On a Metasemantic Protocol for Modeling Language Extension

Ed Seidewitz[a]
Model Driven Solutions, 235A E. Church St., Frederick, MD, U.S.A.
ed-s@modeldriven.com

Keywords: Modeling Language Extension, Modeling Language Semantics, Metamodeling, SysML.

Abstract: A *metaobject protocol* is an object-oriented interface that allows a programming language to be efficiently extended by users of that language from within the language itself. A *metasemantic protocol* is a generalization of that idea, providing a mechanism to allow users of a formally defined modeling language to syntactically and semantically extend that language from within the language. Such an approach is fundamental to the language architecture being developed for the proposed second version of the Systems Modeling Language (SysML). SysML v2 is being effectively defined as an extension to a foundational Kernel Modeling Language (KerML), and then users can define domain-specific languages in the same way as extensions of SysML. This approach is already being worked out in the ongoing pilot implementation of SysML v2, but there is still much to do before the vision of a true metasemantic protocol is fully realized.

1 INTRODUCTION

Almost thirty years ago, Gregor Kiczales and his colleagues proposed the idea of a *metaobject protocol* for programming languages (Kiczales, 1991). They defined metaobject protocols as "interfaces to the language that give users the ability to incrementally modify the languages behavior and implementation". With this approach, "users are encouraged to participate in the language design process", but without compromising "program portability or implementation efficiency".

For better or for worse, few languages in the ensuing years provided metaobject protocols for their users. As Kiczales et al. admit in their original work, developing such protocols is hard, and they did not claim to have entirely solved the problem. Further, the implementation they provided was embedded within the Common LISP Object System (CLOS). CLOS provides powerful metalinguistic tools for such an implementation, but it has a LISP syntax that is daunting to many and a dwindling user community that has limited the understanding of the proposed meta-object protocol approach.

Nevertheless, the need for user-specialized languages did not go away. Today we see much work on domain-specific programming languages (DSLs), sometimes created from scratch using language workbench technology (such as Xtext [1] or JetBrains/MPS[2]). However, some level of domain-specificity can be provided in any programming language using specialized libraries. And some general purpose languages (such as Ruby [3]) are particularly well-suited to being tailored in this way to provide so-called "internal DSLs", somewhat in the spirit of the original metaobject protocol concept.[4]

The tension between general-purpose and domain-specific languages also exists in the modeling world. Well-known general-purpose modeling languages include the Unified Modeling Language™ (UML, 2017), the Systems Modeling Language™ (SysML, 2018) and the Business Processing Model and Notation™ (BPMN, 2014), all standardized by the Object Management Group (OMG).[5] On the other hand, DSL workbenches can also be used to create very targeted domain-specific modeling languages (e.g., see Voelter, 2018).

Further, the *profile* mechanism in UML is essentially intended as an internal DSL capability.

[a] https://orcid.org/0000-0002-7647-4769
1 See http://xtext.org
2 See https://www.jetbrains.com/mps/
3 See https://www.ruby-lang.org/
4 See, for example, https://thoughtbot.com/blog/writing-a-domain-specific-language-in-ruby
5 All indicated trademarks are of the Object Management Group.

Indeed, SysML is currently defined as a standard profile of UML, tailoring the more general language for use within the more specific systems engineering community. User-defined profiles are also often used within organizations and teams to succinctly capture modeling patterns and domain-specific concepts commonly used within those groups.

Unfortunately, profiles in UML do not actually provide a full language extension mechanism. In particular, the lack of a formal semantic definition for UML means that profiles cannot provide any precise way to extend the semantics of the language. (While Foundational UML and succeeding standards based on it do provide precise execution semantics for a subset of UML, this subset does not yet include a precise semantics for profiling.) At best, profiles provide a formal means for syntactic extension UML. However, most UML modeling tools provide only limited support for even syntactic tailoring of UML using profiles, often with little more support of user profiles in UML 2 than the basic stereotyping and tagging capabilities of UML 1.

