection in UDDL whereas it is a dynamic selection when used with a DBMS.

View SQL SELECT statements are a subset of SQL – but they cover quite a bit. You can:

1. select specific composition elements;
2. JOIN between different Conceptual Entities;
3. Use aliases
4. Use WHERE clauses

And more.

This tutorial will not go into detail on Views. For that, see the tutorial on Concepts + Components.

One last thing: You may have noticed Domain and BasisEntity in this final class diagram. Those are advanced concepts that will not be covered in this set of tutorials. For details, see the Data Modeling Guide and the UDDL specification.

Conceptual + Components (includes Queries)

Defining data is only the first step. Once you define data, you need to use it. That means defining Components and the Connections that use that data.

The first step in that process is Views. In the Basic Conceptual Modeling tutorial, you learned that a View is how you can select slices or subsets of Conceptual Entities.

In this class diagram, you see details on a Conceptual View. Each View can be a single Query or a CompositeQuery. This approach embodies the Composite pattern, supporting any level of complexity needed. For now, we consider just single Queries.

Each Query has a specification. That specification is an SQL SELECT statement. Queries support SQL clauses in the form of:

> **SELECT** select_clause
> **FROM** from_clause
> **JOIN** join_clauses
> **WHERE** where_clause
> **ORDER BY** order_by_clause

Note that JOINs in a Query are not dynamic. Instead, they are used to navigate Association participant paths. For details on query syntax, see the Data Modeling Guide.

As you might guess, using data means defining components that use that data. In FACE, a component is called a UnitOfPortability. They come in 2 flavors: PortableComponent and PlatformSpecificComponent. The distinction between these does not matter at the Conceptual level. These components each have one or more Connections, each of which defines either an input to or output from that component.

The Connection Types are:

- ClientServerConnection
- PubSubConnection (Publish – Subscribe)
    - QueuingConnection
    - SingleInstanceMessagingConnection

The FACE docs provide the detailed differences between these. For our purposes right now, the important thing to note is that Connections specify how data is transmitted into/out of a component and reference requestType/responseType/messageTypes that specify the data that is transmitted over that connection.

The official FACE spec only supports FACE UoP MessageType which uses Templates that are bound to Views. FACE assumes that Components only work with fully specified data – data that includes both units (Logical) and storage (Platform) information. Templates are useful when you want to use code generation because they enable structuring the data in any way you want. That level of detail is unnecessary when working at the conceptual level / doing rule evaluation.

To reduce the modeling effort required, EMT extends the FACE spec so that ConceptualView and ConceptualEntity can also be referenced directly on a Connection, bypassing the need for a Template. Unless you plan to use a Conceptual Entity in its entirety, you will need to specify a ConceptualView on a Connection.

If you do want to do code generation, then you'll want to use UoP MessageType on Connections. As you can see from this diagram, these are then bound to Queries.

Wiring Diagrams

Defining components, either PortableComponents (PCs) or PlatformSpecificComponents (PSCs), just identifies the capability you want available. You still need to put it to use. To do that, you need a wiring diagram. These diagrams route data between the outputs and inputs of component instances.

IntegrationModels (the formal name for wiring diagrams) consist of 2 categories of wiring. Standardized predefined functionality (source, sink, filter, aggregation, transformation and basic transportation) is handled using TransportNodes. Application / custom functionality is handled by UoPinstances, each of which realizes a UnitOfPortability previously specified elsewhere. All nodes, either TransportNodes or UoPInstances, are wired together in an IntegrationContext using TSNodeConnections. Note that TransportNodes themselves are also part of a specific IntegrationContext, whereas UoPInstances are not – the same UoPInstance could appear in multiple IntegrationContexts.

Before going any further, I need to make sure the concept of 'realization' or 'realizes' is clear. Realization, as seen here, is a way of associating a UnitOfPortability (effectively, a class) with all its UoPInstances (instances of that class). You will see the term 'realizes' again in the tutorial about code generation.

There is another minor distinction between UoPInstances and TransportNodes. UoPInstances are wired via UoPEndPoints that reference Connections specified on the UnitOfPortability realized. Those Connections refer to the MessageType (or ConceptualView or ConceptualEntity) defining the data moving through that interface. TransportNodes are wired via TSNodePorts that directly reference the MessageType (or ConceptualView or ConceptualEntity).

It should be noted that the predefined functionality handled by TransportNodes is predefined at the meta-model level. The Privacy Shared Component Model (SCM) will include numerous predefined components (e.g. AWS/ Azure/ GCP / OCI functionality, and more) – those will use UnitOfPortability.

Privacy Specific Modeling

Everything discussed so far has been about (usually software) process design. Many privacy laws require knowing more than just the how and what of a process. To support that, EMT introduces 2 layers above the UDDL/FACE specification:

- Privacy
- Omodel

EMT introduces two layers to separate reusable pieces from model parts that are specific to a given organization. The Privacy layer includes most of the extensions to the UDDL/FACE specifications. This separation is maintained to retain UDDL/FACE compatibility if needed US DoD purposes. In addition, the 'Privacy' layer contains everything except privacy oriented except organization specific information, making it possible to create Privacy model fragments that can be reused. Example information tracked in the Privacy layer include purpose, PET, processingBasis, jurisdiction, and more.