However, in 2017 OMG issued a request for proposals (RFP) for version 2 of SysML that no longer requires the language to be a profile of UML (SysMLv2, 2017). But the RFP includes requirements calling for the language to have a formal semantic basis that could be truly extended by users. In addition, even though SysML v2 no longer needs to be based on UML, it would be desirable to continue to have interoperability between software and systems modeling in the context of today's many software-intensive cyber-physical systems.

In response, the cross-industry team currently working on a submission to the RFP[6] has proposed a language architecture in which the desired SysML v2 is based on a *Kernel Modeling Language* (KerML) that acts as a general foundation for building modeling languages. KerML moves a step beyond the current OMG Meta Object Facility (MOF) technology, which formalizes only the definition of the abstract syntax of a modeling language, by also providing a formal grounding for the definition of the semantics of modeling languages built on it. And, as with the metaobject protocol, the idea is that, just as SysML v2 is based on KerML, users can, in the same way, build their own more tailored domain-specific language extensions, syntactically and semantically, on SysML v2. However, in contrast to the inherently object-oriented programming mechanism proposed by Kitczales et al., this approach can be more generally termed a *metasemantic protocol* for modeling language extension.

This protocol can be summarized by the following steps:

1. Define a model library that specifies the formal semantic concepts needed in a language extension.
2. Define the abstract syntax for the language extension in terms of patterns of usage of the model library that can be mapped back to the abstract syntax of the language being extended.
3. Define a user-friendly surface syntax for the language extension that maps to the extended language abstract syntax.

The first two steps of the protocol have been applied to UML and SysML in (Bock 2011) and (Bock 2019). This paper gives an outline of how the protocol is being further developed in the ongoing work on SysML v2, and, in particular, examples of the application of the third step. While the full SysML v2 definition includes a complete abstract syntax as well as both graphical and textual concrete surface syntaxes, the presentation in this paper will focus on the textual surface syntax, due to space limitations.

Section 2 presents an overview of basic structural modeling in KerML and its model-library-based approach to language extension. Section 3 describes how the metasemantic protocol is used to build SysML v2 on KerML, and Section 4 then discusses how the same protocol can be used to build domain-specific language extensions to SysML v2.

2 KERNEL MODELING LANGUAGE

KerML provides a basic set of modeling capabilities based on a formally defined semantic core. For the purposes of this paper, we will discuss only some simple structure modeling capabilities and only describe the underlying core semantics informally. The focus, in subsequent sections, will then be on how successive layers of user-focused modeling languages can be built, syntactically and semantically, on this foundation.

UML and SysML are primarily graphical languages, and SysML v2 will also define graphical views of models, as SysML v1 does. However, unlike UML and SysML v1, KerML and SysML v2 will also

[6] For public incremental pre-submission releases of SysML v2 from the submission team, go to http://openmbee.org/sysml-v2-release.

both provide standard textual modeling notations. For the purposes of this paper, and for general discussions of language extensibility, it is easier to focus on just this textual notation.

So, to begin, consider a very simple model of a car that has an engine and four wheels, and which may have a driver. Using the KerML textual notation, this model may be represented as follows:

```
class Car {
    feature driver: Person[0..1];
    composite feature engine: Engine[1];
    composite feature wheels: Wheel[4];
}
```

As in UML, a *class* specifies a type whose instances are *objects* that have various *features*. A feature defines a mapping from instances of the featuring class to instances of the type of the feature, with some multiplicity. A *composite* feature is considered to be an integral part of the featuring object, while a non-composite feature is simply referential.

Next, define an *association* between an engine and the wheels driven by that engine:

```
assoc DriveTrain {
    end engine: Engine[0..1];
    end wheel: Wheel[*];
}
```

Again, as in UML, an association is a type whose instances are *links* between objects of the types of the *end features* of the association. The basic difference between an end feature and a regular feature is that the multiplicity of an end is for navigation *across* the association. That is, the above association specifies that one engine can be linked to multiple wheels, but a wheel can only be linked to (at most) a single engine. Nevertheless, each instance of the association will link exactly *one* engine to exactly *one* wheel.

Given the `DriveTrain` association, we can now extend our earlier Car model to reflect the fact that (in this simple model) exactly two of the wheels of a car are driven by the car's engine:

```
class Car {
    feature driver: Person[0..1];
    composite feature engine: Engine[1];
    composite feature wheels: Wheel[4];
    connector drive: DriveTrain
        from engine[1] to wheels[2];
}
```

A *connector* is a feature whose instances are links specifically between values of features of the containing class of the connector. Thus, the connector `drive` specifies that the single engine of the car has links to two of the wheels of the car. It is fundamental to the semantics of the connector that the linked engine and wheels are parts of the *same* car.

Now, some of the semantic statements made in the description of the example above are actually captured more formally in terms of elements of a *Kernel Model Library* that is integral to the specification of KerML. For instance, this model library includes the following elements (this is a simplification of the actual library, adequate for our present purposes):

```
class Object specializes Anything;

class BinaryLink specializes Object {
    end source: Anything[*];
    end target: Anything[*];
}
```

The type `Anything` is at the root of the KerML type specialization hierarchy. The class `Object` is then the base type of all classes of things with identity and properties that may change over time (as distinct from data values, not considered in detail in this paper, which are immutable things with value semantics). The class `BinaryLink` is the base type of all binary associations, whose instances are objects that link a source thing to a target thing.

The notation previously used for the association `DriveTrain` is actually a shorthand for the more explicit class model shown below:

```
class DriveTrain specializes BinaryLink
{
    end engine: Engine[1]
        redefines BinaryLink::source;
    end wheels: Wheel[*]
        redefines BinaryLink::target;
}
```

That is, `DriveTrain` is a class that specializes `BinaryLink`, redefining its source and target ends to have more restrictive types and multiplicities. In this way, the semantics of associations are reduced to the semantics of classes with end features (and the only difference between end features and the fundamental semantics of regular features is in how multiplicity is interpreted).

The earlier model of a car, as extended with a connector, is then itself a shorthand for:

```
class Car specializes Object {
    feature driver: Person;
    composite feature engine: Engine;
    composite feature wheels: Wheel[4];
```

```
    composite feature drive:
                    DriveTrain[*] {
        end engine: Engine[1]
            redefines DriveTrain::engine
            subsets Car::engine;
        end wheel: Wheel[2]
            redefines DriveTrain::wheel
            subsets Car::wheels;
    }
}
```

That is, the connector `drive` is actually just a composite feature whose type is the association `DriveTrain`. But, further, the actual connection between the `engine` and `wheels` features of `Car` is modeled using *local redefinitions* of the end features of `DriveTrain`. For example, the redefinition of `wheel` specifies that, in the context of a car, the engine is linked to exactly two wheels and the linked wheels must be a subset of the overall set of wheels that are part of the car.

Allowing a feature to have nested features is a critical capability in KerML that is not found in UML or most other modeling language (nor even in ontological languages like OWL). One can think of nested features as being part of a local specialization of the base type of the containing feature, but with a specifically contextual semantics. This contextuality is what ensures, for example, that the feature `drive` in the class model above fundamentally has the desired connector semantics in which the engine and wheels linked by the feature must be part of the same car.

As we will see, this ability to define features contextually nested in other features is especially important for achieving semantic extensibility.

3 SYSTEMS MODELING LANGUAGE

The current proposal for SysML v2 is to define it as an extension to KerML. Semantically, this extension is captured in the *Systems Model Library* for SysML, which extends KerML's Kernel Model Library. The elements of the Systems Model Library capture the key systems modeling concepts that are important for the systems engineering user community of SysML.