The OModel layer contains organization specific information. It may be possible that SaaS providers will publish reusable OModel fragments – that is not yet clear. Example information tracked at the OModel layer include customer/ user base, dataSource (1st vs. 3rd Party), responsible parties and more.

These layers will evolve rapidly as additional privacy rules are included.

Code Generation

Code generation is optional with EMT. EMT currently supports creation of code stubs in multiple languages, and data structure definitions in even more. Because these stubs are generated from the EMT model, they are always correct by construction. Generated code will always include any needed dynamic checks (e.g. consent). Code will only be generated if the EMT model passes all rule checks – no code will be generated that would break rules.

Code generation is optional – and requires some additional work to enable. Specifically, modelers must specify both units and storage for all the data they intend to use. In UDDL, units are specified at the Logical level, and storage is defined at the Platform level.

The Logical level has the same core organization as the Conceptual level. There are Measurements which realize Observables, Logical Entities that realize Conceptual Entities, Logical Compositions that realize Conceptual Compositions, Logical Associations that realize Conceptual Associations, and Logical Participants that realize Conceptual Participants. The Logical level also has Views/Queries that realize their Conceptual equivalents.

What does ‘realize’ mean in this context? It is similar to the meaning discussed previously with UnitOfPortability and UoPInstance. Just as a UoPInstance is a realization of a UnitOfPortability, so here a Logical element is a realization of its Conceptual counterpart – it is an embodiment of that concept with specific information, in this case units.

Notice that the major difference for the Logical level is how it handles Measurements. Because FACE/UDDL was designed initially for US DoD use, it has complete support for engineering/ physical world measurements. While the metamodel for this is complex, FACE/UDDL comes with a Shared Data Model that includes definitions for many significant measurement systems. In addition, EMT adds measurement systems for privacy specific concepts. Modelers should need only choose from available options. If you discover that you need a measurement system not currently supported by EMT, contact Epistimis and we will update the Privacy SDM with the measurement system you need.

The Platform level has the same core organization as the Logical and Conceptual levels, with the same realizations – only the Platform level realizes its Logical Level counterparts. The major difference at the Platform level is PlatformDataType – this is used to specify the storage used for the data being realized.

So how does all this work in a model? In this example, we see the Conceptual Entity ‘Dog’. It has several Composition elements, each an Observable. We also see 2 Logical Entities ‘Dog’ that both realize the same Conceptual Entity ‘Dog’. The left ‘Dog’ uses metric units while the right ‘Dog’ uses English units. Note that, although not indicated in this diagram, the individual Logical Compositions realize their corresponding Conceptual Composition elements.

Note that each Logical ‘Dog’ also has 2 Platform Entities ‘Dog’ that realize it. Each Platform Entity ‘Dog’ realizing the same Logical ‘Dog’ specifies different data storage. Note that it is possible to define 2 different Platform Entities that have identical storage allocation but realize different Logical Entities – these are treated as unique Platform Entities.

What does all this mean?

1) Entities that realize the same thing can participate in automatic data conversion. Think about the Observable ‘Money’. A Conceptual Entity that has a ‘Money’ element may have a Logical Entity that realizes the ‘Money’ element in Euros and another realization in USD. Because both these realize the same Observable, it is possible to automatically convert values in those fields from one set of units to the other. This automated conversion will automatically be added to generated code whenever needed.

   Note that simply having related units is not enough – both Logical realizations must realize the same Observable. Consider the Observables ‘Temperature’ and ‘TemperatureDelta’. All Logical realizations of either of these Observables will always use the same units (*F, *C, *K) – but we cannot automatically convert between these realizations because Temperature and TemperatureDelta don’t mean the same thing.

2) Though not directly supported in FACE/UDDL, EMT will support the use of Platform Entity 'generics' that can be repurposed by simply setting the Logical Entity they realize. This will prevent combinatorial explosion that could otherwise occur.
3) To make this easier / less tedious, EMT also supports several types of defaults. For example, every Measurement can have a default storage type. Using default storage types, a modeler can generate code without specify any model specific Platform level info. (NOTE: Using default storage implies that every Composition element specified at the Logical level will be instantiated and will use the logical rolename – both of which can be overridden if a Platform Entity is specified realizing that Logical Entity.)

While code generation is optional, it has several advantages. As stated above, all generated code is correct by construction. It won't be generated if it breaks any rules. If it is generated, it will compile cleanly and have any dynamic checks needed included. And, it will be possible to generate code in multiple languages and be assured that all generated code is mutually compatible – because it all comes from the same EMT model.

Most companies do not start with a clean slate – instead they start with an existing code base. It is possible to reverse engineer existing code to create an EMT model. There are multiple tools available that can be used to help with that reverse engineering. Epistimis will consider adding EMT specific reverse engineering capability if there is customer demand.