For example, a central idea in the structural modeling of systems is that a system can be decomposed into *parts,* and that these parts can have *ports,* which define specific points through which the parts can be interconnected. This conception is captured as follows in the model library (again, this is a simplification of the actual SysML model library):

```
class Part specializes Object {
    composite feature ports: Port[*];
}

class Port specializes Part {
    feature inputs: Anything[*];
    feature outputs: Anything[*];
}
```

Given the concepts of parts and ports, we can then provide the ability to specify the *interfaces* between parts in terms of how their ports may be interconnected. Unlike the way the term "interface" is used in software engineering, to a systems engineer, an interface generally specifies not just the externally visible face of a single part, but *both* ends of an allowed connection between two parts. Thus, we define an interface as an association that connects two ports:

```
class Connection specializes
        BinaryLink, Part {
    end source: Part[*]
        redefines BinaryLink::source;
    end target: Part[*]
        redefines BinaryLink::target;
}

class Interface specializes Connection
{
    end source: Port[*]
        redefines Connection::source;
    end target: Port[*]
        redefines Connection::target;
}
```

Given just the above systems engineering concepts, we can now improve our earlier model of a drive train into a (slightly) better systems engineering interface model:

```
class DrivePort specializes Port {
    feature torque: Torque
        subsets outputs;
}

class DrivenPort specializes Port {
    feature torque: Torque
        subsets inputs;
}
```

```
assoc DriveTrain specializes Interface
{
    end drive: DrivePort
        redefines Interface::source;
    end driven: DrivenPort[*]
        redefines Interface::target;
}
```

Unlike the model given in the previous section, the interface model for `DriveTrain` is no longer specific to the `Engine` and `Wheel` types, but can be used to specify the interface between any two parts that have `DrivePort` and `DrivenPort` connection points.

Note that the `DrivePort` class defines a torque output and that the `DrivenPort` class has a corresponding torque input. This means that the two port definitions are *compatible* for specifying the opposite ends of an interface. In general, two ports are compatible if, for any input of one of the ports, there is a corresponding output of the other port.

Next, we can use the above interface definition for `DriveTrain` in our `Car` model (presuming that `Engine` and `Wheel` are also redefined as parts):

```
class Car specializes Part {
    feature driver: Person;
    composite feature engine: Engine {
        composite feature enginePort:
            DrivePort subsets ports;
    }
    composite feature wheels: Wheel[4];
    composite feature driveWheels:
            Wheel[2] subsets wheels {
        composite feature wheelPort:
            DrivenPort subsets ports;
    }
    connector drive: DriveTrain
        from engine::enginePort[1]
        to driveWheels::wheelPort[2];
}
```

Now, rather than the `drive` connector being directly between the engine and the wheels, it is a connection between the corresponding ports on those parts. Note that we have added the ports locally to the `engine` and `driveWheels` features of `Car`, rather than to the `Engine` and `Wheel` classes themselves, to emphasize that these are *localized* connection points to the engine and wheels in a specific car. Further, this allows us to more specifically model that only two of the wheels of a car are drive wheels and that *only* these wheels have the ability to be interfaced to the engine.

The Systems Model Library provides the additional semantic concepts necessary to use KerML in a more specialized domain like systems engineering. However, syntactically, the resulting system engineering models are rather cumbersome and inconsistent with the kind of structural modeling terminology familiar to system engineers. This may be remedied by providing a specialized surface syntax for SysML, in just the way that the notation for associations and connectors in KerML was really just syntactic sugar for specific patterns of usage of classes and features.

Thus, using the specialized SysML textual notation, the interface definition given above for `DriveTrain` is written as follows:

```
port def DrivePort {
    out torque: Torque;
}

interface def DriveTrain {
    end drive: DrivePort;
    end driven: ~DrivePort[*];
}
```

In this syntax, special keywords like `port` and `interface` are used as markers for corresponding linkages to the semantic library. Further, the notation `~DrivePort` specifies a conjugation of the port definition `DrivePort`. The conjugate of a port definition is a port definition with similar features, but with inputs and outputs reversed. In this way, a conjugated port definition is conveniently always compatible as the type of the opposite end in an interface definition to a port typed by the original port definition.

In SysML terminology, the class of a part is called a *block.* Therefore, the model of a `Car` is given in SysML by the following block definition:

```
block Car {
    ref driver: Person;
    part engine: Engine {
        port enginePort: DrivePort;
    }
    part wheels: Wheel[4];
    part driveWheels: Wheel[2]
        subsets wheels {
        port wheelPort: ~DrivePort;
    }

    interface drive: DriveTrain
        connect engine::enginePort[1]
        to driveWheels::wheelPort[2];
}
```

This provides a more succinct structural model of a car, using more familiar systems engineering terminology. However, the underlying semantics is still formally specified by mapping this notation back

to KerML and expanding to the required patterns of usage of the Systems Model Library.

4 USER DOMAIN-SPECIFIC LANGUAGES

SysML v1 is defined as a standard profile of UML v2. However, it also allows the profile capability of UML to be leveraged so that users can define profiles that further specialize SysML. Indeed, this capability is widely used by major SysML-using organizations, which often define corporate and even project-specific profiles of SysML.

Unfortunately, as discussed previously, the UML profile mechanism falls far short of a full domain-specific language extension capability. While SysML tools are well-tailored to the modeling and diagrammatic specializations of the standard SysML profile, they rarely allow the same level of tailoring for a user-defined profile. It is thus often frustrating and cumbersome to use a SysML profile, which does not allow true syntactic or semantic extensibility.

An important requirement for SysML v2 is to improve the ability for users to extend the language. The proposal is to do this by leveraging the same approach to linguistic extension used to build SysML v2 on KerML in order to build user domain-specific languages on SysML. This is quite similar in concept to the intent of further leveraging the UML profile mechanism in SysML v1, but using a much more powerful mechanism for linguistic extensibility than what is possible with the UML profile mechanism.

As an example of this, consider a simple language for failure mode and effects analysis (FMEA) that is actually being developed as an example within the SysML v2 submission team. Using the same approach as was used for defining SysML in the previous section, we first define a model library that captures the basic concepts of FMEA (once again, this is a gross simplification of what would be needed for a real FMEA model library):

```
block FMEAItem {
    part situations: Situation[*];
    part causation: Causes[*];
}

block Situation {
    ref referent: Part;
}

block Cause specializes Situation {
    value occurrence: Real[0..1];
}

block Effect specializes Situation {
    value severity: String[0..1];
}

block FailureMode
    specializes Cause, Effect {
    value detectability: Real[0..1];
}

assoc block Causes {
    end cause: Cause[*];
    end effect: Effect[*];
}
```

In this model, an item of an FMEA involves a set of *situations*, each of which pertains to some part of the system under analysis. Situations are divided into *causes* and *effects*, with a *failure mode* being both an effect and a possible cause of other effects. Finally, an FMEA item may also contain a model of the causal links between the situations it is covering.

For instance, suppose we are analyzing the possible failure modes of a glucose meter, which monitors the glucose level of a patient and applies therapy as necessary by pumping glucose from a reservoir to the patient:

```
block GlucoseMeter {
    ref patient: Patient;
    part battery: Battery;
    part pump: Pump;
    part reservoir: Reservoir;
}
```

This analysis includes the following model of the effect of the possible failure of the battery of the meter to recharge:

```
block GlucoseMeterBatteryFMEAItem
    specializes FMEAItem {
    ref meter: GlucoseMeter;

    part 'battery depleted': Cause
        subsets situations {
        ref redefines referent =
            meter::battery;
        value redefines occurrence =
            0.005;
    }
    part 'battery cannot be charged':
        FailureMode subsets situations {
        ref redefines referent =
            meter::battery;
        value redefines detectability =
            0.013;
    }
    part 'glucose level undetected':
        FailureMode subsets situations {
        redefines ref referent = meter;
    }
```

```
    part 'therapy delayed': Effect
        subsets situations {
        ref redefines referent =
            meter::patient;
        value redefines severity =
            "High";
    }

    link :Causes subsets causation
        connect 'battery depleted'
        to 'battery cannot be charged';
    link :Causes subsets causation
        connect
            'battery cannot be charged'
        to 'glucose level undetected';
    link :Causes subsets causation
        connect
            'glucose level undetected'
        to 'therapy delayed';
}
```

As we saw when defining SysML earlier, the resulting model here may be semantically correct, but it is syntactically cumbersome and unfriendly. And, once again, this can be remedied by providing an appropriate surface syntax:

```
item GlucoseMeterBatteryFMEAItem {
    ref meter: GlucoseMeter;

    cause 'battery depleted'
        on meter::battery
        occurs 0.005;
    causes failure
        'battery cannot be charged'
        on meter::battery
        detected 0.013;
    causes failure
        'glucose level undetected'
        on meter;
    causes effect 'therapy delayed'
        on meter::patient
        severity "High";
}
```

This notation is much clearer to a human reader, but its semantics are still given formally by mapping it back to the earlier SysML model, which is, in turn, formally defined by mapping it back to KerML and its core semantics.

5 CONCLUSION

Ongoing work on a pilot implementation for KerML and SysML has already demonstrated the feasibility of each of the three steps of the metasemantic protocol. Unfortunately, the current prototype approach uses different technologies for each of the steps:

1. Model libraries are written in KerML or SysML (or, potentially, in any further domain-specific extension of SysML).
2. Abstract syntax is modeled using current OMG MOF technology and implemented using the Eclipse Modeling Framework.
3. Concrete syntax for textual notation is implemented using the Xtext DSL framework.

For a true metasemantic protocol, however, all of the above steps need to be achievable within a single linguistic framework that is accessible to end users. That is, all the steps need to be achievable using KerML or SysML. This is already the case for the first step, and it will not be conceptually difficult to achieve for the second step, since the modeling capabilities already available in KerML are sufficient to cover the metamodeling capabilities of MOF. However, providing a robust linguistic capability for the third step is more difficult, especially when one includes requirements for both textual and graphical surface notations.

That said, the SysML v2 RFP already has requirements for the language to include a robust capability for defining user views and viewpoints of a model, well beyond the simple capability found in SysML v1. Our plan is to leverage this required capability in order to fill out the final part of the language extension mechanism for SysML v2. The hope is to realize as much of the full vision of a full metasemantic protocol by the time final submissions are due for the SysML v2 RFP in June 2021.

ACKNOWLEDGEMENTS

I would like to thank Andrius Armonas, No Magic Europe/Dassault Systemes, for allowing me to adapt his work on the FMEA example language extension of SysML v2.

REFERENCES

Bock, C., Odell, J., 2011. Ontological Behavior Modeling. *Journal of Object Technology* 10 (3): 1–36, https://doi.org/10.5381/jot.2011.10.1.a3.

Bock, C., Galey, C., 2019 Integrating four-dimensional ontology and systems requirements modelling. *Journal of Engineering Design* 30 (10–12): 477–522, https://doi.org/10.1080/09544828.2019.1642461.

BPMN, 2014. Business Process Model and Notation (BPMN), Version 2.0.2, Object Management Group, https://www.omg.org/spec/BPMN/2.0.2.

Kiczales, G., des Rivières, J., Bobrow, D. G., 1991. *The Art of the Metaobject Protocol,* The MIT Press, Cambridge.

SysML, 2018. *OMG Systems Modeling Language (OMG SysML), Version 1.6,* Object Management Group, https://www.omg.org/spec/SysML/1.6/.

SysML v2, 2017. *Systems Modeling Language (SysML) v2 RFP,* Object Management Group,
https://www.omg.org/cgi-bin/doc.cgi?ad/2017-12-2.

UML, 2017. *OMG Unified Modeling Language (OMG UML), Version 2.5.1,* Object Management Group, https://www.omg.org/spec/UML/2.5.1.

Voelter, M., 2018. Fusing Modeling and Programming into Language-Oriented Programming. In *Leveraging Applications of Formal Methods, Verification and Validation. Modeling.* 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I, LNCS 11244, Springer Nature, Switzerland, https://doi.org/10.1007/978-3-030-03418-4_19.